Programowanie niskopoziomowe – Lab 2

Wstęp Teoretyczny

Kompilatory (GCC, Clang) oraz generowanie kodu asemblera:
Kompilatory tłumaczą kod wysokopoziomowy (C/C++) na kod maszynowy. GCC (GNU Compiler Collection) i Clang (kompilator C/C++ oparty na LLVM) są popularnymi kompilatorami na systemie Linux. Mogą one generować wynik w postaci kodu asemblerowego zamiast pliku binarnego, wykorzystując opcję -S. Na przykład, uruchomienie:

```
g++ -O0 -S program.cpp -o program.s
```

Output from C/C++ source in gcc

zatrzyma proces kompilacji po wygenerowaniu kodu asemblera i utworzy plik program.s zawierający kod asemblera. Flaga -O0 dezaktywuje optymalizacje, co sprawia, że wygenerowany asembler jest bardziej czytelny. Wyższe poziomy optymalizacji (-O1, -O2, -O3) stosują różne transformacje poprawiające wydajność, co może znacząco zmienić wygenerowany kod (np. usuwanie nieużywanego kodu, inline'owanie funkcji). Na przykład, przy -O2 kompilator wykonuje niemal wszystkie dostępne optymalizacje, które nie wiążą się z poważnymi kompromisami między rozmiarem a prędkością. Używanie -S na różnych poziomach optymalizacji pozwala zaobserwować, jak ten sam kod C++ może dawać różne wyniki w kodzie maszynowym.

Narzędzia debugowania (GDB, LLDB):
Debuggery pozwalają uruchamiać programy krok po kroku oraz inspekcję ich stanu. GDB (GNU Project Debugger) umożliwia zobaczenie, co dzieje się „wewnątrz" programu podczas jego wykonywania. Dzięki GDB można ustawiać punkty przerwania (aby zatrzymać wykonanie w określonych liniach lub funkcjach), przechodzić krok po kroku przez instrukcje, sprawdzać wartości zmiennych, rejestrów, a nawet deasemblować kod „w locie". LLDB jest podobnym narzędziem z projektu LLVM, umożliwiającym debugowanie C/C++ poprzez inspekcję pamięci, kontrolę wykonania i przerywanie kodu w określonych miejscach. Oba narzędzia zazwyczaj używa się z programami skompilowanymi z symbolami debugowania (flaga -g), co umożliwia powiązanie kodu maszynowego z liniami źródłowymi. Przykładowe użycie GDB:

- gdb ./a.out – uruchomienie debuggera
- break main – ustawienie punktu przerwania w funkcji main
- run – uruchomienie programu
- next/step – przechodzenie przez kolejne linie
- info registers – wyświetlenie zawartości rejestrów
- disassemble /m main – wyświetlenie asemblera funkcji main z wkomponowanym kodem źródłowym

https://www.tutorialspoint.com/gnu_debugger/index.htm

Narzędzia do śledzenia kodu (objdump, radare2):
Czasami jest potrzeba przeanalizowania kodu maszynowego programu poza sesją debugowania. objdump to narzędzie z pakietu GNU binutils, które wyświetla informacje o plikach obiektowych i wykonywalnych. Użycie:

```
objdump -d a.out > disasm.txt
```

deasembluje sekcje wykonywalne programu i zapisze wynik do pliku disasm.txt. W pliku znajduje się sekcja odpowiadającą funkcji main (oznaczoną jako <main>:).
Innym narzędziem jest **radare2** – otwartoźródłowy framework do inżynierii wstecznej. Radare2 umożliwia deasemblację, debugowanie, analizę plików binarnych oraz edycję plików binarnych z poziomu linii poleceń. Na przykład, uruchomienie:

```
r2 -A hello
```

(oznaczające automatyczną analizę) pozwala użyć komend takich jak:

- afl – wyświetlenie listy funkcji
- pdf @ main – wyświetlenie deasemblowanego kodu funkcji main

