SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 3248

Izgradnja zaštićene okoline za izvršavanje i praćenje nepoznatih aplikacija

Frane Kurtović

Zagreb, lipanj 2013.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

SADRŽAJ

1. Uvod **1**
1.1. Motivacija 1
1.2. Pregled rada 1
1.3. Zahtjevi sustava 2

2. Pregled različitih pristupa problemu **3**
2.1. Pristupi izgradnji sigurne okoline 3
2.1.1. Zabrana sistemskih poziva 3
2.1.2. Virtualizacijske tehnologije 3
2.2. Pristupi mjerenju potrošenih resursa 4
2.2.1. wait4() sistemski poziv 4
2.2.2. Praćenje /proc datotečnog sustava 4
2.2.3. Složeniji alati za mjerenje 5

3. Dizajn evaluatora **6**
3.1. Evaluator 6
3.2. Perf 7

4. Implementacija i testiranje evaluatora **8**
4.1. Evaluator 8
4.2. Perf 10

5. Zaključak **12**

Literatura **13**

A. Postovi s bloga **14**
A.1. Linux sandbox 14
A.2. Evaluator for algorithm implementations 16

A.3. LXC Python bindings 18
A.4. Implementation of evaluator and measurment tools 21
A.4.1. Initial version of perf 21
A.4.2. Initial version of evaluator 22
A.5. Design and implementation of online judge system 24
A.5.1. Notifying workers about new tasks 25
A.5.2. Permdata service 25
A.6. Assessing compiler safety 27
A.7. Calling eventfd from Python 28

1. Uvod

1.1. Motivacija

Motivacija za izradu zaštićene okoline pronalazim u postojećim sustavima za algoritamska natjecanja, poput `http://www.spoj.pl`. Za Hrvatska natjecanja postoje već neki sustavi, ali svi su pomalo zastarjeli ili manjkavi. Zadaci na takvim natjecanjima su osmišljeni tako da natjecatelj mora napisati rješenje u nekom od ponuđenih programskih jezika, te se ulazni podaci nalaze na standardnom ulazu, a izlazni podaci se ispisuju na standardni izlaz. Takvo rješenje se na zadanim test primjerima mora izvršavati unutar zadanih vremenskih i memorijskih ograničenja. Problem koji se javlja je izvršavanje korisničkog rješenja na poslužitelju jer je taj programski kod nepoznat, tj. može sadržavati razne propuste ili čak pokušati naštetiti računalu na kojem se izvodi. Za takve aplikacije je praktički pravilo da se izvode na Linuxu, tako da će i sva opisana rješenje biti usko vezana za Linux programsko okruženje.

1.2. Pregled rada

U poglavlju 2 su opisani mogući pristupi rješenju pomoću postojećih tehnologija i alata. Zatim se u poglavlju 3 opisuje rješenje u odabranoj tehnologiji, te se u poglavlju 4 objašnjavaju detalji implementacije dvaju razvijenih alata u jezicima Python i C, te se prikazuju primjeri njihovog korištenja u nekim situacijama. Na kraju rada su dodaci preuzeti s bloga `http://sgros-students.blogspot.com/search/label/sandbox` koji su nastali tijekom proučavanja i izrade ovog evaluacijskog sustava. Oni su nastali na blogu i mogu biti od interesa široj publici, te su zato pisani na engleskom.

1.3. Zahtjevi sustava

Iz prethodnog teksta se već daju naslutiti neki zahtjevi, ali je dobra ideja sve zapisati na jednom mjestu, jer neki možda i nisu toliko očiti. Prvo ćemo zahtjeve podijeliti na:

1. sigurnosne zahtjeve — njima se traži od sustava da pokrene aplikaciju na siguran način, bez da ikako može pristupiti ostatku računala, te da ju ugasi nakon nekog zadanog vremena.

    (a) Onemogućavanje pristupa cijelom datotečnom sustavu, već samo jednom ograničenom dijelu, npr. dijeljenim bibliotekama jer se bez njih većina programa neće moći pokrenuti.

    (b) Onemogućavanje pristupa drugim procesima.

    (c) Gašenje procesa i sve njegove djece nakon zadanog vremenskog intervala.

    (d) Ograničavanje ukupne memorijske potrošnje procesa, tj. ukupnog memorijskog adresnog prostora, kako ne bi preopteretio sustav ili jednostavno potrošio više memorije nego li mu je namijenjeno.

    (e) Ograničavanje broja procesa i dretvi koje je moguće stvoriti. Vrlo često će se ova vrijednosti koristiti za potpunu zabranu stvaranja dodatnih procesa i dretvi.

2. zahtjeve praćenja aplikacije — njima se traži od sustava da izmjeri točno koliko resursa je proces potrošio. Najbitniji resursi za mjerenje su količina potrošene memorije i vrijeme izvršavanja procesa.

Uz sve ove zahtjeve, bitno je napomenuti da razvijeno rješenje ne smije stvoriti veliku nadgradnju u usporedbi s neograničenim izvođenjem istog koda.

2. Pregled različitih pristupa problemu

Kao što su postojala dva skupa zahtjeva, tako će se i pristup problemu odvojiti u dva smjera, jedan kojemu je cilj izgradnja sigurne okoline za izvođenje aplikacije, te drugi kojemu je cilj mjerenje potrošenih resursa aplikacije.

