Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Maciej Wachulec
Nr albumu: 371845
Przemysław Jakub Kozłowski
Nr albumu: 371120

Tadeusz Dudkiewicz
Nr albumu: 370782
Wojciech Dubiel
Nr albumu: 371280

SIO2Jail
Narzędzie do nadzorowania wykonania programów zgłaszanych w ramach konkursów algorytmicznych

Praca licencjacka
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr. Robert Dąbrowski

Instytut Informatyki

Maj 2017

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpisy autorów pracy

Streszczenie

Streszczenie jak już będziemy mieć całość pracy

Słowa kluczowe

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

Social and professional topics
D.127. Computing education
D.127.6. Student asessment

Spis treści

Wprowadzenie

Obecnie systemy Olimpiady Informatycznej, oraz platforma Szkopuł przyjmują co roku setki tysięcy zgłoszeń będących rozwiązaniami zadań programistycznych. Zgłoszone programy są automatycznie kompilowane, uruchamiane na odpowiednich dla zadania danych wejściowych i oceniane. Podstawą dla oceny programu jest jego czas działania, modelowany przez liczbę wykonanych instrukcji, co zapewnia powtarzalność i sprawiedliwość pomiaru. Dodatkowo, na uruchamiane programy nakłada się inne ograniczenia, np: rozmiar dostępnej pamięci, brak możliwości łączenia z siecią, tworzenia procesów potomnym, korzystania z plików.

Na dzień dzisiejszy do sprawdzania rozwiązań Olimpiada Informatyczna wykorzystuje narzędzie OITimeTool. Zostało ono stworzone ponad 6 lat temu przez Szymona Acedańskiego. Narzędzie to korzysta z biblioteki Pin firmy Intel, która "(...)przepisuje kawałki kodu maszynowego przed pierwszym ich wykonaniem, dodając w odpowiednie miejsca elementy instrumentacji, oraz w minimalnym możliwym stopniu modyfikuje oryginalne wywołania. "[1]. Dzięki temu możliwe jest zliczanie wykonywanych instrukcji. W OITimeTool zapewnienie izolacji ocenianego programu odbywa się poprzez ograniczanie dostępnych wywołań systemowych.

Narzędzie OITimeTool nie jest konfigurowalne, zestaw obsługiwanych języków programowania jest mały i dodanie kolejnych wymaga dużego nakładu pracy, co ogranicza zestaw konkursów możliwych do zorganizowania. Ponadto programy uruchamiane pod kontrolą OITimeTool dzielą z nim przestrzeń adresową, co jest zagrożeniem bezpieczeństwa i powoduje problemy administracyjne - trudno jest odróżnić błąd OITimeToola od błędu sprawdzanego programu.

W międzyczasie pojawiło się wiele nowych rozwiązań: technologia perf pozwalająca na sprzętowe liczenie instrukcji, oraz dużo mechanizmów pozwalających na izolację i ograniczanie uprawnień programów. W związku z tym możliwym stało się stworzenie nowego, zmodernizowanego narzędzia rozwiązującego część problemów z którymi boryka się OITimeTool. Zaprojektowanie, zaimplementowanie i zintegrowanie z istniejącymi systemami Olimpiady Informatycznej alternatywy jest tematem niniejszej pracy. Zapraszamy do lektury.

Rozdział 1

Dostępne technologie

Jądro Linuxa udostępnia wiele mechanizmów związanych z izolacją procesów oraz pomiarem wydajności, które zostały lub mogłyby zostać wykorzystane w implementacji narzędzia mierzącego wykonanie programów. Powstało również wiele nazrzędzi mających zapewnić izolacje, które korzystają z tych technologii.

1.1. Technologie

1.1.1. perf

Nowoczesne procesory wyposażone są w sprzętowe liczniki zdarzeń występujących podczas wykonywania programu[2]. Dostępne liczniki zależą od używanego modelu procesora. Często spotykane są między innymi:

- liczba cykli podczas których procesor nie spał
- liczba wykonanych instrukcji maszynowych
- liczba odwołań do pamięci
- liczba trafień i nietrafień w cache procesora, z podziałem na poziomy

Licznik liczby wykonanych instrukcji maszynowych jest dostępny prawie na każdym obecnie używanym procesorze firmy Intel i AMD. Można z niego również korzystać w większości maszyn wirtualnych (Xen). Właśnie ten licznik postanowiliśmy wykorzystać do mierzenia wydajności w naszym narzędziu.

Perf to mechanizm jądra Linuxa umożliwiający odczytywanie tych liczników przez programy w przestrzeni użytkownika. Pozwala on pobrać liczbę zdarzeń które zaszły w kontekście wybranego procesu, lub grupy procesów (zdefiniowanej za pomocą mechanizmu `cgroup`). Można odróżnić zdarzenia które wystąpiły gdy proces był w przestrzeni użytkownika, od tych które wystąpiły gdy był w przestrzeni jądra.

1.1.2. seccomp-bpf

Seccomp[3] to dostępny w Linuxie mechanizm filtrowania wywołań systemowych. Pozwala on przygotować i przekazać do jądra program zapisany w kodzie BPF [1], który decyduje o dozwolonych dla danego procesu wywołaniach systemowych. Filtr ten jest sprawdzany

[1] BPF - Berkeley Packet Filter, jest rodzajem kodu bajtowego, pierwotnie służącego do budowania filtrów pakietów sieciowych. Przykładowo, do niego kompilują się filtry programu Wireshark.

bardzo wcześnie podczas obsługi wywołań systemowych, jeszcze zanim sterowanie trafi do kodu odpowiedzialnego za konkretne wywołanie systemowe. Pozwala to ograniczyć interakcje wykonywanego programu z jądrem i w konsekwencji zmniejszyć powierzchnię ataku na jądro. Dodatkową zaletą seccompa jest jego szybkość, filtry wykonują się w całości w przestrzeni jądra, więc unikamy narzutu spowodowanego przełączeniem kontekstu.

1.1.3. ptrace

Jądro linuxa umożliwia śledzenie i ingerowanie w wykonanie programu. Proces śledzący ma możliwość zatrzymywania procesu śledzonego przy wystąpieniu różnych zdarzeń (m.in. rozpoczęcie wywołania systemowego, powrót z wywołania systemowego, otrzymanie sygnału, zwrócenie odpowiedniej wartości przez filtr seccomp), pobierania danych z rejestrów i przestrzeni adresowej, oraz ich modyfikację. Ptrace jest używany przez debuggery (gdb) i nie służy do izolacji procesu.

1.1.4. namespaces

Namespaces to grupa mechanizmów izolacji w jądrze Linuxa pozwalających na wydzielenie niektórych zasobów jądra widocznych z punktu widzenia wybranej grupy procesów, które tracą dostęp do gloalnych zasobów. Składa się ona z następujących mechanizmów:

mount namespaces pozwala zmienić dostępne dla procesu drzewo plików. Proces widzi inny katalog główny, oraz inne punkty montowania. Jest to rozwiązanie podobne do wywołania `chroot`, ale od niego bardziej elastyczne i stworzone z myślą o izolacji.

PID namespaces pozwala odizolować widoczne drzewo procesów oraz ich numery. Procesy wewnątrz namespace'u nie widzą procesów z zewnątrz, nie mogą na nie oddziaływać i otrzymują identyfikatory z niezależnej puli od 1 wzwyż, jak gdyby były jedynymi procesami w systemie. Są one jednak widoczne dla procesów z zewnątrz, które mogą pobierać o nich informacje.