Disasemblery i analizatory binarne (IDA, Ghidra):
Do bardziej przyjaznej lub zaawansowanej analizy statycznej używa się narzędzi disasemblujących. **IDA Pro (Interactive Disassembler)** to komercyjne narzędzie (dostępna jest też wersja darmowa o ograniczonych możliwościach), które graficznie wyświetla zdisasemblowany kod i ułatwia analizę. IDA generuje kod asemblera z kodu maszynowego i wspiera wiele architektur.
Ghidra to darmowy, otwartoźródłowy pakiet do inżynierii wstecznej opracowany przez NSA. Oferuje interaktywny interfejs graficzny do analizy plików binarnych, z funkcjami podobnymi do IDA, w tym wbudowanym dekompilatorem, który przekształca kod maszynowy w pseudokod przypominający C. Narzędzia te pomagają przekształcić surowy kod maszynowy w zrozumiałą strukturę, co jest szczególnie przydatne przy analizie kodu po optymalizacjach.

Wpływ optymalizacji kompilatora:
Optymalizacje mogą sprawić, że wygenerowany kod asemblera będzie znacznie różnił się od kodu źródłowego. Przy poziomie -O0 (bez optymalizacji) kod asemblera ściśle odpowiada kodowi źródłowemu – każda linia C++ może odpowiadać kilku instrukcjom, a kompilator często używa stosu do przechowywania zmiennych. To ułatwia śledzenie, ale powoduje, że kod jest wolniejszy i większy. Przy wyższych poziomach optymalizacji kompilator stara się utrzymywać wartości w rejestrach, usuwać zbędne obliczenia, inline'ować małe funkcje i usuwać nieużywany kod. Na przykład, nieużywana zmienna lub obliczenie, które nie wpływa na wynik programu, może być całkowicie pominięte przy -O2 lub -O3. Pętle mogą zostać rozwinięte lub przekształcone, a wskaźnik ramki (%rbp) może zostać pominięty (np. dzięki opcji -fomit-frame-pointer aktywnej przy wyższych optymalizacjach). Zmiany te poprawiają wydajność, ale mogą utrudniać debugowanie i korelację z kodem źródłowym. Porównując kod asemblera wygenerowany z -O0 i -O2, można zauważyć, że np. w trybie -O0 funkcja wywołania zawiera instrukcje push/pop rejestrów, a przy -O2 funkcja może być inline'owana lub mnożenie przez 2 realizowane za pomocą pojedynczej instrukcji przesunięcia.

Tłumaczenie konstrukcji C++ na asembler:
Konstrukcje wysokopoziomowe (pętle, instrukcje warunkowe, wskaźniki) mają swoje odpowiedniki w kodzie niskopoziomowym.

- **Pętla:** W C++ (np. for, while lub do-while) jest tłumaczona na zestaw instrukcji asemblerowych z użyciem skoków warunkowych. Kompilator tworzy etykietę na początku pętli oraz instrukcję skoku na jej koniec, wraz z warunkowym skokiem wychodzącym z pętli. Przykład:

```
  movl   $0, i      ; i = 0 (inicjalizacja)
Lloop_start:
  cmpl   $5, i      ; porównanie i z 5 (warunek pętli)
  jge    Lloop_end  ; jeśli i >= 5, zakończ pętlę
  addl   i, sum     ; sum += i  (ciało pętli)
  addl   $1, i      ; i++     (inkrementacja)
  jmp    Lloop_start ; skok z powrotem do początku pętli
Lloop_end:
```

Asembler używa etykiet (Lloop_start, Lloop_end) zamiast konstrukcji języka wysokopoziomowego.

- **Instrukcja warunkowa:** Przykładowo, instrukcja if (x < 0) y = 1; else y = 2; zostaje przetłumaczona na:

```
  cmpl   $0, x
  jge    .Lelse        ; skocz do else, jeśli x >= 0
  movl   $1, y         ; część then: y = 1
  jmp    .Lafter
.Lelse:
  movl   $2, y         ; część else: y = 2
.Lafter:
```

Instrukcja cmpl ustawia flagi, a jge decyduje, która część instrukcji warunkowej zostanie wykonana.