2.1. Pristupi izgradnji sigurne okoline

2.1.1. Zabrana sistemskih poziva

Jedan od postojećih pristupa je zabrana određenih sistemskih poziva, ili ekstremnije, samo propuštanje onih s liste dopuštenih sistemskih poziva. Takva rješenja koriste ptrace() sistemski poziv u Linuxu. Pomoću njega jedan proces može nadzirati drugi tako da prije i poslije svakog sistemskog poziva kojeg obavlja praćeni proces se obavijesti proces pratitelj. Ovime je moguće jako dobro kontrolirati proces, ali je teško odrediti točno koje sistemske pozive dozvoliti i u kojem trenutku. Pri pokretanju praćenog procesa se događaju razni sistemski pozivi koji ne bi trebali biti dozvoljeni za vrijeme "normalnog izvođenja" procesa. I tom problemu se može doskočiti tako da se tek nakon neke inicijalne sekvence sistemskih poziva oni počnu filtrirati.

Ovaj pristup je donekle i prihvativ za jezike koji se prevode direktno u strojni kod. Ali postoji cijeli skup interpretiranih jezika, ili onih koji se izvode u virtualnim mašinama, poput Pythona ili Jave. Za njih je puno teže odrediti koje će uopće sistemske pozive oni pokušati koristiti u svom radu. Dakle ovaj pristup je prerestriktivan, te bismo htjeli pronaći neko drugo općenitije rješenje.

2.1.2. Virtualizacijske tehnologije

Pokretanje aplikacije unutar neke virtualne mašine je moćno rješenje jer je aplikacija u potpunosti odvojena od ostatka operacijskog sustava, ali dolazi uz veliku cijenu smanjenja performansi. U zadnjih nekoliko godina pojavili su se Linux containeri, skra-

ćeno **lxc**. Oni spadaju u lagane virtualizacijske tehnologije jer ne emuliraju sklopovlje računala, već procese pokreću u odvojenim prostorima imena, te tako odvojeni procesi ne mogu nikako referencirati neku strukturu operacijskog sustava koja nije unutar njihovog prostora imena. Virtualizirani proces ima pristup istom kernelu kao i ostatak operacijskog sustava, tako da nije moguće virtualizirati neki drugi operacijski sustav, ali u ovom slučaju to nije niti potrebno. Lxc ima ugrađenu i podršku za cgroups [1], kojima je moguće pratiti i limitirati resurse koje troši grupa procesa. Ovdje je automatski ta grupa procesa ona koja je odvojena u linux container (zasebni prostor imena). Iz svega navedenog ovaj način izgradnje sigurne okoline za proces je vrlo fleksibilan, i s njim ne dolaze smanjenja performansi.

Iz svega navedenog za izradu zaštićene okoline je odabrana lagana virtualizacijska tehnologija — lxc.

2.2. Pristupi mjerenju potrošenih resursa

Mjerenje vremena koje je proces potrošio moguće je ostvariti jednostavno računanjem razlike u vremenu netom prije pokretanja procesa i netom nakon gašenja procesa, no to ne daje točne podatke o tome koliko se stvarno proces izvršavao jer je procesor dijeljen između više procesa u operacijskom sustavu. Iz tog razloga je potrebno upotrijebiti neke naprednije metode mjerenja.

2.2.1. wait4() sistemski poziv

wait4() sistemski poziv čeka da zadani proces završi, te predaje strukturu podataka koja sadržava razne informacije o tome koliko je proces potrošio kojih resursa tijekom svog izvođenja. Minus kod ovog pristupa je što dobiveni podaci sadrže informacije o potrošnji samo tog procesa, a ne i njegove djece.

2.2.2. Praćenje /proc datotečnog sustava

Linux ima poseban datotečni sustav koji se nalazi u /proc direktoriju, te sadrži razne podatke o trenutnom stanju operacijskog sustava. Tako postoji i direktorij za svaki proces koji trenutno postoji u operacijskom sustavu. On sadrži podjednako informacija o procesu kao što ih vraća i već opisani sistemski poziv wait4(). Kada proces završi, on nestaje iz /proc direktorija, te bi to značilo da je potrebno konstantno pratiti datoteke

vezane uz praćeni proces, te zapisivati vrijednosti koje nas zanimaju. Pošto nas zanima koliko je proces maksimalno memorije trošio u nekom trenutku, to bismo trebali zapisivati svako T milisekundi. Takav pristup nije dobar jer ako želimo precizne podatke moramo jako smanjiti interval provjere T, te time dodatno opteretiti operacijski sustav.

2.2.3. Složeniji alati za mjerenje

Postoje i složeniji alati: perf [5], systemtap [7], lttng [2] i drugi. Pomoću njih je moguće ostvariti puno kompliciranije zadaće od traženih, te nisu intenzivno proučavani zbog svoje složenosti.

wait4() sistemski poziv je odabran radi svoje jednostavnosti i jer ne utječe na performanse — jedini mu je minus što ne može mjeriti sumiranu potrošnju svih procesa, ali to nije toliki problem jer će se ovaj evaluator prvenstveno koristiti za pokretanje programa koji ne smiju stvarati druge procese.

3. Dizajn evaluatora

Prije smo govorili o dva skupa zahtjeva sasvim odvojeno, oni koji se brinu oko sigurnosti, te oni koji prate koliko je aplikacija potrošila resursa. U ostvarenju sustava će se ti dijelovi često ispreplitati te neće biti baš toliko odvojeni kao prije. Od sada razvitak sustava dijelimo na dva dijela: onaj koji stvara i pokreće aplikaciju unutar lxc kontejnera, te onaj dio koji će biti pokretan unutar samoga kontejnera, te koji zapravo pokreće traženu aplikaciju, te radi neka mjerenja i ograničenja unutar samoga kontejnera.

