CHALMERS |  GÖTEBORGS UNIVERSITET

AMBA: Interaktiv visualisering av symbolisk fuzzing

Kandidatarbete i Data och Informationsteknik

Loke Gustafsson, Samuel Kyletoft, Enayatullah Norozi, Albin Otterhäll, Clara Salberg, Linus Wallman

Chalmers Tekniska Högskola
Göteborgs Universitet
Institutionen för Data och Informationsteknik
Göteborg, Sverige 2023

Kandidatarbete 2023

AMBA: Interaktiv visualisering av symbolisk fuzzing

Loke Gustafsson
Samuel Kyletoft
Enayatullah Norozi
Albin Otterhäll
Clara Salberg
Linus Wallman

Institutionen för Data och Informationsteknik
Chalmers Tekniska Högskola
Göteborgs Universitet
Göteborg, Sverige 2023

AMBA: Interaktiv visualisering av symbolisk fuzzing
Loke Gustafsson, Samuel Kyletoft, Enayatullah Norozi, Albin Otterhäll, Clara Salberg, Linus Wallman

Handledare: Iulia Bastys, Institutionen för Data och Informationsteknik
Examinatorer: Wolfgang Ahrendt och Arne Linde, Institutionen för Data och Informationsteknik
Betygsättare: Joachim von Hacht, Institutionen för Data och Informationsteknik

Chalmers Tekniska Högskola
Göteborgs Universitet
Institutionen för Data och Informationsteknik
SE-412 96 Göteborg
Telefon +46 31 772 1000

Göteborg, Sverige 2023

Abstract

This thesis introduces AMBA, an interactive system for visualising the symbolic execution of binary executables in combination with symbolic fuzzing. AMBA's graphical user interface visualises the different execution paths of binary executables in real time. The user can then based on the visualisations interactively prioritise execution paths for analysis. Information about the different execution paths is generated with the help of symbolic fuzzing, which is enabled by the S^2E engine. S^2E supports full-system emulation, which makes AMBA stand out from existing tools. AMBA is evaluated in comparison to a selection of existing tools for analyzing programs on the machine code level.

Keywords: symbolic execution, binary analysis, S^2E, fuzzing.

Sammandrag

Det här kandidatarbetet introducerar AMBA, ett interaktivt system för att visualisera symbolisk exekvering av exekverbara binärfiler i kombination med symbolisk fuzzing. AMBAs grafiska användargränssnitt visar de olika exekveringsvägarna för exekverbara binärfiler i realtid. Användaren kan sedan baserat på visualiseringarna interaktivt prioritera olika exekveringsvägar för analys. Information om de olika exekveringsvägarna genereras med hjälp av symbolisk fuzzing, vilket möjliggörs av S^2E-motorn. S^2E stödjer fullsystemsemulering, vilket gör att AMBA skiljer sig från befintliga verktyg. AMBA utvärderas genom jämförelse med ett urval befintliga verktyg för att analysera program på maskinkodsnivå.

Tack

Först och främst vill vi tacka vår handledare Iulia Bastys som under projektets gång har hjälpt oss mycket med feedback och reda ut frågetecken. Tack så mycket!

Vi vill även tacka Francisco ”Klondike” Blas Izquierdo Riera som utöver handledningen av hans egna ordinarie kandidatgrupper lagt ner tid för att ge oss feedback. Tack så mycket!

Slutligen vill vi även tacka Joachim von Hacht som i slutet av projektet hjälpe oss att komma över mållinjen. Tack!

Innehåll

Figurförteckning

Begreppslista

Assembler Assembler (jfr. eng. *assembly*) är ett lågnivåspråk som direkt kan översättas till maskinkod.

Basic block *Basic block* (jfr. sv. grundblock) är en sekvens av instruktioner som saknar hopp eller förgreningar bortsett från början och slutet av blocket. Är oftast expanderade till att bli så långa som möjligt

Black-box analys Black-box analys (jfr. sv. svartlådeanalys) är en analys av ett objekt som endast betraktar dess yttre utseende och beteende, till skillnad från white-box-analys som även betraktar vad som händer inuti.

Binär En binär (jfr. eng. *binary*) är en fil som innehåller ett programs maskinkod och data.

Återanropsfunktion Återanropsfunktion (jfr. eng. *callback function*) är funktioner som läggs i en *hook* (jfr. sv. krok) för att exekveras vid ett visst tillstånd.

CTF CTF (*Capture the Flag*, jfr. sv. fånga flaggan) är i datorsäkerhetssammanhang är en utmaning eller övning i att hitta gömda ”flaggor” i program med avsiktliga säkerhetshål.

Dynamisk analys Dynamisk analys (jfr. eng. *dynamic analysis*) är när en binär analyserar genom att exevera binären i en kontrollerad miljö för att i detalj registrera vad binären gör.