- **Wskaźniki:** Operacje na wskaźnikach, np. *(p+3) = 7;, są tłumaczone na arytmetykę adresów. Kompilator oblicza p + 3*4 (dla 4-bajtowego int) i zapisuje 7 pod tym adresem:

```
movq   p, %rax
movl   $7, 0xc(%rax)   ; 0xc = 12 dziesiętnie, czyli 3*4 bajty
```

Pokazuje to, że arytmetyka wskaźników to prosta arytmetyka na adresach.

ABI (Application Binary Interface) i wywołania funkcji:
ABI definiuje sposób komunikacji funkcji na poziomie maszynowym – jak przekazywane są argumenty, które rejestry są zachowywane, a jak zwracana jest wartość.
W systemie **x86-64 Linux (System V AMD64 ABI)**:

- Pierwsze sześć argumentów (całkowitych lub wskaźników) przekazywane jest w rejestrach: %rdi, %rsi, %rdx, %rcx, %r8, %r9.
- Dalsze argumenty są przekazywane przez stos (od prawej do lewej).
- Wartość zwracana zwykle trafia do rejestru %rax.
  Jeśli typ zwracany jest większy (np. struktura), wywołujący może zarezerwować miejsce i przekazać wskaźnik jako ukryty pierwszy argument. ABI określa także, które rejestry muszą być zachowane przez wywoływaną funkcję (callee-saved: np. %rbp, %rbx, %r12-%r15) oraz które mogą być nadpisane (caller-saved: np. %rax, %rcx, %rdx).
  Przykładowo, wywołanie funkcji add(int a, int b) z add(x, y) spowoduje, że x trafi do %edi, a y do %esi. Wewnątrz funkcji te wartości są pobierane z tych rejestrów. Funkcja zwracająca int umieści wynik w %eax.
  ABI tłumaczy też użycie stosu – wywołujący umieszcza adres powrotu na stosie, a lokalne zmienne mogą być tam przechowywane. Każda funkcja ma prolog (rezerwację stosu) oraz epilog (przywracanie stanu stosu i zwrot).

Zarządzanie pamięcią dynamiczną (malloc/new i free/delete):
Przydzielanie pamięci w C++ (new i delete) oraz w C (malloc i free) jest tłumaczone na wywołania funkcji bibliotecznych, które zarządzają stertą.

- Używając new w C++ wywoływana jest funkcja **operator new**, która domyślnie wywołuje malloc(), aby przydzielić pamięć.
- Analogicznie, delete wywołuje funkcję **operator delete**, która zazwyczaj wywołuje free(). W asemblerze, operacja:

```
int *p = new int(42);
```

przetłumaczona może wyglądać następująco:

```
movl   $4, %edi      ; rozmiar 4 bajtów (int) jako argument
call   _Znwm@PLT     ; wywołanie operator new(size_t) – zwraca wskaźnik w RAX
movl   $42, (%rax)   ; zapis 42 do przydzielonej pamięci
movq   %rax, -8(%rbp) ; zapis wskaźnika p na stosie
```

Następnie delete p; zostanie skompilowane do:

```
movq   -8(%rbp), %rdi  ; załaduj wskaźnik p do RDI
call   _ZdlPv@PLT      ; wywołanie operator delete(void*)
```

W C, wywołanie malloc tłumaczy się na wywołanie funkcji malloc z biblioteki, a free analogicznie wywołuje free.
Ważne jest, że w asemblerze widać jedynie wywołania funkcji zewnętrznych, a szczegóły implementacji są ukryte. Dodatkowo, alokacja pamięci często korzysta z wywołań systemowych (np. brk lub mmap) w celu przydzielenia większej przestrzeni, jednak są one abstrakcją dla programisty.