Napravljena su dva alata koja se koriste u komandnoj liniji. Prvi se zove *evaluator* i on pokreće aplikaciju unutar kontejnera, a drugi zvan *perf* samo pokreće aplikaciju, primjenjuje neka ograničenja na nju i stvori datoteku s mjerenjima.

3.1. Evaluator

Evaluator očekuje da na disku već postoji kontejner pod nazivom *sandbox* i uvijek pokreće zadani program unutar tog kontejnera. Primjer komandne linije sa svim podržanim opcijama.

```
./evaluator -m 65000000 -c 2000 ./user_prog user_arguments
```

Parametar -m se tumači kao ograničenje na ukupnu memoriju u oktetima koju svi procesi unutar stvorenog kontejnera u sumi smiju koristiti. Ta memorija uključuje i virtualnu memoriju (swap), tj. cjelokupni adresni prostor procesa.
Parametar -c se tumači kao maksimalno vremensko ograničenje za dani proces. Ako proces ne završi unutar zadanog broja milisekundi, onda ga se gasi na silu.
Prvi parametar koji ne odgovara niti jednom od navedenih se tumači kao putanja do procesa kojeg se želi pokrenuti unutar kontejnera.
Svi ostali parametri se ne konzumiraju, već se samo predaju procesu kojeg pokrećemo unutar kontejnera.

Nakon završetka rada, evaluator stvara datoteku s rezultatima evaluacije. U toj datoteci se nalazi:

1. je li pokrenuti proces prekoračio vremenski limit, tj. je li ga bilo potrebno zaustavljati ili je sam izašao
2. vrijeme koje je proteklo od pokretanja do zaustavljanja procesa
3. je li prekoračen definirani memorijski limit.
4. izlazni kod pokrenutog procesa

3.2. Perf

Perf za zadaću ima smanjiti ovlasti procesa, postaviti ograničenja na memoriju, vrijeme izvođenja te na broj procesa i dretvi koje proces smije stvoriti. Perf čeka da proces završi, te u datoteku zapisuje podatke o potrošenoj memoriji i o vremenu izvršavanja. Uočite da perf radi sasvim nevezano za linux containere.

```
./perf -m 65000000 -c 2 -t 2 ./user_prog user_arguments
```

Parametar -m se tumači kao ograničenje na ukupnu memoriju u oktetima koje nasljeđuje svaki proces. Ta memorija uključuje i virtualnu memoriju (swap), tj. cjelokupni adresni prostor procesa.
Parametar -c se tumači kao maksimalno vremensko ograničenje za dani proces.
Parametar -t se tumači kao ukupan dozvoljeni broj procesa i dretvi koje proces smije stvoriti. U taj broj je uključen i sam proces.
Prvi parametar koji ne odgovara niti jednom od navedenih se tumači kao putanja do procesa kojeg se želi pokrenuti.
Svi ostali parametri se ne konzumiraju, već se samo predaju procesu kojeg pokrećemo.

4. Implementacija i testiranje evaluatora

4.1. Evaluator

Za implementaciju evaluatora je odabran Python 3 jer lxc 9.0 dolazi s bibliotekom za Python, te je njeno korištenje detaljno opisano u dodatku A.3. Tamo je prikazano kako stvoriti kontejner koristeći predložak *busybox*. Za izgradnju kontejnera kojeg koristi evaluator korišten je izmijenjeni, ili bolje reći pojednostavljeni predložak *busybox*. Taj predložak zapravo samo stvara datotečnu strukturu kontejnera. Koristi se busybox-ov init proces, te su u /bin direktoriju samo busybox-ovi standardni alati. Najbitnije je da su /lib i /usr/lib prazni direktoriji, te se tek kod podizanja kontejnera u njih bind mountaju isti ti direktoriji, samo od stvarnog operacijskog sustava. Tako je moguće pokrenuti svaki program kojeg je moguće pokrenuti i izvan kontejnera. Također se stvara i /tmp folder koji je prazan, te se koristi za spremanje programa koje potrebno pokrenuti unutar samoga kontejnera.

Program je koncipiran tako da postoji glavna dretva i dretva koja pokreće dijete u kontejneru. Na dijete dretvu se čeka najviše T milisekundi, te se nakon toga gasi kontejner, a ako se nakon sekundu ne ugasi regularno onda se šalje signal *SIGKILL* svakom procesu u njemu. Nakon svega se u datoteku s rezultatima zapiše izlazni status djeteta, te koliko se dugo izvršavalo dijete. Još jedino preostaje implementirati neki način za pratiti je li kontejner prekoračio memorijsko ograničenje. Za to se koristi već spomenuti cgroups memorijski kontroler. On omogućava ne samo limitiranje memorije, već omogućava i obavještavanje da je neka predefinirana razina memorije u kontejneru dosegnuta. Zato postoji još jedna dretva koja čeka na notifikaciju od memorijskog kontrolera. Više o tome piše u dodatku A.4, te kako se poziva potrebna sistemska funkcija *eventfd()* iz Pythona objašnjeno je u dodatku A.7.

Naravno, potrebno je i testirati radi li sva navedena funkcionalnost. Zato je osmišljen program *mem_consumer* koji u beskonačnoj petlji svaki put alocira niz od 4 kb,

i nakon svake uspješne alokacije ispisuje ukupnu alociranu sumu. Ovime možemo testirati i funkcionalnost memorijskog limita i vremenskog limita.