IPC namespaces pozwala oddzielić obiekty komunikacji międzyprocesowej System V oraz kolejki komunikatów POSIX widoczne dla wybranych procesów

user namespaces pozwala przedstawić wybranej grupie procesów inne identyfikatory użytkowników i grup. W szczególności użytkownik, który utworzył namespace, może otrzymać wewnątrz niego UID równy zero (być w nim użytkownikiem root). Otrzymuje on również pełne uprawnienia (`capabilities`) do zarządzania obiektami utworzonymi w ramach tego namespace'u. Na przykład, nieuprzywilejowany użytkownik może utworzyć user namespace, a później utworzyć mount namespace. W nim będzie mu wolno montować systemy plików w dowolnym miejscu (jakby miał uprawnienia roota). Nie będzie to jednak miało wpływu na procesy na zewnątrz user namespace'u, oraz znajdujące się w innych mount namespace'ach. User namespace'y tworzą hierarchię drzewa, ukorzenioną w tzw. `init user namespace` utworzonym przez jądro systemu. Użytkownik root w tym namespac'ie jest "prawdziwym rootem", tzn. jego uprawnienia dotyczą wszystkich obiektów jądra, w tym wszystkich podrzędnych namespace'ów, oraz dostępu do prawdziwych urządzeń sprzętowych.

Wszystkie namespace'y oprócz user namespace'ów mogą być tworzone tylko przez procesy uprzywilejowane (wymagane jest `CAP_SYS_ADMIN`). Procesy mogą znaleźć się w namespac'ie jedynie dziedzicząc go po rodzicu, lub samemu do niego wchodząc. Wejście do już istniejącego

namespace'u (a więc być może ucieczka z namespace'u obecnego do nadrzędnego) wymaga posiadania otwartego deskryptora pliku odnoszącego się do namespace'u docelowego. Dzięki temu proces tworzący namespace nie może złośliwie umieścić w nim innych już istniejących procesów, a proces utworzony wewnątrz (bez deskryptora pliku odnoszącego się do namespace'u na zewnątrz) nie jest w stanie uciec na zewnątrz.

1.1.5. rlimit

Standard POSIX określa mechanizm ograniczania maksymalnego zużycia zasobów przez każdy proces. W Linuxie z jego pomocą można ustawiać górne ograniczenia między innymi na rozmiar przestrzeni adresowej, stosu, czy czas procesora zużyty podczas działania programu.

1.1.6. cgroup

Mechanizm cgroup w jądrze Linuxa pozwala grupować procesy i ograniczać rozmiar zużywanych przez nie zasobów (takich jak czas procesora, pamięć operacyjna, czy liczba operacji wejścia/wyjścia na sekundę). Istnieje możliwość pobierania wartości liczników procesora udostępnianych przez perf dla całych cgroup. W przeciwieństwie do rlimit, procesy są łączone w grupy, a limity dotyczą sumarycznego zużycia zasobów przez wszystkie procesy w grupie. Podobnie jak w przypadku namespace'ów, przynależność do cgroup'y jest automatycznie dziedziczona przez procesy potomne, a same cgroup'y tworzą drzewo.

1.2. Narzędzia

1.2.1. nsjail

1.2.2. docker

1.2.3. lxc

Rozdział 2

Implementacja

Do napisania narzędzia wybrany został język `C++`, ze względu na łatwość wykorzystania niskopoziomowego API jądra Linuxa, oraz możliwość wygodnego budowania abstrakcji. Wykorzystaliśmy bibloteki libseccomp[1] (do budowy filtrów wywołan systemowych), libcap[2] (do obsługi uprawnień), oraz libtclap[3] (do parsowania argumentów wiersza poleceń).

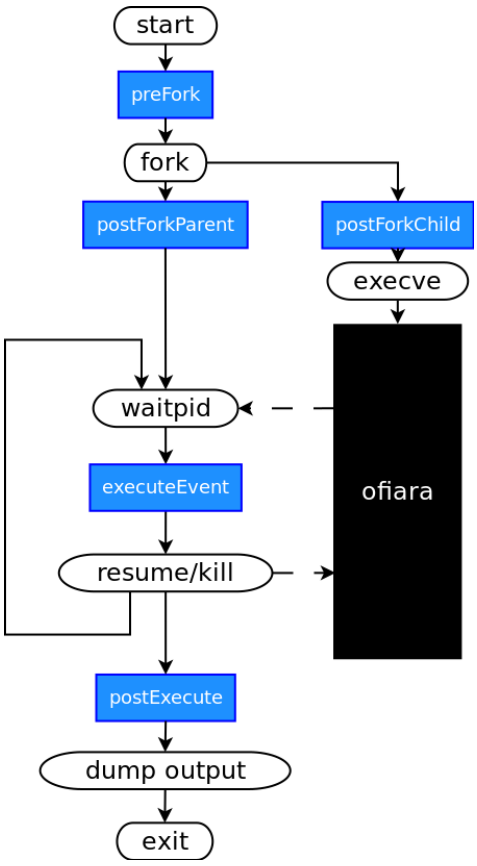
2.1. Architektura

Działanie narzędzia dzieli się na kilka faz. W pierwszej z nich następuje przygotowanie środowiska uruchomieniowego. Następnie tworzony jest proces potomny, który finalizuje przygotowanie środowiska, i przekazuje sterowanie do programu nadzorowanego (wywołanie `execve`). Proces rodzica również finalizuje przygotowanie środowiska, a następnie wchodzi w pętle obsługi zdarzeń z procesu nadzorowanego. Po zakończeniu się procesu nadzorowanego, proces rodzica zbiera wyniki, wypisuje je i kończy swoje działanie.

Do obsługi każdej użytej technologii wydzieliliśmy moduł (ptrace, perf, user namespace, pid namespace, mount namespace, seccomp, memory limit, time limit), powstał też moduł do wykonania całego cyklu i do zrzekania się uprawnień. Podczas całego cyklu moduły zbierają statystyki dotyczące wykonania (ilość instrukcji, zużycie pamięci, błędy wykonania), które następnie są wypisywane w jednym z dostępnych foramtów.

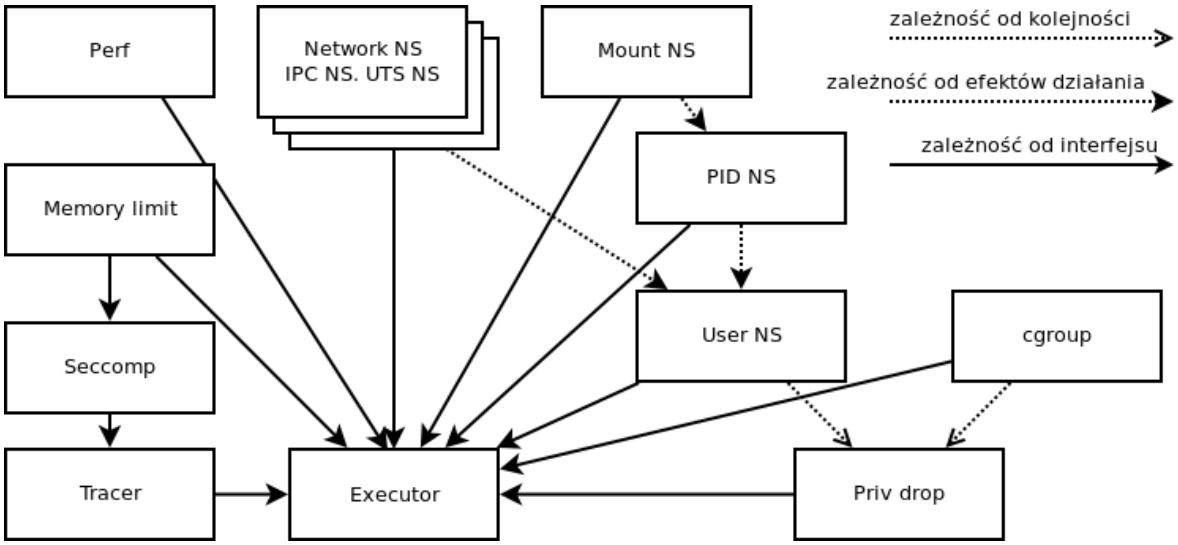
Ponieważ wykorzystane przez nas technologie wymagają wykonania różnych operacji w różnych etapach tego cyklu, wyróżniliśmy w nim kilka etapów i dla każdego stworzyliśmy listę funkcji, które mają się w nim wykonać. Każdy moduł programu definiuje które momenty go interesują i jakie funkcje chce w nich wykonać. Zdefiniowaliśmy:

- preFork – przygotowanie środowiska



Rysunek 2.1: Diagram sterowania

[1] https://github.com/seccomp/libseccomp
[2] http://www.friedhoff.org/posixfilecaps.html
[3] http://tclap.sourceforge.net



Rysunek 2.2: Diagram zależności

- postForkChild – finalizacja środowiska w procesie potomnym
- postForkParent – finalizacja środowiska w procesie rodzica
- executeEvent – reagowanie na zdarzenia związane z procesem potomnym, w trakcie wykonywania się programu nadzorowanego
- postExecute – po zakończeniu się procesu nadzorowanego

Moduły odpowiadające za ptrace i seccomp definiują własne zdarzenia wykonania programu, na które inne moduły mogą reagować:

- ptrace – opakowuje zdarzenie wykonania, rozszerzając je o dodatkowe informacje o procesie, umożliwia też ingerowanie w wykonywany proces, w tym, zakończanie jego działania
- seccomp – opakowuje te zdarzenia ptrace, które polegały na wykonaniu się wywołania systemowego, dodatkowo filtrując zdarzenia i dodając informacje o wywołaniu systemowym

Pomiędzy modułami występują nietrywialne zależności. Kolejność w jakiej moduły wykonują swoje funkcje w danym momencie cyklu jest istotna.

	preFork	postForkChild	postForkParent	executeEvent	postExecute
memory limit		setlimit		read	
output limit		setlimit		detect	
time limit			start timers		verify limits
trace		traceme	ptrace init	trace event	
perf	create barrier	sync	perf open; sync		read stats
user NS	enter NS				
pid NS	enter NS				
UTS NS		enter NS			
IPC NS		enter NS			
network NS		enter NS			
mount NS		enter NS			cleanup
privilege drop		drop	drop		
seccomp	prepare filter	load filter		syscall event	

Rysunek 2.3: Etapy wykorzystywane przez moduły

2.2. Wykorzystanie technologii

Do zliczania instrukcji wykonanych przez nadzorowany program użyliśmy liczników sprzętowych dostępnych przez api perf. Do pobierania zużycia pamięci użyliśmy standardowych metod udostępnianych przez kernel Linuxa. Ograniczanie zużycia tych zasobów (czasu i pamięci) zrealizowaliśmy za pomocą kombinacji mechanizmu `rlimit`, monitorowania aktywności procesu (informowania o wywołaniach systemowych mających wpływ na zużycie zasobów przy pomocy `seccomp` i `ptrace`), oraz okresowego sprawdzania stanu zużycia zasobów.

Izolacje procesu oparliśmy na technologii `namespace`, oddzielając jego system plików (zamontowany w trybie tylko do odczytu), obiekty komunikacji międzyprocesowej, oraz urządzenia sieciowe. Są one dobrym mechanizmem bezpieczeństwa, zostały stworzone z myślą o izolacji procesów, jednak zapewniają one bezpieczeństwo o dużej granularności, izolowane procesy mają pełne możliwości działania w ramach swojego namespace'u. Nieuczciwy zawodnik mógłby próbować użyć możliwości systemu operacyjnego w celu obejścia nałożonych ograniczeń (np, tworzyć łącza nazwane i obchodzić limity pamięci). Do zabezpieczenia się przed takim działaniem użyliśmy technologi `seccomp`, ograniczając dostępne wywołania systemowe.

[Tutaj można wstawić akapit dotyczący boxów.]

2.3. Moduły

2.3.1. limit pamięci

Zużycie pamięci przez proces nadzorowany pobieramy z pliku `/proc/$pid/status` z wartości `vmPeak`. Jest to wartość, która odpowiada maksymalnemu rozmiarowi przestrzeni adresowej od początku działania programu. W celu ograniczenia zużycia pamięci przez program ustawiamy przy pomocy wywołania systemowego `rlimit` maksymalny rozmiar przestrzeni adresowej. Gdy proces przekroczy ten limit przez stos, otrzyma sygnał `SIGSEGV` i zostanie zakończony, natomiast gdy zrobi to z pomocą wywołań `brk` albo `mmap` (z którego korzystają do alokacji dużych obszarów pamięci np, funkcja `malloc`) otrzyma błąd `ENOMEM` (funkcja `malloc` zwraca wtedy wartość `NULL`). Program użytkownika mógłby więc 'testować' limit pamięci,

ukrywając ten fakt przed naszym narzędziem. Żeby temu zapobiec zdecydowaliśmy się:

- ustawiać twardy limit pamięci o pewną ilość x MB większy niż ten podany jako argument, w ten sposób jeżeli program przekroczył limit z pomocą małych alokacji (mniejszych niż x MB), zużycie pamięci większe niż limit zostanie zapisane w `vmPeak`. Wtedy po zakończeniu programu stwierdzimy niepoprawne wykonanie.
- z pomocą filtrów seccomp generować zdarzenia ptrace dla alokacji pamięci (do których użyto funkcji systemowych mmap i mmap2) większych niż x MB. Takich alokacji program nie wykonuje dużo, więc narzut na ich obsługę nie jest odczuwalny. Narzędzie może sprawdzić aktualne zużycie pamięci i w razie próby przekroczenia limitu zakończyć działanie procesu potomnego.

2.3.2. limit rozmiaru wyjścia

Program nadzorowany musi mieć możliwość korzystania z standardowego wejścia i standardowego wyjścia (na które wypisywane jest rozwiązanie testu). W celu uniemożliwienia wygenerowania zbyt dużego wyjścia, co mogłoby spowodować awarię systemu sprawdzającego, konieczne jest ograniczenie jego rozmiarów.

Zrealizowaliśmy to przy pomocy rlimitu na rozmiar pliku (`RLIMIT_FSIZE`). Ogranicza on sumaryczny przyrost wszystkich używanych przez program plików, który może spowodować program. Po przekroczeniu limitu kernel wysyła procesowi sygnał `SIGXFSZ`, który jest przechwytywany i zamieniany na informacje o przekroczeniu limitu rozmiaru wyjścia.

2.3.3. limit czasu

Program nadzorowany może działać długo mimo wykonywania małej liczby instrukcji. Może stać się tak w wyniku świadomego działania (użycia funkcji sleep), albo błędu (zakleszczenie, niepoprawne użycie systemów synchronizacji w programie jednowątkowym). Długo (nieskończenie długo) sprawdzający się program blokuje możliwość sprawdzania innych programów, w związku z czym oprócz limitu instrukcji konieczne jest stosowanie limitów czasu mierzonego na różne sposoby:

- real - czas rzeczywisty
- user - czas procesora spędzony w przestrzeni użytkownika
- sys - czas procesora spędzony wewnątrz wywołań systemowych
- sumaryczny czas procesora

Czas działania programu ograniczyliśmy przy pomocy timera, tworzonego w fazie inicjalizacji. Przy każdym wywołaniu program sprawdza wszystkie dostępne limity (czasu rzeczywistego na podstawie systemowego zegaru monotonicznego, czasów procesora na podstawie wartości z plików z `procfs`). W razie przekroczenia któregoś z limitów, kończy działanie procesu nadzorowanego i zgłasza informacje o przekroczeniu limtu czasu rzeczywistego.

2.3.4. perf

Na etapie `preFork` moduł odpowiedzialny za perfa tworzy barierę, która służy do synchronizacji między procesem rodzica a procesem potomnym.