Układ pamięci i translacja adresów:
Pamięć uruchomionego programu jest podzielona na segmenty:

- **Segment tekstowy** zawiera instrukcje wykonywalne (kod) i jest zazwyczaj oznaczony jako tylko do odczytu.
- **Segment danych** zawiera zmienne globalne i statyczne. Dzieli się na część inicjalizowaną (dla zmiennych z wartością początkową) oraz BSS (dla zmiennych niezainicjalizowanych, domyślnie zerowanych).
- **Sterta (heap)** to obszar dynamiczny, zaczynający się po segmencie danych, rosnący ku wyższym adresom. Funkcje malloc i new przydzielają pamięć z tego obszaru.
- **Stos** jest używany do przechowywania ramek wywołań funkcji oraz zmiennych lokalnych. Zaczyna się zazwyczaj od wysokich adresów i rośnie w dół.
  Dodatkowo, mogą istnieć segmenty dla bibliotek współdzielonych oraz regiony zaalokowane przez mmap.
  Kiedy wypiszesz adresy zmiennych, zauważysz, że są one oddzielone – np. globalne zmienne (adresy typu 0x55555555xxxx), sterta (np. 0x7f08c0000000), a stos (np. 0x7ffde9f2abc0).
  Wszystkie te adresy są **wirtualne** – program operuje w swoim wirtualnym adresowym, a system operacyjny używa MMU i tablic stron, aby przetłumaczyć je na adresy fizyczne. Na przykład, adres 0x7ffde9f2abc0 to adres wirtualny, który system operacyjny tłumaczy na fizyczną lokalizację w pamięci RAM. To tłumaczenie odbywa się automatycznie i pozwala na izolację procesów, tak że dwa procesy mogą mieć takie same wirtualne adresy, ale odnoszą się do różnych miejsc w fizycznej pamięci.

Warto zobaczyć:

https://cs61.seas.harvard.edu/site/2018/Asm2/#:~:text=One%20set%20of%20calling%20convention,rax

Zadania praktyczne

Generowanie kodu asemblera za pomocą kompilatora:

Napisz prosty program w C++ (np.Hello World lub prostą pętle for). Użyj kompilatora, aby wygenerować kod asemblera. Skompiluj go bez optymalizacji oraz z optymalizacją:

```
g++ -O0 -S hello.cpp -o hello_O0.s

g++ -O2 -S hello.cpp -o hello_O2.s
```

Przejrzyj oba pliki (hello_O0.s oraz hello_O2.s).
Zadanie: Zidentyfikuj różnice w kodzie asemblera (np. dodatkowe instrukcje lub różnice w sposobie przechowywania zmiennych/rejestrów).

Następnie przepisz ten program do C. Użyj kompilatora, aby wygenerować kod asemblera. Skompiluj go bez optymalizacji oraz z optymalizacją:

```
gcc -O0 -S hello.c -o helloC_O0.s

gcc -O2 -S hello.c -o helloC_O2.s
```

Przejrzyj oba pliki (helloC_O0.s oraz helloC_O2.s).
Zadanie: Zidentyfikuj różnice w kodzie asemblera (np. dodatkowe instrukcje lub różnice w sposobie przechowywania zmiennych/rejestrów).

Warto zobaczyć: https://www.cs.fsu.edu/~baker/opsys/notes/assembly.html

Używanie objdump do deasemblacji pliku binarnego:
Weź skompilowany program (np. hello.out skompilowany bez opcji -S). Uruchom:

```
objdump -d hello.out > disasm.txt
```

W pliku disasm.txt znajduje się wyeksportowany asembler z pliku hello.out. Otwórz plik i znajdź sekcję odpowiadającą funkcji main (będzie oznaczona jako **<main>**).

Zadanie: Porównaj asembler wygenerowany przez opcję -S z deasemblacją uzyskaną za pomocą objdump. Czy są zgodne? (Powinny być, z niewielkimi różnicami w nazwach symboli lub formatowaniu). Zaznacz, gdzie w deasemblacji pojawiają się fragmenty odpowiadające instrukcjom (np. dodawaniu czy pętli).