- Vremenski limit — postavimo vremensko ograničenje na jednu sekundu, a memorijsko na 128 mb. Program ne bi trebao stići alocirati toliko memorije u tom vremenu, te bi izlaz trebao pokazivati da je vremensko ograničenje prekoračeno. Doista zadnja linija koju ispiše *mem_consumer* je "Allocated so far 33276 kb", a datoteka results.txt sadrži:

  ```
  True
  1.0041999816894531
  False
  255
  ```

  Prva linija je True, što znači da je vremensko ograničenje prekoračeno, kao što je bilo i za očekivati.

- Memorijski limit — postavimo vremensko ograničenje na pet sekundi, a memorijsko na 32 mb. Program će ovaj put stići alocirati dovoljno memorije da prekorači memorijski limit, te bi izlaz trebao pokazivati da je memorijsko ograničenje prekoračeno. Doista zadnja linija koju ispiše "Allocated so far 31216 kb", a datoteka results.txt sadrži:

  ```
  False
  1.3221566677093506
  True
  255
  ```

  Treća linija je True, što znači da je memorijsko ograničenje prekoračeno, kao što je bilo i za očekivati. Iz zadnje linije ispisa se vidi da je uspio alocirati 31 965 183 okteta, a ograničenje je postavljeno na točno 32 000 000 okteta. Razlika je 34 816 okteta, što je također očekivano jer *mem_consumer* nije jedini proces u kontejneru, već postoji i *init* proces koji također ima neko memorijsko zauzeće.

- Normalno izvođenje — ovaj put bismo htjeli pokrenuti program koji će se uspješno izvesti unutar kontejnera, za to ćemo odabrati alat *ls*. Postavimo vremensko ograničenje na pet sekundi, a memorijsko na 32 mb. Doista program je ispisao sadržaj direktorija, a datoteka results.txt sadrži:

  ```
  False
  ```

```
0.06973433494567871
False
0
```

Sve linije su false, program se izvršavao jako kratko, te je izlazni kod 0.

4.2. Perf

Već je rečeno da perf radi nezavisno od linux kontejnera. Njegove zadaće su redom:

1. Promijeniti korisnika u nekog koji još ne postoji na sustavu koristeći *setuid()* sistemski poziv. To je potrebno jer root-u nije moguće postavljati nikakva ograničenja i zato što se neka ograničenja obračunavaju u sumi za sve procese pojedinog vlasnika.

2. Stvoriti novi proces koristeći *fork()* sistemski poziv, koji kasnije pokreće zadani program koristeći *execvp()*.

3. Nakon što je stvoren novi proces, postavljaju se limiti na njega koristeći *setrlimit()* sistemski poziv. On provodi ograničenja na vrijeme izvršavanja, ukupnu dozvoljenu memoriju i ukupan dozvoljeni broj procesa, te također onemogućava stvaranje core dumpova.

4. Glavni program čeka na završetak djeteta koristeći *wait4()*, te u datoteku results.txt sprema izlazni status djeteta, vrijeme provedeno u korisničkom načinu, vrijeme provedeno u sistemskom načinu te najveću memorijsku potrošnju u kilobajtima.

I kod ovog programa će biti testirani slični slučajevi kao i kod prošlog, samo što će se još provjeriti i ponašanje ograničenja broja procesa.

– Memorijski limit — postavimo vremensko ograničenje na pet sekundi, a memorijsko na 32 mb. Program će stići alocirati dovoljno memorije da prekorači memorijski limit. Doista, zadnje koliko je proces uspio alocirati je 29116 kb, te je ispisao pogrešku da ne može više memorije alocirati. Datoteka results.txt sadrži:

```
0
0.012000
0.268000
29564
```

Prva linija je izlazni kod, druge dvije redom pokazuju vrijeme provedeno u korisničkom načinu rada, te vrijeme u sistemskom načinu. Zadnja linija pokazuje najveću memorijsku potrošnju od 29564 kb, što otprilike odgovara zadnjoj uspješnoj alokaciji. Primijetite da ovaj pristup ne daje informaciju o tome je li memorijski limit prekoračen.

– Ograničenje na broj procesa — ovaj put se pokreće program koji stvara novi proces, a ograničenje na broj procesa je 1. Program stvarno ne uspijeva napraviti novo dijete, te datoteka results.txt sadrži:

```
1
0.000000
0.004000
436
```

Kao i u prošlom primjeru, nije moguće znati je li proces pokušao stvoriti nove procese te tako prekoračiti ograničenje.

– Terminiranje procesa signalom — cilj je isprobati što se dogodi kada proces primi signal. Testni program će raditi dijeljenje s nulom, za što je poznato da proces dobije signal *SIGFPE* (broj 8). Perf je implementiran tako da za gašenje signalom umjesto izlaznog statusa zapiše negativnu vrijednost signala. Datoteka results.txt sadrži:

```
-8
0.000000
0.020000
452
```

Stvarno prva linija je -8, što znači da je proces prekinut signalom 8, što je signal *SIGFPE*.

5. Zaključak

U ovom radu su opisana i napravljena dva programa koja izoliraju i stvaraju nekakva ograničenja na aplikaciju. Moguće ih je pronaći na githubu `https://github.com/kfrane/evaluator`. Prvi zvan evaluator radi tako da pokrene aplikaciju unutar linux kontejnera čiji je osnovni princip da odvajaju prostore imena. Drugi program radi jednostavnije, tako da samo ograničava resurse koje aplikacija smije koristiti. Ovi programi su dizajnirani tako da *evaluator* može pokretati bilo koji program, pa tako i ovdje razvijeni *perf*, a onda on može pokretati stvarni program koji se želi ograničiti. Time je pružena zaštita u dva sloja. Također oba dva programa stvaraju datoteku *results.txt* koja sadrži informacije o izvođenju programa.