ELF ELF (*Executable and Linkable Format*, jfr. sv. exekverbart och länkbart format). Filformatet för exekverbara filer på Linux och liknande system. Innehåller maskinkod och länkar till andra ELF-filer.

Exekveringsmotor En exekveringsmotor (jfr. eng. *execution engine*) är ett program som exekverar ett programs instruktioner.

Fuzzing Fuzzing är att generera indata till ett system i ett försök att hitta buggar eller genom frånvaron av dåligt beteende betryggas i systemets kvalité. Vissa verktyg genererar ny indata med genetiska algoritmer och vissa använder *white-box* binärinstrumentation för att evaluera indata.

Grey-box analysis *Grey-box analysis* (jfr. sv. *grålådeanalys*) är en teknik som kombinerar *black-box* och *white- box* för att bilda en bredare analys. Ett användingsområde kan vara där dokumentationen är begränsad om ett programs interna struktur.

Heap Heapen är ett område i minne som samtliga trådar i en process har tillgång till. Används för objekt som man inte vet storleken på innan man exekverar programmet; samt objekt som ska delas mellan en process trådar.

Heuristik Heuristik är en praktisk metod för att lösa problem baserat på tidigare erfarenhet. En metod som är inte är fullständig utan baserad på tumregel.

Hook *Hook* (jfr. sv. krok) är en lista på återanropsfunktioner som ska köras vid ett specifikt tillstånd.

Instrumentering Instrumentering (jfr. eng. *instrumentation*) är mätning av ett programs prestanda. Används för att hitta buggar och hitta kontrollflöden.

IPC IPC (*Inter-process communication*, jfr. sv. interprocesskommunikation) är ett samlingsbegrepp för olika tekniker för att trådar i olika processer att kommunicera med varandra.

KLEE KLEE är den symboliska exekveringsmotorn som används av S^2E.

Maskinkod Maskinkod (jfr. eng. *machine code*) är digital kod som CPU:n kan tolka och arbeta med.

Tillståndssammanslagning Tillståndssammanslagning (jfr. eng. *state merging*) möjliggör att antingen automatiskt eller manuellt slå ihop olika stigar mellan tillstånd. Används för att öka presdandan.

Systemanrop Systemanrop (jfr. eng. *syscall*) är mjukvaruavbrott som program

använder för att kunna anropa operativsystemskärnan på ett sätt som liknar funktionsanrop.

Pekare Pekare (jfr. eng. *pointer*) är en minnesadress som vanligtvis pekar på ett värde på stacken eller heapen.

Process En process är en operativsystemabstraktion som speciellt innehåller en memory map(jfr. sv. minneskarta) tillsammans med ett antal trådar och andra operativsystemsresurser såsom fildeskriptorer (jfr. eng. *File descriptor*).

QEMU QEMU (*Quick Emulator*, jfr. sv. snabb emulator) är ett välunderhållet öppen källkods emulatorramverk med stöd för många plattformar och som stöder både *user space* emulering av en process samt emulering av ett helt system.

Utpressningsprogram Utpressningsprogram (jfr. eng. *ransomware*) är en sorts skadeprogram som syftar till att göra ett it- system oanvändbar genom kryptering av data som endast kan avkrypteras med en nyckel.

RCE RCE (*Remote code execution*, jfr. sv. fjärrkodsexekvering) är en sårbarhet som tillåter körning av godtyckliga kommandon eller kod på en måldator.

Reverse Engineering *Reverse engineering* (jfr. sv. demontering eller baklängeskonstruktion) är en process där man från en befintlig artefakt försöker återskapa källkodsinstruktionerna skrivna av de ursprungliga utvecklarna av artefakten.

Runtime *Runtime* (jfr. sv. körtid) är tidsspannet från det att ett program börjar exekveras, tills dess att programmet har slutat att exekveras.

S^{2}E S^{2}E (*The Selective Symbolic Execution Platform*, jfr. sv. Den selektiva exekveringsplattformen) är en platform som tillhandahåller symbolisk exekvering inuti den virtuella maskinen QEMU.

SMT-lösare SMT-lösare (jfr. eng. *Satisfiability Modulo Theories solver*) är ett program som kan lösa ekvationssystem för olika matematiska objekt. Exempel på teorier är modulär aritmetik eller bitvektorer. En SMT-lösare kan till exempel lösa ekvationer konstruerade i symbolisk exekvering.

Stack Område i minnet som reserveras för varje enskild tråd. På stacken förvaras värden där man redan vid kompilering känner till värdets minnesstorlek.

Starkt anslutna komponenter Starkt anslutna komponenter (jfr. eng. *Strongly Connected Components*) är en riktad delgraf där varje nod har en väg sådant att det går att nå alla andra noder i delgrafen.