Warto zobaczyć:

- https://www.geeksforgeeks.org/objdump-command-in-linux-with-examples/
- https://www.thegeekstuff.com/2012/09/objdump-examples/
- https://www.infosecinstitute.com/resources/secure-coding/how-to-use-the-objdump-tool-with-x86/

Debugowanie za pomocą GDB i LLDB:
Skompiluj program z symbolami debugowania:

```
g++ -O0 -g sum.cpp -o sum_dbg.out

lub

gcc -O0 -g sum.c -o sumC_dbg.out
```

Uruchom program w GDB:

```
gdb ./sum_dbg.out

lub

gdb ./sumC_dbg.out
```

W GDB:

- Ustaw punkt przerwania komendą break main, a następnie wpisz run. Program zatrzyma się na początku funkcji main.
- Używaj step lub next, aby wykonywać kolejne linie kodu. Wpisz disassemble /m main, aby zobaczyć kod asemblera z wkomponowanym kodem źródłowym. *Zadanie:* Przejdź krok po kroku przez program i użyj info registers (lub print nazwa_zmiennej), aby sprawdzić, jak wartości zmiennych odpowiadają zawartości rejestrów lub pamięci. Na przykład, po operacji dodawania sprawdź, w którym rejestrze znajduje się wynik operacji.

GDB Tutorials:

- https://www.tutorialspoint.com/gnu_debugger/index.htm
- https://www.geeksforgeeks.org/gdb-step-by-step-introduction/
- https://chyla.org/artykuly/cpp/gdb-tutorial.html

Spróbuj użyć LLDB w podobny sposób:

```
lldb sum_dbg.out
```

i użyj analogicznych komend (breakpoint set -n main, run, step, itp.).

LLDB Tutorial: https://lldb.llvm.org/use/tutorial.html

Eksploracja z radare2:
Uruchom radare2 na skompilowanym programie (hello):

```
r2 -A hello
```

Gdzie flaga -A wykonuje automatyczną analizę pliku binarnego. W powłoce radare2 wykonaj:

- afl – aby wyświetlić listę wykrytych funkcji (powinieneś zobaczyć main oraz inne funkcje, np. wywołania biblioteczne).
- pdf @main – aby wydrukować deasemblowany kod funkcji main.
  Zadanie: Znajdź w wyjściu radare2 instrukcje odpowiadające fragmentom twojego kodu wysokopoziomowego (np. instrukcję dodawania lub początek pętli). Po skończonej analizie wpisz q, aby wyjść z radare2.

Radare2 materiały dodatkowe:

- https://book.rada.re/
- https://www.geeksforgeeks.org/how-to-use-radare2/
- https://kindawingingit.medium.com/radare2-an-introduction-d6762dceeac5
- https://www.megabeets.net/a-journey-into-radare-2-part-1/
- https://www.megabeets.net/a-journey-into-radare-2-part-2/
- https://www.megabeets.net/reversing-a-self-modifying-binary-with-radare2/

Porównanie analizy w IDA/Ghidra
Jeśli masz dostęp do IDA Free lub Ghidra, załaduj do niego skompilowany program sum.c. Przejdź do funkcji main w interfejsie graficznym.
Zadanie: Korzystając z interfejsu deasemblacji, dopasuj fragmenty do twojego kodu źródłowego. Zauważ, jak narzędzie może oznaczać gałęzie oraz pętle. Jeśli używasz Ghidry, skorzystaj z dekompilatora, aby zobaczyć reprezentację przypominającą C i porównaj ją z oryginalnym kodem. (Nawet jeśli nie możesz uruchomić tych narzędzi, zastanów się: jakie zalety mają narzędzia takie jak IDA i Ghidra w porównaniu z surową deasemblacją przez objdump? Rozważ aspekty interaktywnej nawigacji, identyfikacji wywołań funkcji czy dekompilacji do pseudo-kodu.)