Smatram da je ovime postignuta dovoljna sigurnost za potrebe algoritamskih natjecanja. Prostor za nadogradnju ovog sustava još uvijek postoji. Jedna od mogućnosti je nadograditi *perf* tako da filtrira neke sistemske pozive, kao što je opisano u poglavlju 2.1.1.

LITERATURA

[1] Cgroups documentation. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt. [Online; pristupano 25-Travanj-2013].

[2] Lttng documentation. https://lttng.org/. [Online; pristupano 10-Svibanj-2013].

[3] lxc linux containers. http://lxc.sourceforge.net/. [Online; pristupano 25-Travanj-2013].

[4] Moe contest environment. http://www.ucw.cz/moe/. [Online; pristupano 15-Ožujak-2013].

[5] Perf documentation. https://perf.wiki.kernel.org. [Online; pristupano 10-Svibanj-2013].

[6] Ptrace documentation. http://linux.die.net/man/2/ptrace. [Online; pristupano 20-Ožujak-2013].

[7] Systemtap documentation. http://sourceware.org/systemtap/. [Online; pristupano 10-Svibanj-2013].

[8] Michael Kerrisk. *The Linux programming interface*. 2010.

Dodatak A
Postovi s bloga

A.1. Linux sandbox

I am trying to build an evaluation system for algorithmic problems. The user will have to solve a problem which receives it's input data through standard input and prints a solution to standard out.

The goal is to solve the given problem within some predefined memory and time constraints. When a user thinks he solved a problem, he submits it for evaluation.

There are two problems when evaluating users source code:

1. Execution of the compiled source code must be done in an isolated way (jailing the application), such that the executable can't harm the host (operating system that is running user process). There are several ways how the process could harm the host: use all available memory or disk space, exceed maximum number of threads or processes on the system (or other kernel structures), run for a very long time (infinite loop), access files it isn't supposed to (maybe a solution file).

2. Measuring the resources that the application used during it's execution, such as time and memory (heap, stack).

This post will concentrate on the approaches to solve the first step – making a jail.

1. Monitor every system call a process makes using the ptrace() [6] system call. This enables the process called tracer to be notified before and after every system call his tracee makes. At this point the tracer can decide whether to allow it or not. Problem is how to decide which system calls are allowed, and this list changes during the tracees execution. Beginning sequence of system calls contains dangerous system calls that must be disabled when the process is completely

loaded into the memory. Ptrace() also enables inspecting every signal that is received by the tracee, and is able to read tracees memory. These two features will probably not be used in this approach.

This approach works well for languages that are translated to machine code, because they are executed directly, but Java and Python (and many others) are problematic because they are executed through a different executable (virtual machine or interpreter). This means that these executables would have to be traced, and this is not possible, because they often do forbidden things, like reading files, creating multiple threads. We wouldn't be able to determine if the users code did this, or virtual machine by itself. There are also workarounds for that problem, like compiling java with gcj to machine code or python with various python to C++ compilers (nuitka, cython). Thus, the main concern is difficulties with adding support for another language.

2. Try putting the application in a container that has no means of interacting with the rest of the system, which is accomplished using different namespaces for process IDs, network, IPC structures (semaphores, shared memory...), file system mount, UTS (system name - uname), user and group ids. If a process can't reference anything outside of the jail than it isn't able to do any harm. Cgroups are way of limiting resources of a process group. These two mechanisms are used by various tools that implement application jailing, one of these tools is lxc [3]. The approach with the lxc is more flexible, because it doesn't limit applications inside the container in any way, since they can't do any harm to the outside world, maybe only crash the jail they are in. If the applications aren't limited, then they can be executed using a support program, like java virtual machine or python interpreter, which wasn't possible in the first approach.

The second approach using namespace separation and cgroups is a better way to go, because it is more extensible than the first approach using ptrace, primarily in terms of adding support for another language.

The Moe contest environment [4] is an already existing solution to this problem. It has two modes of isolation. The old one is ptrace, and the new one is using linux namespaces. Lxc was built on top of the linux namespaces and that makes these two approaches similar.

These are the reasons for studying lxc in more detail.

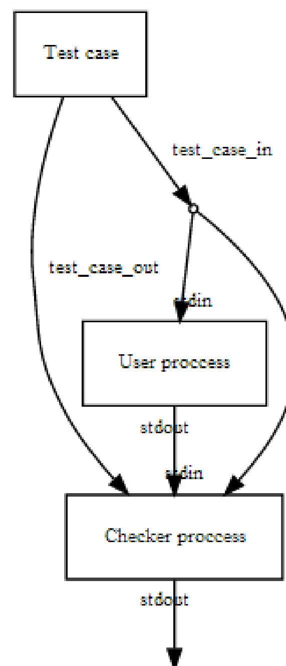
A.2. Evaluator for algorithm implementations

Algorithm tasks evaluator is a system that runs untrusted user process on test data and outputs its correctness.

Test data is a pair of input and output file.

User process reads that input file from standard input and writes the solution to standard output. The task it solves is of an algorithmic nature and must complete in some predefined time and consume no more than some predefined amount of memory (RAM). This means it doesn't have access to disk or network or anything other outside that process, note that this also means it mustn't create any new processes.