Na etapie `postForkParent` moduł inicjalizuje perfa za pomocą wywołania systemowego `perf_event_open` ustawiając opcje tak, aby liczyć zdarzenia typu `PERF_COUNT_HW_INSTRUCTIONS`,

liczyć je tylko z przestrzeni użytkownika, i rozpocząć zliczanie dopiero po wykonaniu wywołania `execve`. Ponadto użyliśmy opcji `sample_period` i `wakeup_events` (które umożliwiają otrzymanie powiadomienia, gdy wartości liczników przekroczą określone wartości) do zrealizowania limitu instrukcji.

Następnie następuje synchronizacja na wcześniej utworzonej barierze między kodem modułu wykonującym się w procesie potomnym (etap `postForkChild`) a kodem w procesie rodzica (etap `postForkParent`). Synchronizacja jest potrzebna, aby proces potomny nie uruchomił programu nadzorowanego (wywołaniem systemowym `execve`) przed incializacją perfa w procesie nadzorującym.

W razie otrzymania sygnału `SIGIO`, moduł odczytuje z perfa liczbę wykonanych instrukcji, i jeśli przekracza ona większa limit, kończy działanie nadzorowanego procesu i zgłasza przekroczenie limitu liczby wykonanych instrukcji.

Na koniec, na etapie `postExecute`, moduł odczytuje z perfa liczbę wykonanych instrukcji i umieszcza ją w danych wynikowych.

2.3.5. user namespaces

Z modułów odpowiedzialnych za namespace-y jako pierwszy wykonuje się ten odpowiedzialny za user namespaces. Dzieje się tak, ponieważ w przypadku, gdy SIO2Jail jest uruchamiany jako użytkownik nieuprzywilejowany, utworzenie innych namespace-ów wymaga wcześniejszego utworzenia user namespace'a i uzyskania uprawnień pseudo-roota wewnątrz niego. Dlatego jest on jako pierwszy na liście funkcji do wywołania przed forkiem (`preFork`)

Moduł ten najpierw tworzy nowy user namespace za pomocą wywołania systemowego `unshare(CLONE_NEWUSER)`, a następnie wypełnia pliki `/proc/self/uid_map`, `/proc/self/gid_map` i `/proc/self/setgroups`, wpisując do nich odwzorowanie między identyfikatorami użytkowników i grup wewnątrz namespace'u, a tymi na zewnątrz. Użytkownikowi, jako który został uruchomiony SIO2Jail, przypisuje wewnątrz namespace'u UID zero, tj. roota . Do pliku `setgroups` zapisywany jest napis `deny`, co uniemożliwa zapis do `gid_map` kosztem zablokowania wywołania systemowego `setgroups`.[4] Następnie odwzorowanie numerów grup jest wypełniane w sposób analogiczny do odwzorowania numerów uzytkowników.

2.3.6. PID namespaces

Moduł odpowiedzialny za PID NSy wykonuje się na etapie `preFork`. Jego zadaniem jest ukrycie przed procesem nadzorowanym pozostałych procesów w systemie.

Moduł tworzy PID NS wywołaniem systemowym `unshare(CLONE_NEWPID)`. Niestety, zmiana PID NSu do którego należy istniejący proces nie jest możliwe, i powyższe wywołanie zmienia jedynie PID namespace procesów potomnych, które w przyszłości powstaną z obecnego procesu. Stąd wynika umieszczenie tego wywołania przed forkiem, aby proces potomny utworzony przez fork był pierwszym procesem w nowym PID NSie.

2.3.7. UTS namespaces

Moduł odpowiedzialny za UTS NSy wykonuje się na etapie `postForkChild`, ale na kolejność jego wykonania względem innych modułów nie ma dużo ograniczeń – wystarczy aby był po user NSie. Celem jego działania jest ukrycie nazwy hosta przed programem nadzorowanym

[4] Jądro wymaga tego po to, aby nieuprzywilejowane procesy, tworząc user namespace, nie mogły zrzec się grup ze swojej listy grup suplementarnych. Jest to podyktowane tym, że bycie członkiem grupy bywa czasami używane jako sposób obniżenia uprawnień, a nie ich podniesienia.

i zastąpienie jej stałym napisem, tak aby zachowanie programu nadzorowanego nie zależało od tego, na którym hoście się on wykonuje.

Tworzy on swój NS wywołaniem systemowym `unshare(CLONE_NEWUTS)` i zmiana nazwy hosta oraz nazwy domeny wewnątrz niego.

2.3.8. IPC namespaces

Moduł odpowiedzlany za ICP NSy również wykonują się na etapie `postForkChild`, i w jego przypadku również jedynym wymaganiem co do kolejności jest to, aby były po user NSie. Jego zadaniem jest uniemożliwić procesowi nadzorowanemu dostęp do utworzonych przez procesy zewnętrzne obiektów komunikacji międzyprocesowej (takich jak pamięć współdzielona System V, czy kolejki komunikatów) nawet w przypadku, gdy wywołania systemowe związane z nimi nie są blokowane.

Moduł tworzy nowy NS wywołaniem systemowym `unshare(CLONE_NEWIPC)`. NS tego typu nie dziedziczy nic z NSu-rodzica, więc nie trzeba nic więcej robić, bo świeżo utworzony IPC NS jest pusty.

2.3.9. network namespaces

Moduł odpowiedzialny za network NSy wykonuje się na etapie `postForkChild`, i podobnie jak w przypadku dwóch poprzednich modułów, wystarczy że wykona się po utworzeniu user NSu. Jego celem jest uniemożliwienie komunikacji sieciowej procesu nadzorowanego ze światem zewnętrznym, w wypadku gdyby wywołania systemowe związane z siecią nie były blokowane.

Tworzy on nowy network NS wywołaniem `unshare(CLONE_NEWNET)`. Ten typ NSu nie dzieczyczi nic z NSu-rodzica. Jedynym interfejsem sieciowym w nowo powstałym network NSie jest nowa instancja interfejsu loopback (inna niż w NSie-rodzicu), przez którą można się komunikować wyłącznie z procesami w tym samym NSie. Usunięcie tego interfejsu wydaje się być niemożliwe, poza tym pozostawienie go pozwala na łatwe przystosowanie narzędzia do nadzorowania programów, które wykorzystują sieć wewnętrznie, np. nasłuchują na jakimś porcie oczekując na podłączenie debugera (taki mechanizm ma np. Java).

2.3.10. mount namespaces

Moduł odpowiedzialny za mount NSy wykonuje się na etapie `postForkChild`, i ważne jest, żeby wykonał się później niż moduły odpowiedzialne za PID NSy i user NSy. Jego zadaniem jest umieszczenie procesu nadzorowanego w odizolowanym systemie plików, w którym znajdą się wszystkie pliki potrzebne do jego działania, ale nie będzie żadnych innych plików, w szczególności ważnych plików systemowych. Opcjonalnie może też zamontować w nim procfs odnoszący się do obecnego w momencie montowania PID NSu. Ponieważ moduł odpowiedzialny za PID NS wykonuje się wcześniej, NSem do którego odnosi się zamontowany procfs będzie ten utworzony przez ów moduł. Montowanie procfsu nie jest to potrzebne do działania nadzorowanego programu, ale jest przydatne przy debugowaniu samego narzędzia.

Moduł ten wykonuje następujące kroki:

- Wywołanie systemowe `unshare(CLONE_NEWNS)`, które tworzy nowy mount NS.
- Rekurencyjna zmiana typu wszystkich punktów montowania na prywatny wywołaniem `mount` z flagami `MS_PRIVATE|MS_REC`. Wyłącza to propagację zmian punktów montowania z wewnątrz NSu na zewnątrz. Od tego momentu montowanie i odmontowywanie

systemów plików wewnątrz NSu nie wymaga posiadania uprawnień na zewnątrz, ponieważ działania te nie wpływają na procesy poza obecnym namespacem.