Statisk analys Statisk analys (jfr. eng. *static analysis*) är binäranalys där man endast utifrån maskinkoden på disk försöker dra slutsater om binärens beteende.

Symbolisk exekvering Symbolisk exekvering (jfr. eng. *symbolic execution*) är att tilldela variabler symboliska, i motsats till konkreta värden under programmets exekvering. Med denna analysteknik kan enskilda körningar ge information som annars hade krävt en uttömmande sökning.

Symbolisk fuzzing Symbolic fuzzing är en teknik som kombinerar symbolisk exekvering och fuzzing. Kombinationen bevarar kodstrukturen och kan samtidigt lösa komplexa symboliska restriktioner.

Symbolisk operation Symbolisk operation (jfr. eng. *symbolic operation*) är operationer med symboliska värden.

Tråd Thread (jfr. eng. *thread*) är en av flera parallella instruktionssekvenser inom en process.

White-box analysis *White-box analysis* (jfr. sv. vitlådsanalys) är en analysmetod som behandlar en applikations interna struktur i kontrast till black-box analys som enbart betraktar funktionaliteten.

1 Inledning

Programförståelse är processen där människan får kunskap om hur ett datorprogram eller mjukvarusystem fungerar [1]. Programförståelse är ofta, helt eller delvis, en informell process. Programförståelse används till exempel när nyanställda läser dokumentation för att förstå hur ett mjukvarusystem är uppbyggt. Ett annat exempel är när säkerhetsforskare undersöker skadlig programvara (jfr. eng. *malware*) för att kunna utveckla motmedel [2]. I fortsättningen använder vi ordet förståelse som en förkortning av programförståelse.

För att bilda sig en förståelse av ett program formulerar man ofta hypoteser om programmet, för att sedan testa om hypotesen stämmer [3]. Dessa hypoteser kan vara övergripande, som till exempel om programmet kan utgöra ett säkerhetshot. Hypoteserna kan även vara specifika och fokuserade på detaljer, som till exempel om en specifik formell specifikation är implementerad på ett korrekt sätt eller vilka filsystemkataloger programmet använder sig av.

1.1 Metoder för datorn

Vissa hypoteser lämpar sig för att undersökas av en dator som effektivt kan utföra automatiserade tester. Till exempel kan man ha implementerat en sorteringsalgoritm som ska sortera olika värden i en specifik ordning. En hypotes är då att implementationen av algoritmen är korrekt. Hypotesen kan då testas genom att skapa ett antal testfall som kontrollerar att utdatan blir korrekt. Däremot kan testning i det generella fallet aldrig verifiera att ett program är helt korrekt.

En testmetod som automatiskt kan generera testfall och potentiellt hitta buggar är fuzzing [4]. Fuzzing går ut på att generera indata för att sedan undersöka om indatan kan orsaka icke önskvärt beteende eller krascher hos programmet [4]. Ett sätt att generera indata inom fuzzing är att betrakta typen på indatan och försöka testa olika kategorier som kan orsaka felaktiga beteenden. Om indatan exempelvis är en numerisk typ kan man generera ett litet värde, ett stort värde och ett negativt värde för att testa. Men om endast ett specifikt värde leder till ett visst beteende är det sannolikt att detta värde aldrig gissas och därmed inte inträffar.

Symbolisk fuzzing är en variation på fuzzing som utforskar flera möjliga vägar

genom ett program utan att köra det med alla konkreta värden som indata [5]. Istället för att använda konkreta värden används variabler för att symbolisera indata och programtillstånd. Genom att spåra villkor för dessa variabler kan man generera testfall som täcker olika vägar genom programmet. Målet är att undersöka så många vägar som möjligt för att utforska programmets olika möjliga tillstånd som kan nås med olika indata.

För att demonstrera detta kan vi betrakta funktionen i figur 1.1. I funktionen kan det uppkomma en krasch om indatan x har värdet 2 eftersom nämnaren i returuttrycket $\frac{1}{2-x}$ blir 0 och således leder till en division med 0. Med symbolisk fuzzing representeras x som en symbol istället för ett konkret värde. Därefter exekveras funktionen rad för rad. Vid villkoret $2 \leq x < 10$ övervägs båda möjligheter: när villkoret är sant och när det är falskt. Symboliskt sett betyder det att vi har två exekveringsvägar: vägen där villkoret är sant, där $2 \leq x < 10$, samt vägen där villkoret är falskt, där $x < 2$ eller $x \geq 10$. Den andra vägen avslutas när funktionen returnerar x. Men den första vägen delas ytterligare när divisionen ger ett division-med-noll-undantag för $x = 2$ och returnerar den beräknade kvoten för övriga x.

```
def foo(x):
    if  2 <= x < 10:
        return 1 / (2 - x)
    return x
```

Figur 1.1: Exempelfunktion som beräknar $\frac{1}{2-x}$ om $2 \leq x < 10$, annars returnerar x. Ett division-med-noll-undantag uppstår om $x = 2$

För program med tillräckligt mycket indata sker en så kallad kombinatorisk explosion där en fullständig sökning av möjlig indata ej kan genomföras då antalet fall växer exponentiellt med storleken på indatan. Ett motsvarande problem, vägexplosionsproblemet, uppstår i symbolisk fuzzing eftersom antalet möjliga vägar genom programmet kan växa exponentiellt med längden på den längsta vägen.