There is another component called checker used for determining whether the output produced by user process is a correct output for the given test data. This means it uses three sets of data to determine the correctness of a solution: input file from test data output file from test data solution from user process that had been written to the standard output The need for checker may not be so obvious, but it is very simple, there may be multiple correct solutions for some tasks. Lets consider a task of calculating π constant with the precision of 10^{-5}. This means that the correct answer is everything from the interval of $[\pi - 10^{-5}, \pi + 10^{-5}]$, and this check is the checkers responsibility.



Slika A.1: Evaluator architecture

We need to have in mind that the user process and checker can not be trusted, which

means they need to run in a limited environment, such as a sandbox that was discussed in a previous post[1].

This system will be implemented using python and python lxc library which is available under the latest lxc version.

[1] http://sgros-students.blogspot.com/2013/03/linux-sandbox.html

A.3. LXC Python bindings

The latest lxc version (0.9.0) has Python bindings included. You can download the latest version from github or install it from standard repositories on Ubuntu 13.04. If you have chosen to install it from source then don't forget to run configure with –enable-python flag. The library has been written for Python 3.

Good example of usage is in the api_test.py[2] file, and following code snippets will show how to create, start, stop and destroy container named test_container. This library requires root permissions, so don't forget to run Python as root.
In the following code we instantiate lxc container.

```
import lxc
container = lxc.Container('test_container')
```

First we have to import lxc library. Next line instantiates new container with container name test_container. It doesn't have to exist yet, but it can. The constructor can take one more argument called config_path, which is a path to the configuration file for the container. If it is not specified, the default configuration will be used, usually located in /etc/lxc/default.conf[3].
Here we show how to create a container from template.

```
container.create('busybox')
```

If it doesn't exist yet, you can create it using create method and specify which template will be used to create it. Creating a container is just creating a rootfs folder for the container, and creating a configuration file. They will be located in /var/lib/lxc/test_container/. There are several templates such as, ubuntu, fedora, debian, sshd. I chose busybox because it is light weight, only few megabytes, while ubuntu would start downloading ubuntu minimal package which is over 300 megabytes. Container templates are just scripts that create rootfs folder for container. If custom behavior is needed, then a new script can be created or existing modified.
This code shows how to start a container.

```
container.start()
container.wait("RUNNING", 5)
print ("Container state", container.state)
```

The start method starts the container in daemonized mode and doesn't block the execution, that is why the wait method is used to wait at most 5 seconds for the container

[2]https://github.com/lxc/lxc/blob/staging/src/pythonlxc/examples/api_test.py
[3]https://github.com/lxc/lxc/blob/staging/config/default.conf.ubuntu

to reach the running state. Output of this script should just be a line saying that the container state is running, if not then something is wrong with the container or the lxc installation.

There are two optional arguments that the start command expects:

- cmd argument which expects a tuple. If specified then the default init process is replaced with that command. Note that this resembles to lxc-start command, because it can also replace standard init process with the command specified after the – flag. The first element of the tuple should be a path to executable, relative to containers rootfs, and the rest is passed as command line arguments to it. When using it with ubuntu template the start method sometimes prints an error message, even do it successfully runs the application and returns true.
- useinit argument uses the lxc-init executable as init process, this is used to run a command in a similar way the lxc-execute runs it. I have not been able to successfully run a container with useinit set to true.

Here we show how to stop a container.

```
container.stop()
container.wait("STOPPED", 5)
print ("State should be stopped", container.state)
```

Stop method takes no arguments and stops the container without blocking the execution, and this is why wait method has to be used to reach the stopped state. This is very similar behavior to the start method.
This line shows how to destroy a container.

```
container.destroy()
```

Destroy method removes the container and its rootfs from the disk. To use it again one would have to create it with create method. This method is implemented just by calling lxc-destroy command.

These were the basic functions to operate the container, but there are still few more interesting functions to look at.
The following code shows how to access container console.

```
container.console()
```

Once the container had started and reached the running state its console can be accessed with console method. This will bring the console and will prompt for username and password.
This code snippet shows to attach a single process to an already running container.

```
container.attach('ALL', '/bin/ls', '-l')
```

This runs the command inside an already running container. First argument regulates which namespaces to attach to, ALL is the default and safest choice, but if you want to specify particular namespaces just enter it as OR-ed list of arguments such as 'NETWORK|IPC'. The second argument is path to the command that should be attached to the container, and the rest of the parameters are passed to it as command line arguments. This command is useful for running just one command, and it doesn't close its file descriptors. This means that the stdin, stdout and stderr are still open and available to use.

All of these methods resemble the lxc-* commands in the command line. To find more information about them use man pages, and to find some info on how to use them inside Python just call help(lxc).

A.4. Implementation of evaluator and measurment tools

The idea is to create two command line tools. The first one, which will be called evaluator is responsible for running another process in a safe and confined environment. It will run the untrusted program inside an lxc container. It is also going to be configurable with parameters such as maximum allowed memory usage and maximum execution time. These limits refer to the whole container and not just the untrusted process. This is why the container will be made to consume the least amount of memory and cpu possible. Its command line will look like this:

```
./evaluator -m 64000000 -c 2000 ./user_prog user_params
```

This will run the user_prog inside an lxc container and pass to it all of the user_params. It will also limit the container's memory usage to 64MB and execution time to 2000 ms.