- Wykonanie bind-mountów w celu umieszczenia odpowiednich katalogów (podanych jako parametry) w nowym katalogu głównym. Zależnie od parametrów, każdy z tych katalogów może być zamontowany w trybie read-only albo read-write.
- Zmiana katalogu głównego na nowy katalog główny wywołaniami:
  `chdir(new_root)`, `pivot_root(".", ".")`, `chroot(".")`
- Jeśli włączone jest montowanie procfsu wewnątrz NSu – utworzenie katalogu `/proc` i zamontowanie w nim procfs.
- Wywołanie `umount("/", MNT_DETACH)`, co odmontowuje poprzedni katalogu główny (który był do tego momentu zamontowany w tym samym miejscu co nowy katalog główny).
- Przemontowanie nowego katalogu głównego na read-only.

2.3.11. privilege drop

Moduł ten wykonuje się po forku oraz wszystkich modułach odpowiedzialnych za NSy. Wykorzystuje do tego etapy `postForkChild` oraz `postForkParent`. W obu z nich wykonuje te same czynności, przy czym raz dotyczą one procesu potomnego, a raz rodzica.

Celem tego modułu jest zrzeknięcie się uprawnień uzyskanych podczas tworzenia user NSa, pozwalających na manipulowanie zasobami wewnątrz niego oraz innych NSów w nim utworzonych, oraz możliwie maksymalne ograniczenie uprawnień, które proces może w przyszłości uzyskać.

Najpierw moduł ogranicza możliwość uzyskania uprawnień w przyszłości:

- Usuwa wszystkie bity z capability bounding set.
- Włącza bity `SECBIT_NOROOT` i (na nowych kernelach) `SECBIT_NO_CAP_AMBIENT_RAISE` oraz ich wersje `*_LOCKED` w securebits (`prctl` z argumentem `PR_SET_SECUREBITS`)
- Włącza bit no-new-privileges wywołując `prctl` z argumentem `PR_SET_NO_NEW_PRIVS`. Jest on potrzebny do nieuprzywilejowanego korzystania z filtrów seccompa.

Na koniec usuwane są wszystkie przywileje (`capabilities`) za pomocą wywołania systemowego `capset`.

Należy zwrócić uwagę, że normalnie, gdy `uid=0`, wywołanie `execve` ustawia wszystkie capabilities na 1. W takim przypadku powyższe usuwanie przywilejów nie miałoby sensu. Na szczęście da się ten mechanizm wyłączyć – wystarczy że ustawiony jest co najmniej jeden z bitów: no-new-privileges lub `SECBIT_NOROOT`.

2.3.12. trace

Pozostałe moduły często potrzebują wykonać jakieś działania w sytuacji, gdy proces nadzorowany otrzyma jakiś sygnał lub wykona `exit()`, ale zanim się zakończy. Często też potrzebują zareagować na wykonane przez niego wywołanie systemowe złożonymi działaniami, których nie da się wyrazić w postaci filtru seccomp.

Aby to umożliwić, stosujemy `ptrace`.

Na etapie `postForkChild` moduł odpowiedzialny za tą funkcjonalność wykonuje `PTRACE_TRACEME`, co powoduje, że proces rodzica staje się jego procesem śledzącym. Następnie wysyła do samego siebie `SIGTRAP`, co powoduje jego zatrzymanie do momentu, gdy proces śledzący obsłuży to zdarzenie.

Na etapie `postForkParent`, moduł ten czeka na za pomocą `waitpid` na moment, w którym proces potomny otrzyma sygnał `SIGTRAP`. Otrzymawszy tą informację wie, że jest już procesem śledzącym. Ustawia wtedy opcje ptrace-a na `PTRACE_O_EXITKILL | PTRACE_O_TRACESECCOMP`.

Pierwsza z nich powoduje, że w wypadku śmierci procesu nadzorującego, proces nadzorowany zostanie automatycznie zabity. Jest to konieczne, ponieważ część ograniczeń, np. ograniczenie rzeczywistego czasu działania, jest zaimplementowana w sposób, który wymaga ciągłego działania procesu nadzorującego. Gdyby przez przypadek lub w wyniku ataku proces nadzorujący zakończył się, a proces nadzorowany kontynuował swoje działanie, mógłby on ominąć te ograniczenia.

Druga z flag powoduje, że proces nadzorowany jest powiadamiany o wywołaniach systemowych, w których filtr seccomp zwrócił `SECCOMP_RET_TRACE`.

Na etapie `executeEvent`, moduł dodaje do informacji o zdarzeniu obiekt `Tracee`, który jest wrapperem na proces nadzorowany i pozwala wykonywać takie operacje jak odczyt numeru i argumentów wywołania systemowego, które próbuje wykonać proces nadzorowany, czy odczyt fragmentów pamięci tegoż procesu. Operacje te są zaimplementowane za pomocą wywołań ptrace. Następnie moduł przekazuje zmodyfikowane zdarzenie, odtąd zwane `traceEvent`, wszystkim modułom na nie oczekującym. Mogą one podjąć decyzję, czy proces ma być wznowiony czy zabity. Na koniec, jeśli podjęta była decyzja o zabiciu procesu, lub jeśli w procesie nadzorowanym wystąpił błąd (co spowodowało wygenerowanie sygnału), proces nadzorowany jest zabijany sygnałem `SIGKILL`, przed którym nie może się bronić. W przeciwnym wypadku jest wznawiany.

2.3.13. seccomp

...

Stworzony filtr syscalli jest konfigurowalny, oraz celowo dosyć zgrubny. Ponieważ nie oparliśmy na nim bezpieczeństwa sio2jail uznaliśmy, że najważniejsza jest jego elastyczność, tak żeby łatwo było go dostosować do zmienionych zasad konkursu, oraz aby przyszłości nie utrudniał on implementacji obsługiinnych języków programowania (które mogą wymagać od środowiska większej ilości funkcji niż statyczne binarki napisane w języku `c++`).

Rozdział 3

Zarządzanie projektem

Projekt był realizowany na przestrzeni 7 miesięcy, od września 2017 do kwietnia 2018. W skład zespołu realizującego projekt wchodziły cztery osoby: Wojciech Dubiel, Tadeusz Dudkiewicz, Przemysław Kozłowski i Maciej Wachulec.

3.1. Zbieranie wymagań projektowych

W związku z udziałem, części osób wchodzących w skład zespołu projektowego, w organizacji Olimpiady Informatycznej, materia, której dotykał projekt, była dla zespołu dość bliska. Dzięki temu możliwym było szybkie i precyzyjne określenie wymagań projektu.

Zamawiającym był Szymon Acedański - członek Komitetu Głównego Olimpiady Informatycznej. W ramach zbierania wymagań projektowych odbyły się dwa spotkania z zamawiającym. Pierwsze, na początku listopada, w celu uzgodnienia wizji projektu i doprecyzowania opisu funkcjonalności. Drugie, w lutym, w celu przedstawienia funkcjonalnego prototypu rozwiązania i zebrania uwag zwrotnych. Zadanie było dobrze zdefiniowane, wobec czego więcej spotkań nie było potrzebnych.

3.2. Zarządzanie projektem

Prace nad projektem odbywały się w jednotygodniowych i dwutygodniowych iteracjach.

W celu organizacji prac wykorzystywaliśmy platformę Youtrack. W jej ramach utworzona została tablica zadań, którym przypisywane były priorytety. Na początku tygodnia były zbierane nowo powstałe (lub wynikające z nadrzędnego planu prac) zagadnienia i umieszczane w tabeli. Następnie każde zagadnienie było odkładane na później lub przypisywane do konkretnej osoby. W trakcie tygodnia zagadnienia wędrowały po kolejnych kolumnach tabeli: otwarte, w realizacji, w recenzji i zakończone.

3.2.1. Kontrola jakości

W ramach organizacji pracy wykorzystaliśmy system kontroli wersji git z centralnym repozytorium umieszczonym na platformie gitlab.com. Ponadto, na otrzymanym od Olimpiady serwerze, zainstalowane zostało oprogramowanie Jenkins. Za każdym razem gdy umieszczony był w repozytorium kod, automatycznie był on budowany oraz uruchamiane były podstawowe testy automatyczne. Narzędzie było uruchamiane na wyselekcjonowanych wcześniej programach, zaś wyniki były porównywane z oczekiwanymi.

