13.3.4. Konfigurowanie architektury Tritona

Funkcja `main` narzędzia `backward_slicing` wywołuje `set_triton_arch`, by dostosować Tritona do zestawu instrukcji pliku binarnego i uzyskać nazwę rejestru wskaźnika instrukcji używanego przez daną architekturę. Listing 13.3 pokazuje sposób implementacji `set_triton_arch`.

```
static int
set_triton_arch(Binary &bin, triton::API &api, triton::arch::registers_e &ip)
{
❶  if(bin.arch != Binary::BinaryArch::ARCH_X86) {
     fprintf(stderr, "Unsupported architecture\n");
     return -1;
   }

❷  if(bin.bits == 32) {
❸    api.setArchitecture(triton::arch::ARCH_X86);
❹    ip = triton::arch::ID_REG_EIP;
   } else if(bin.bits == 64) {
❺    api.setArchitecture(triton::arch::ARCH_X86_64);
❻    ip = triton::arch::ID_REG_RIP;
   } else {
     fprintf(stderr, "Unsupported bit width for x86: %u bits\n", bin.bits);
     return -1;
   }

   return 0;
}
```

Listing 13.3: backward_slicing.cc (ciąg dalszy)

Funkcja ta przyjmuje trzy parametry: referencję do obiektu `Binary`, zwracanego przez loader binarny, referencję do API Tritona i referencję do `triton::arch::registers_e`, w którym przechowuje nazwę rejestru wskaźnika instrukcji. Jeżeli wszystko w porządku, `set_triton_arch` zwraca 0, jeżeli wystąpił błąd, zwraca -1.

Najpierw `set_triton_arch` upewnia się, że ma do czynienia z plikiem binarnym na x86 (32- albo 64-bitowym) ❶. Jeżeli nie, zwraca błąd, ponieważ na razie Triton nie obsługuje innych architektur, niż x86.

Jeżeli nie ma błędu, `set_triton_arch` sprawdza długość słowa dla pliku ❷. Jeżeli plik używa 32 bitów, konfiguruje Tritona w trybie 32 bity na x86 (`triton::arch::ARCH_X86`) ❸ i ustawia `ID_REG_EIP` jako nazwę rejestru wskaźnika instrukcji ❹. Podobnie, jeżeli jest to plik 64-bitowy, ustawia architekturę w Tritonie na `triton::arch::aRCH_X86_64` ❺ i ustawia `ID_REG_RIP` jako wskaźnik instrukcji ❻. Aby skonfigurować architekturę Tritona, używasz funkcji `api.setArchitecture`, która przyjmuje typ architektury jako jedyny parametr.

13.3.5. Obliczanie „wycinka wstecznego"

Aby obliczyć i wydrukować faktyczny „wycinek", `backward_slice` wywołuje funkcję `print_slice`, kiedy emulacja natrafi na adres, od którego należy wycinać. Możesz obejrzeć implementację `print_slice` na listingu 13.4.

```
static void
print_slice(triton::API &api, Section *sec, uint64_t slice_addr,
            triton::arch::registers_e reg, const char *regname)
{
  triton::engines::symbolic::SymbolicExpression *regExpr;
  std::map<triton::usize, triton::engines::symbolic::SymbolicExpression*> slice;
  char mnemonic[32], operands[200];

❶ regExpr = api.getSymbolicRegisters()[reg];
❷ slice = api.sliceExpressions(regExpr);

❸ for(auto &kv: slice) {
    printf("%s\n", kv.second->getComment().c_str());
  }
❹ disasm_one(sec, slice_addr, mnemonic, operands);
  std::string target = mnemonic; target += " "; target += operands;

  printf("(slice for %s @ 0x%jx: %s)\n", regname, slice_addr, target.c_str());
}
```

Listing 13.4: backward_slicing.cc (ciąg dalszy)

Jak pamiętasz, „wycinki" są obliczane względem konkretnego rejestru, który jest określony przez parametr `reg`. Aby obliczyć „wycinek", potrzebujesz wyrażenia symbolicznego skojarzonego z tym rejestrem tuż po emulowaniu instrukcji pod adresem „wycinka". Aby uzyskać to wyrażenie, `print_slice` wywołuje `api.getSymbolicRegisters`, która zwraca mapę wszystkich rejestrów na związane z nimi wyrażenia symboliczne, a następnie indeksuje tę mapę, by otrzymać wyrażenie związane z `reg` ❶. Z kolei uzyskuje „wycinek" wszystkich wyrażeń symbolicznych, które wnoszą wkład do wyrażenia `reg`, za pomocą `api.sliceExpressions` ❷, która zwraca „wycinek" w postaci `std::map`, mapującej całkowitoliczbowe identyfikatory wyrażeń na obiekty `triton::engines::symbolic::SymbolicExpression*`.

Masz teraz „wycinek" z wyrażeniami symbolicznymi, ale tak naprawdę potrzebujesz „wycinka" instrukcji asemblacji na x86. To właśnie jest celem komentarzy do wyrażeń symbolicznych, które wiążą każde wyrażenie z łańcuchami skrótowych nazw asemblacji i operandów dla instrukcji, która dała początek temu wyrażeniu. Stąd, by wypisać „wycinek", `print_slice` po prostu wykonuje pętlę nad „wycinkiem" z wyrażeniami symbolicznymi, uzyskuje komentarze do nich za pomocą `getComment` i drukuje je na ekranie ❸. Dla kompletności `print_slice` deasembluje też instrukcję, od której obliczasz ten „wycinek" i również ją drukuje ❹.

Możesz wypróbować program `backward_slice` na VM, uruchamiając go tak, jak pokazano na listingu 13.5.