The second tool, called perf, is responsible for enforcing some limits on the untrusted program and also for measuring performance of the untrusted program. It has similar format of command line as the evaluator:

```
./perf -m 64000000 -c 2 -t 2 ./user_prog user_params
```

This will execute file ./user_prog and pass it user_params as command line arguments. It will limit its memory usage to 64MB, and set time limit of each thread to 2 seconds and set the allowed number of threads to 2. Perf will also create file named measurements, which will contain information about how much memory did the user_prog use, and what was its execution time.

These two tools are intended to be used together, as in the example below:

```
./evaluator -m 65000000 -c 2000 ./perf -t 1 ./user_prog
```

It will run perf inside an lxc container which has memory limit of 65MB, and if after 2000 ms it hasn't stopped, it will be forced to stop. Perf will then run user_prog and limit the number of allowed threads to 1, that is, the user_prog won't be able to create threads or processes.

A.4.1. Initial version of perf

I have chosen to write perf in C, because it has the most direct access to the necessary system calls. Arguments are being parsed by getopt() library function, and the parsing

stops when the first non option argument is detected. That argument is assumed to be user_exec and everything after it is passed to it as command line arguments.

Limiting the child process is implemented by setrlimit()[4] system call, and not by ulimit()[5] because it is obsolete according to documentation.

The following limits are being used:

- RLIMIT_AS - for virtual memory, the total address space of the process. The value is extracted from the '-m' option. Soft and hard limits are set to the same value, because we won't the process to stop as soon as possible.
- RLIMIT_CORE - size of the core dumps. It is always set to 0, which just means that no core dumps will be created.
- RLIMIT_NPROC - limit on the number of processes, or more precise threads and it is extracted from '-t' option. This limit is per user, by looking at real user id. This is why the perf will have to change user id to something else.
- RLIMIT_CPU - sets cpu time limit in seconds, extracted from '-c' option.

A.4.2. Initial version of evaluator

Evaluator is implemented in Python 3, because it needs to operate with lxc container which only has Python 3 bindings.

- Memory limit, which is extracted from '-m' option, is the allowed memory for the whole container, and it is enforced by cgroups memory controller.
- Time limit, which is extracted from '-c' option, is implemented by a join which has timeout equal to specified time limit.

Cgroups memory controller is only available by default on the newest linux distributions, some of the older distributions can enable it by installing from backports or by recompiling kernel. Even if it is enabled the swap accounting isn't turned on, and therefore the memory limits do not account for swap space. To enable it, a kernel boot parameter 'swapaccount=1' is needed. It can be added to the grub configuration file.

Cgroup memory controller's notification api is used to alert the application that memory limit has been crossed. It uses a relatively new eventfd()[6] system call to receive notifications. There isn't any python module that wraps this system call, so I

[4]http://linux.die.net/man/2/setrlimit
[5]http://linux.die.net/man/3/ulimit
[6]http://linux.die.net/man/2/eventfd

had to call it directly from glibc library file (libc.so.6). Ctypes module is used to load the library into python, and to call eventfd() from it.

It creates file results.txt, which contains information about crossing the memory and time limits. Its structure is as follows:

- First line is either True or False. True stands for time limit exceeded, and false otherwise.
- Second line is what was the actual time execution, it is just a difference between timestamps.
- Third line is either True or False. True stands for memory limit exceeded, and false otherwise.

A.5. Design and implementation of online judge system

My previous blog posts were describing how to safely compile and run an untrusted user code. They also showed that the user process will only read from standard input and write to standard output. Pair of official input and output data is called test case. Test case also has a checker program associated with it, which compares user output with official test case, and decides if the output is correct. This is also the smallest processing unit of this judge system.
Central part of this judge system is the dispatcher. It is responsible for:

- receiving new tasks for evaluation from web
- notifying the workers about these new tasks
- receiving the results from workers
- notifying the web about the results of evaluation

Workers responsibility is evaluating a single test case and sending the results back to the dispatcher.

Dispatcher is implemented as postgresql[7] database. It has only one table, named queue, which stores tasks that need to be evaluated. Basic design of this table is shown in figure A.1. Field queue_status indicates what is the status with this task, it can be:

Tablica A.1: queue table

Column	Type
queue_id	integer
queue_status	integer
text_in	text
bin_in	bytea
text_out	text
bin_out	bytea

- 0 — the worker hasn't fetched this task yet
- 1 — worker is evaluating this task
- 2 — task is completed

Text_in and bin_in fields are used to describe what is the workers task. Text_out and bin_out describe the workers evaluation results. Text fields are JSON encoded, their

[7] http://www.postgresql.org/

meaning is only meaningful to the worker and to the web. Bin_in and bin_out are not currently used.

A.5.1. Notifying workers about new tasks

Postgresql database supports LISTEN and NOTIFY commands. Clients that issue LISTEN commands, will be notified when the database issues NOTIFY command. Each worker issues LISTEN for channel 'new_queue_row', and the database is configured to issue NOTIFY 'new_queue_row' when new row is inserted into queue table. When workers fetch this newly inserted row they fetch it inside a transaction, which prevents multiple workers from fetching the same row, this function is called queue_fetch_front().

To make this functionality easier to use, it was separated in a service called waiter. It is a thrift[8] service, which has only one funcition.

```
bool wait(string channel, int timeout_ms);
```

This function waits for a notification on the given channel, and returns true when notification is received. If after timeout_ms notification wasn't received, false is returned. After the worker receives true from the wait() function, it executes queue_fetch_front() function in database to retrieve the new row.