3.3. Podział pracy

Projekt był realizowany na przestrzenie 7 miesięcy pomiędzy Wrześniem 2017 a Kwietniem 2018. Ciężko wyłonić po jednej osobie na każdy z modułów zawartych w projekcie, która w całości by zań odpowiadała. Niemniej jednak nadzór nad następującymi obszarami projektu sprawowali odpowiedni:

- Efektywne mierzenie czasu i jego limitowanie - Tadeusz Dudkiewicz
- Mierzenie pamięci i bezpieczeństwo systemu - Wojciech Dubiel
- Przydział prac, devops, kontrola jakości - Maciej Wachulec
- Wdrożenie oprogramowania na platformę SIO2 - Przemysław Kozłowski

Rozdział 4

Testowanie

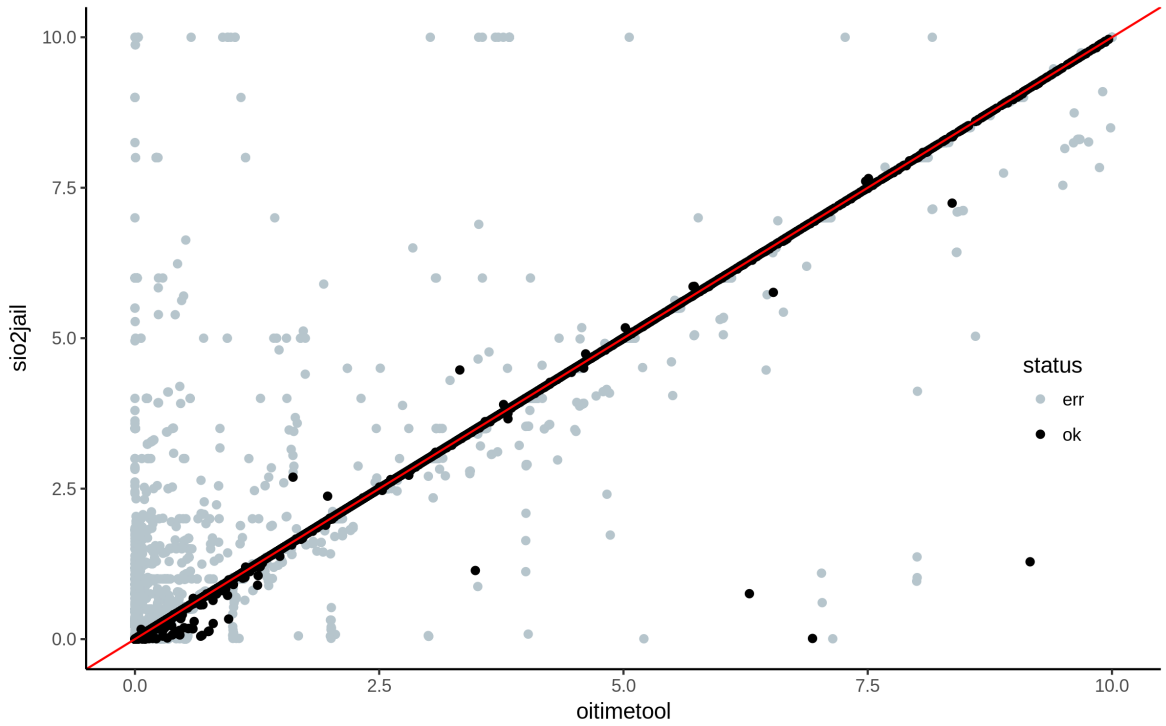
4.1. Środowisko testowe

Chcieliśmy sprawdzić działanie SIO2Jail z programami jak najbardziej zbliżonymi do zgłoszeń konkursowych, oraz porównać jego działanie z działaniem OITimeTool. Wykorzystaliśmy do tego zgłoszenia z II i III etapów XXIII i XXIV Olimpiady Informatycznej. Losowaliśmy (bez powtórzeń) zgłoszenie, oraz test na którym mogło ono być uruchomione (w ramach jednego zadania zgłoszenie jest oceniane na wielu, być może kilkudziesięciu, testach). Zgłoszenia kompilowaliśmy za pomocą gcc-4.9.2, a uruchamialiśmy na systemie linux w wersji 3.16. Każdą parę zgłoszenie - test uruchamialiśmy kolejno, z SIO2Jail, oraz z OITimeTool zapisując statusy sprawdzania, statusy wykonanych programów, zmierzony czas działania (liczbę instrukcji), zmierzoną pamięć, oraz czas trwania procesu sprawdzania. Testy wykonaliśmy dwukrotnie, dla pierwszej wersji narzędzie (te wyniki były zbliżone do zaprezentowanych) i po wprowadzeniu pewnych zmian w narzędziu (drobne poprawki i zmiany w kodzie, z których część powstała po analizie wyników pierwszych testów) wykonaliśmy drugą iteracje testów. W drugiej iteracji wykonaliśmy około 500000 takich uruchomień, których analizę prezentujemy poniżej.