Do porównanie różnych dekompilatorów można wykorzystać: https://dogbolt.org/?id=8f1e28f5-3bfc-4d41-8be2-82c12f54487f

Link do pobrania Ghidra: https://github.com/NationalSecurityAgency/ghidra

Materiały dodatkowe do Ghidry:

- https://medium.com/@zaid960928/ghidra-tutorial-introduction-a3889e138434
- https://medium.com/@acheron2302/ghidra-tutorial-in-reverse-engineering-for-window-absolute-begineer-302ba7d810f
- https://wrongbaud.github.io/posts/ghidra-training/
- https://www.varonis.com/blog/how-to-use-ghidra
- https://byte.how/posts/what-are-you-telling-me-ghidra/

Analiza asemblera pętli:
Napisz krótki program w C++ z pętlą, np.:

```
#include <iostream>
using namespace std;
int main() {
   int sum = 0;
   for (int i = 0; i < 5; ++i) {
      sum += i;
   }
   cout << "Sum = " << sum << endl;
}
```

Skompiluj go poleceniem:

```
g++ -O0 -S loop.cpp -o loop.s
```

Otwórz plik loop.s i znajdź sekcję odpowiadającą pętli for.
Zadanie: Zidentyfikuj etykiety i instrukcje skoku, które implementują pętlę. Która instrukcja sprawdza warunek (i < 5), a która skacze na początek pętli? Wskaż instrukcję odpowiadającą inkrementacji (++i).

Analiza instrukcji warunkowej i arytmetyki wskaźników:
Zmodyfikuj powyższy program lub napisz nowy, dodając instrukcję if oraz operacje na wskaźnikach, np.:

```
#include <iostream>
using namespace std;
int main() {
   int sum = 0;
   for (int i = 0; i < 5; ++i) {
      sum += i;
   }

if (sum < 10) {
   sum = 100;
} else {
   sum = 200;
}

int arr[3] = {1,2,3};
int *p = arr;
*(p + 1) = 42;

   cout << "Sum = " << sum << endl;
}
```

Skompiluj z opcjami:

```
g++ -O0 -S program.cpp -o program.s
```

Zadanie: Znajdź fragment asemblera odpowiadający instrukcji if/else – powinien zawierać porównanie (cmp) oraz skok warunkowy (jge, jl lub je). Zweryfikuj, że jeden blok ustawia sum na 100, a drugi na 200. Następnie znajdź instrukcję odpowiadającą operacji na wskaźniku, gdzie 42 jest zapisywane.

Ćwiczenie wywołań funkcji zgodnie z ABI:
Napisz program w C++ zawierający funkcje, np.:

```
#include <stdio.h>
int add(int a, int b, int c) {
   return a + b + c;
}
int main() {
   int x = add(1, 2, 3);
   printf("%d\n", x);
}
```

Skompiluj program poleceniem:

```
g++ -O0 -S func.cpp -o func.s
```

Zadanie: W asemblerze funkcji add zidentyfikuj, w których rejestrach trafiają argumenty a, b i c (sprawdź odniesienia do %edx, %esi, %edi). Zauważ, że w funkcji main przed wywołaniem add widoczne są instrukcje przenoszące wartości 1, 2, 3 do tych rejestrów. Potwierdź, że wynik zwracany jest w %eax, a następnie wykorzystywany (np. przekazywany do printf).

Ćwiczenie dotyczące pamięci dynamicznej i układu pamięci:
Napisz program, który dynamicznie alokuje pamięć i używa zmiennej globalnej lub statycznej, np.:

```
#include <stdio.h>
int globalVar = 123;
int main() {
   static int staticVar = 456;
   int localVar = 789;
   int *heapVar = (int*) malloc(sizeof(int));
   *heapVar = 999;
   printf("Adresy: global=%p, static=%p, local=%p, heap=%p\n",
        (void*)&globalVar, (void*)&staticVar, (void*)&localVar, (void*)heapVar);
   free(heapVar);
}
```

Skompiluj i uruchom program – wypisze on adresy zmiennych.
Zadanie: Zaobserwuj adresy. Które z nich są blisko siebie, a które daleko? Zazwyczaj globalne i statyczne zmienne będą blisko (np. w zakresie 0x55555555xxxx), zmienne lokalne na stosie będą miały wysokie adresy (np. 0x7ffd...), a adres przydzielony przez malloc – będzie inny (np. 0x55555575b260). To potwierdzi klasyczny układ pamięci.