A.5.2. Permdata service

Permdata service is also a thrift service, which runs with every worker. It is used by the worker to retrieve test cases, or any other blob of data from the database. Permdata has its own database with one table that has only three columns, see figure A.2. A.2. Primary key of this table is data_hash, which is md5(blob). Calculating hash was

Tablica A.2: permdata table

Column	Type
data_hash	text
blob	bytea
t_added	timestamp

chosen over incrementing ids, because this way cache can't get invalidated at any time.

[8] http://thrift.apache.org/

Primary reason for making the permdata service is to avoid storing large blobs of data in the queue table. Instead of storing the whole test case, which can be several megabytes of size, we now need to store only its hash, which is 32 characters. Permdata service is also a good place to cache all of the test cases, to avoid multiple transfers of large data over network.
Permdata thrift service has only one function.

```
struct getDataResult {
  1: bool found;
  2: binary data;
}
getDataResult getData(1:string hash)
```

This also means that the worker only has read permissions on the permdata database.

A.6. Assessing compiler safety

When creating a sandboxed evaluator, an interesting question popped up - Can compiling a source code with some of the popular compilers like gcc or javac be harmful to the system? Being harmful is an action that causes the system to crash, or that gives user access to the system, or any action that can harm the system, even though the code is not being executed, but just compiled.

Causing the system to crash is possible while compiling, because compiling is very complicated process by itself, and can use all of the systems resources and possibly crash the system. This is even more true if complicated optimizations are enabled. There is also a possibility of using C++ template language to do complicated computations during the compile time, by forcing the compiler to evaluate complicated or even infinite expressions.

The second way of being harmful, by gaining access to the system is an unexplored subject. I couldn't find any references about it, and that is why I will ignore such threats to the system.

To conclude, the part of an evaluation system that is responsible for compiling the user source code will only be protected from the first threat, when compiler uses too much memory or takes too much time to complete. This will be easily achieved using perf[9] tool, which was explained in the previous posts. Its main purpose is to limit the processes resources, such as memory and cpu time.

[9]http://sgros-students.blogspot.com/2013/05/implementation-of-evaluator-and.html

A.7. Calling eventfd from Python

Eventfd is a new IPC mechanism used for signaling events between processes or between kernel and user space. It has been introduced in Linux since 2.6.22 kernel version, and in glibc since 2.8 version. This is a fairly new system function, and Python does not yet have any function that wraps this system call.

There are two ways one could call eventfd()[10].

1. Create C extension, which makes wrapper function around eventfd(), compile it and then use it from Python. Such extension module can already be found on github[11].

2. Use ctypes[12] module to load shared library file, which contains eventfd() function and then make a call to it. On Linux, library that contains wrappers around system functions is called GNU C Library[13] (glibc), and it can be found through the symbolic link libc.so.6.

The second method is easier, because it doesn't involve compiling and uses only few lines of Python code listed below.

```
from ctypes import *
libc = cdll.LoadLibrary("libc.so.6")
def eventfd(init_val, flags):
    return libc.eventfd(init_val, flags)
```

The first line imports the necessary class, then we load a glibc shared library. Now we can use this instance to call functions from it. This is what we do in the next two lines when we define a wrapper function and call eventfd() from glibc. This function is easy to call, because it takes only integer arguments and returns an integer, so there is no need to use any data types defined in ctypes, which are used to represent basic C types.

With this function created, it is possible to create new eventfd and read or write to it with Linux read and write functions.

```
import struct, os, sys
fd = eventfd(0, 0)
bytes_written = os.write(fd, c_ulonglong(197348329))
if bytes_written != 8:
```

[10] http://linux.die.net/man/2/eventfd
[11] https://gist.github.com/peo3/934279
[12] http://docs.python.org/2/library/ctypes.html
[13] http://en.wikipedia.org/wiki/Glibc

```
    print("Error writing to eventfd")
    sys.exit(1)
num = os.read(fd, 8)
print(struct.unpack('@Q', num)[0])
num = os.read(fd, 8) # this call will block
```

First new eventfd is created, and its file descriptor is stored. Then we use Linux read and write functions, which operate directly through file descriptors. Eventfd accepts 8 bytes of data that are interpreted as unsigned 64 bit integer, and this is why c_ulonglong from ctypes is needed. Next line just checks if all of the 8 bytes were written. Then we read from the same file descriptor, and read the same number that was written to it. What was read is just an array of bytes that needs to be interpreted as unsigned 64 bit integer, and this is why the unpack from struct module is used with format '@Q', which means interpret these bytes as unsigned 64 bit integers in the systems native endianness. Finally, we try to read again. This call blocks because there was no writing to this eventfd since the last read.

Izgradnja zaštićene okoline za izvršavanje i praćenje nepoznatih aplikacija

Sažetak

U ovom radu je opisano kako izgraditi sigurnu okolinu za aplikaciju koja čita samo sa standardnog ulaza, te zapisuje na standardni izlaz. Glavni primjer takvih aplikacija su implementacije algoritama. Također je objašnjeno kako ograničiti resurse koje aplikacija smije koristiti, te kako izmjeriti točno koliko ih je aplikacija iskoristila. U sklopu ovog rada su napravljena i dva programa koja implementiraju opisane postupke iz rada, te se nalaze na `https://github.com/kfrane/evaluator`.

Ključne riječi: linux, sandbox, lxc

Creating secure environment for executing and tracing unknown applications

Abstract

This paper shows how to build secure environment for application that only reads from standard input and writes to standard output. Best example of such applications are algorithm implementations. It also shows how to limit the resources that application is allowed to use, and also how to measure the exact amount used. Two programs were developed as a part of this paper and can be found on `https://github.com/kfrane/evaluator`.

Keywords: linux, sandbox, lxc