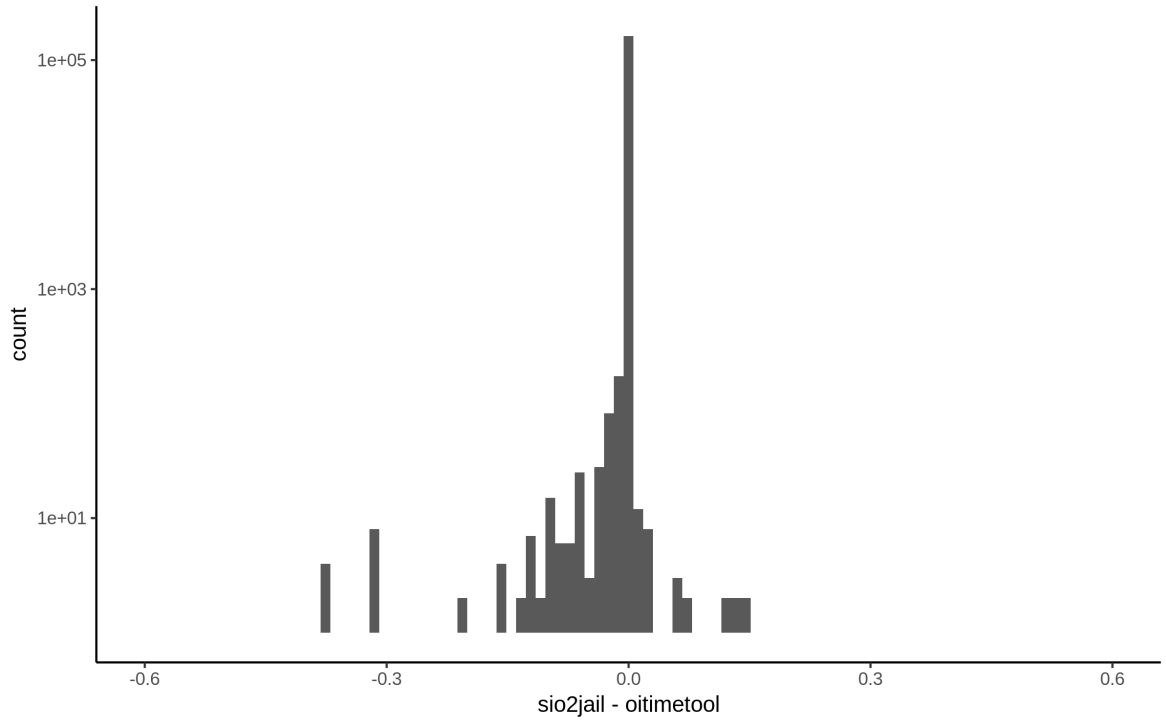
4.2. Wyniki

Ponieważ narzędzie SIO2Jail ma zastąpić OITimeTool najważniejsze było dla nas porównanie czasów mierzonych przez oba narzędzia. Na wykresie 4.2 dla każdego uruchomienia zaznaczyliśmy czasy zmierzone przez oba narzędzia. Wyszarzyliśmy punkty odpowiadające odrzuconym pomiarom - zgłoszeniom niepoprawnym, oraz takich dla których porównywanie czasu nie ma sensu. Czerwona kreska odpowiada zgłoszeniom, dla których zmierzone czasy były identyczne. Niewiele poprawnych zgłoszeń zostało przez SIO2Jail uznanych za działające dłużej, oraz kilkanaście uruchomień działało krócej. Dokładną analizę różnic, oraz przedstawienie odrzuconych wyników zawarliśmy w następnym rozdziale. Należy zwrócić uwagę na to, że są to pojedyncze przypadki, w znakomitej większości oba narzędzia zmierzyły niemal identyczną liczbę instrukcji. Lepiej podobieństwo obu narzędzi oddaje wykres 4.2 przedstawiający histogram różnic w zmierzonej liczbie instrukcji. Uwzględnione zostały w nim tylko nieodrzucone pomiary.

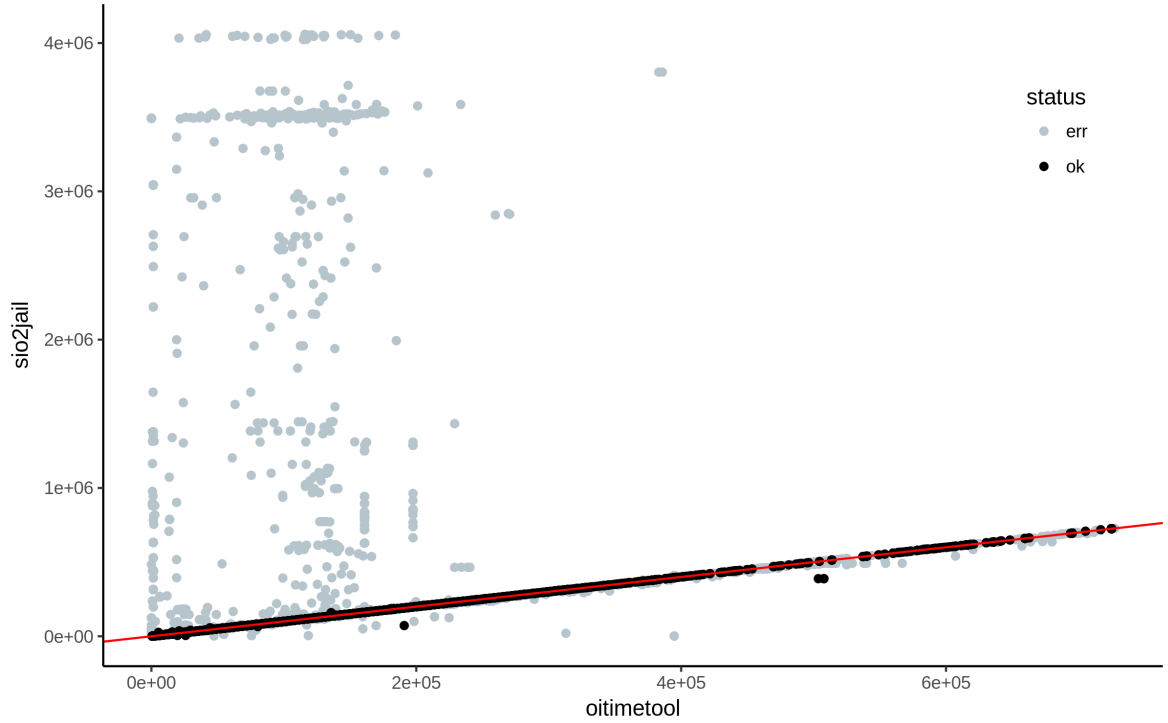
Podobną analizę przeprowadziliśmy dla zużycia pamięci zmierzonego przez oba narzędzia. Analogiczne wykresu jak dla zużycia czasu, to wykres pomiarów 4.2, oraz histogram różnic 4.2. Tutaj wartości zwrócone przez oba narzędzia okazały się jeszcze bardziej zgodne, niż przy pomiarach czasu.



Rysunek 4.1: Czas zmierzony SIO2Jailem w zależności od czasu zmierzonego OITImeToolem



Rysunek 4.2: Histogram różnicy zmierzonych czasów ($SIO2Jail - OITimeTool$)



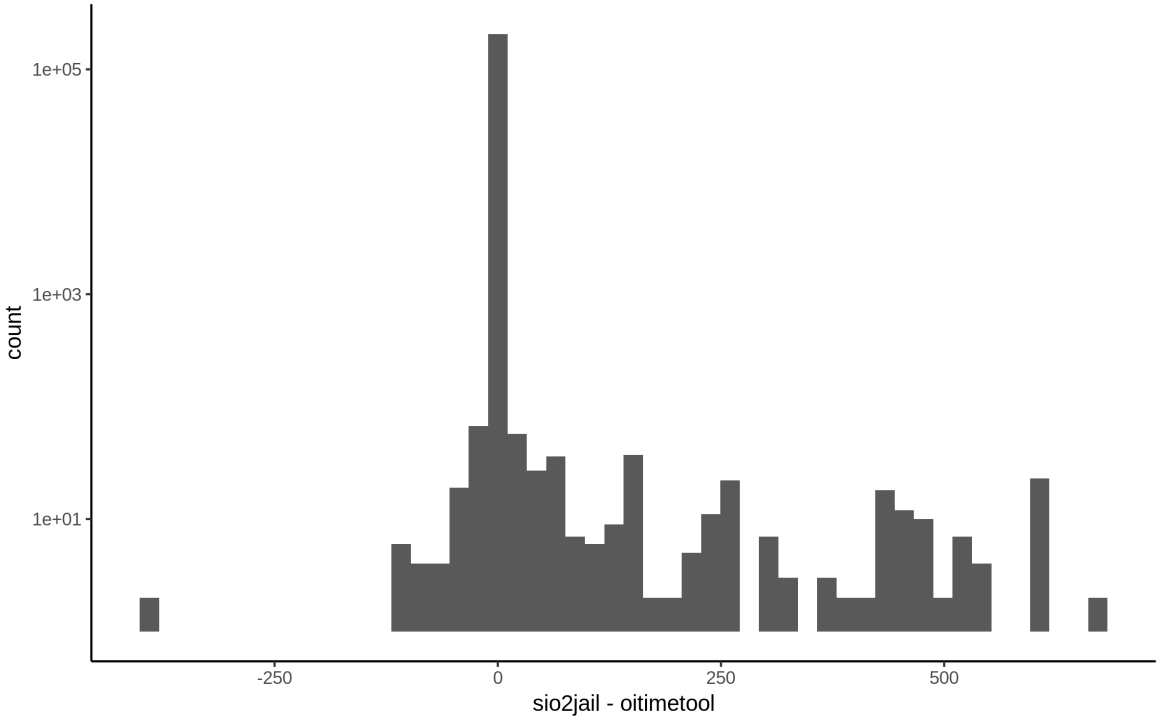
Rysunek 4.3: Zużycie pamięci zmierzone SIO2Jailem w zależności od zmierzonego OITimeToolem

4.3. Analiza różnic

Odrzuciliśmy te uruchomienia, w których choć jedno z narzędzi zwróciło błąd wykonania. Odrzuciliśmy również te programy, które wprost działały niedeterministycznie, czyli korzystały z losowości (funkcja `rand` z ziarnem inicjalizowanym z pomocą `time`, funkcja `std::random_shuffle`), albo polegały na czasie rzeczywistym. Sprawdziliśmy przypadki w których dokładnie jedno z narzędzi stwierdziło błąd wykonania, różnice w ocenie wynikały z:

- Innej polityki dozwolonych wywołań systemowych
- Braku ograniczenia na liczbę wywołanych wywołań systemowych w SIO2Jail
- Błędów w korzystaniu z pamięci, które objawiły się tylko w jednym z narzędzi (programy uruchamiane pod oboma narzędziami miały inny układ pamięci)
- Błędów w korzystaniu z wejścia / wyjścia, które były inaczej obsługiwane przez oba narzędzia
- Różnicy w zmierzonej licznie instrukcji, OITimeTool stwierdził przekroczenie limitu, podczas gdy SIO2Jail tego nie zrobił

Przeanalizowaliśmy również uruchomienia dla których liczby instrukcji zmierzone przez oba narzędzia różniły się istotnie. Różnice wynikały z innej metody zliczania instrukcji. Sprzętowe liczniki procesora zliczają instrukcje wykonujące wiele operacji (z przedrostkiem `REP`) jednokrotnie, podczas gdy OITimeTool liczył każdą operację oddzielnie. Dodatkowo liczniki sprzętowe są zwiększane podczas obsługi przerwań [2, 18-5]. Faktycznie, okazało się,

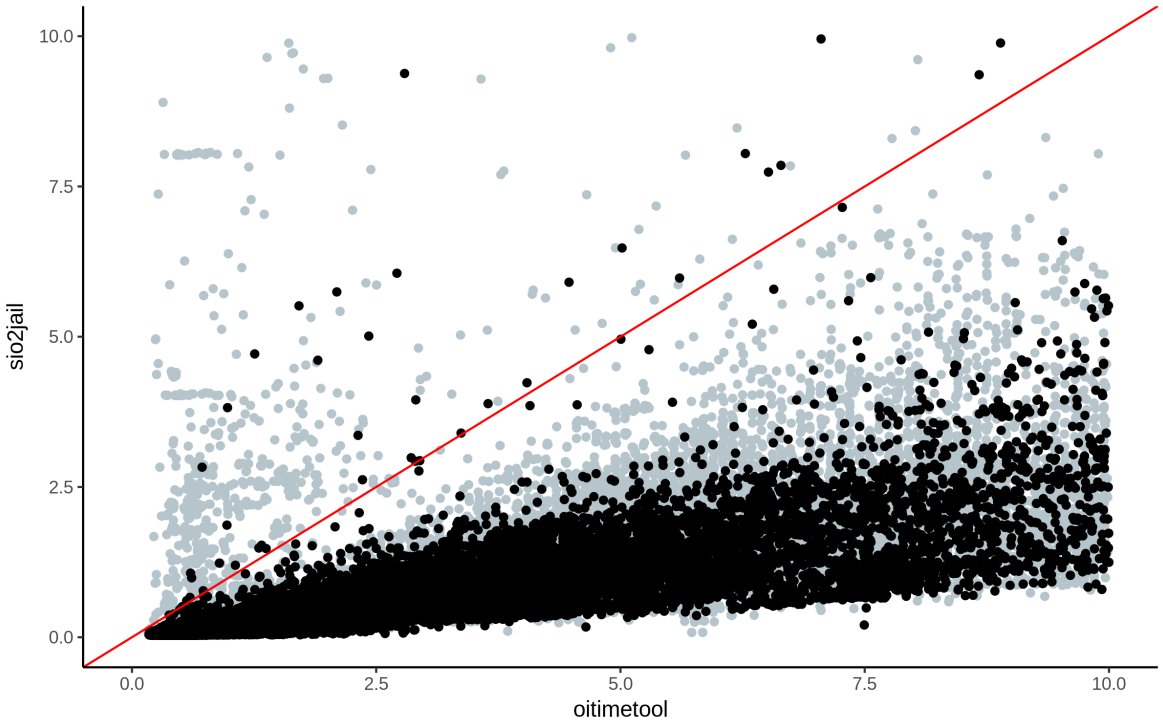


Rysunek 4.4: Histogram różnicy zmierzonych zużyć pamięci ($SIO2Jail - OITimeTool$)

że największe różnice były widoczne przy programach wykorzystujących operacje takie jak `memcpy`, `memset`, albo inicjalizację tablic `int array[n] = 0`. Takie operacje mogą być zaimplementowane z wykorzystaniem instrukcji `REP` i powodować rozbieżności w liczbie zliczonych instrukcji pomiędzy sprzętowymi licznikami, a OITimeToolem. Programy w których takie różnice są istotne okazały się pojawiać rzadko wśród programów zgłaszanych przez uczestników Olimpiady Informatycznej. W związku z tym uznaliśmy, że metoda zliczania instrukcji oparta o sprzętowe liczniki jest dostatecznie dobra do zastosowania w konkursach algorytmicznych.

4.4. Wydajność

Ciekawym wynikiem przeprowadzonych testów było porównanie wydajności obu narzędzi, których czasy działania zaprezentowaliśmy na 4.2. Okazało się, że sprawdzając z SIO2Jail można spodziewać się wydajności około 5-krotnie większej niż przy wykorzystaniu OITimeTool. Wydajność stanowi istotną przewagę SIO2Jail nad OITimeTool, ponieważ uruchamianie zgłoszeń zawodników zużywa najwięcej zasobów spośród zadań systemu Olimpiady Informatycznej.

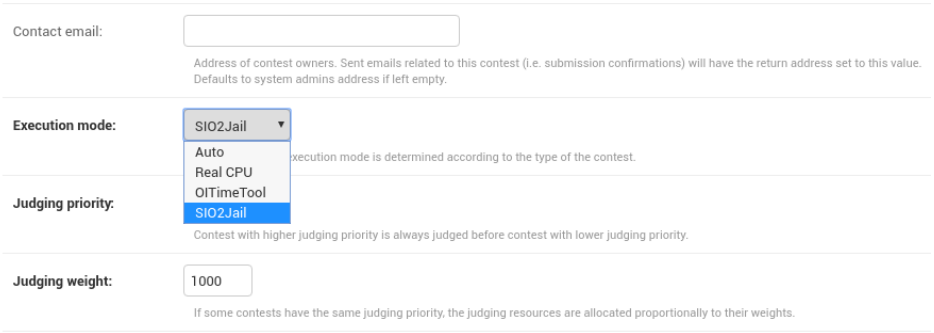


Rysunek 4.5: Czas działania SIO2Jaila w zależności od czasu działania OITimeToola

Rozdział 5

Wdrożenie

Uruchamianie zadań w systemie Olimpiady Informatycznej odbywa się przy pomocy specjalnych serwisów zwanych *sprawdzaczkami*, które są uruchamiane na dedykowanych serwerach. Sprawdzaczki dotychczas obsługiwały kilka rodzajów zadań (między innymi `vcpu-exec` uruchamiający program przy użyciu OITimeTool). W celu zachowania kompatybilności i możliwości przywrócenia sprawdzania przy pomocy OITimeTool, integrację SIO2Jail z systemem sprawdzającym zrealizowaliśmy poprzez nowy rodzaj zadań `sio2jail-exec`. Przełączanie pomiędzy trybami sprawdzania zrealizowaliśmy w ustawieniach, zarówno konkursu, jak i instancji systemu Olimpiady Informatycznej. W globalnych ustawieniach można wybrać domyślny rodzaj sprawdzania oraz umożliwić przełączanie trybów sprawdzania w ustawieniach konkursu.



Rysunek 5.1: Konfiguracja trybu sprawdzania w ustawieniach konkursu

Rozdział 6

Podsumowanie

TODO

Bibliografia

[1] Szymon Acedański, *Wykorzystanie sprzętowych liczników zdarzeń do oceny wydajności algorytmów* (2009).

[2] Intel, *64 and IA-32 Architectures Software Developer's Manual June 2016*

[3] *Linux Kernel Documentation, Seccomp BPF (SECure COMPuting with filters)*