1.2 Metoder för människan

Andra typer av hypoteser kan vara lämpliga att undersökas av en människa, då en människa kan använda sina erfarenheter och kunskaper för att se mönster, dra slutsatser samt bilda sig en övergripande förståelse av ett program. Till exempel kan en människa dra slutsatser från testresultat från manuella eller automatiska

tester. Om en bugg hittas vid testning kan en människa med kunskaper inom IT-säkerhet utvärdera hur säkerhetskritisk buggen är. Ett annat exempel på när det är lämpligt att en människa undersöker en frågeställning är när metoden eller processen som används är svår att automatisera. En process som är svår att automatisera är *reverse engineering* som handlar om att rekonstruera en modell av programmets arkitektur genom deduktiva resonemang för att kunna exempelvis återskapa källkoden för ett program från binärkoden [6]. *Reverse engineering* används exempelvis inom skadeprogramanalys då man ofta inte har tillgång till skadeprogrammets källkod.

1.3 Problem och motivation

I de flesta befintliga verktyg för att analysera program är det antingen människan eller datorn som testar hypoteser. Befintliga verktyg där framförallt människan testar hypoteser präglas av mycket manuellt arbete. Detta innefattar verktyg från dekompilatorer där människan läser pseudokod konstruerad utifrån maskinkoden till debuggers där människan kör programmet steg för steg. Befintliga verktyg där framförallt datorn testar hypoteser präglas av att övergripande insikter missas, av kombinatoriska explosioner såsom vägexplosionsproblemet samt att potentiell programförståelse går förlorad i klumpig kommunikation med användaren. Detta innefattar bland annat de fuzzers som endast ger användaren indata som orsakar vissa specifika beteenden.

Människan och datorn har olika styrkor när det kommer till att undersöka de hypoteser man tagit fram för att öka förståelsen. I sammanhang där det är kritiskt att bilda en djup förståelse är det därför intressant att ge människan möjlighet att samarbeta med datorn för att kunna utnyttja bådas fördelar. Samarbete mellan människan och datorn kan leda till en mer komplett förståelse som i sin tur kan leda till åtgärdade säkerhetshål och buggar.

För att till en människa effektivt förmedla den data som en dator genererar i sin analys behövs ett verktyg som översätter kopiösa mängder data till ett mänskligt överskådligt format. Genom att visualisera denna data kan datorn utöver att kommunicera detaljer även bidra till en mer generell programförståelse på högre abstraktionsnivå. Om verktyget dessutom är interaktivt kan människan med sin generella programförståelse styra datorns analys på ett sätt som undviker fallgropar och ytterligare gynnar den abstrakta förståelsen.

Vi tror att interaktiv visualiserad symbolisk fuzzing, genom att utnyttja både datorn och människans styrkor, kan komplettera befintliga verktyg.

1.4 Mål

Projektet ämnar utveckla AMBA (*Automatisk och Manuell BinärAnalys*), ett verktyg som använder symbolisk fuzzing för att analysera ett program på maskinkodsnivå. Den information AMBA utvinner ska presenteras visuellt för användaren i ett grafiskt gränssnitt. Genom det grafiska gränssnittet ska användaren kunna påverka de beslut AMBA tar i sin vägutforskning.

Målet med AMBA är att genom interaktiv visualisering låta användaren övervaka symbolisk fuzzing.

1.5 Avgränsningar

AMBA kommer att fokusera på binäranalys och inte analys av program på källkodsnivå. AMBA ska demonstrera potentialen i interaktiv visualiserad symbolisk fuzzing, men behöver inte nödvändigtvis uppnå denna potential genom att vara ett praktiskt verktyg. AMBA har inte det primära syftet att angripa vägexplosionsproblemet utan fokuserar på användbar visualisering.

1.6 Rapportstruktur

Avsnitt 2 beskriver bakgrund och förklarar teori i mer detalj. Efterföljande avsnitt (avsnitt 3) beskriver tidigare arbete och kategorisering av olika befintliga binäranalysverktyg. I avsnitt 4 beskrivs binäranalysverktyget AMBA och dess funktionalitet och efterföljs med resultat och implementationsbeskrivning i avsnitt 5. Avsnitt 6 tar upp och evaluerar projektets resultat, process, begränsningar och framtida arbete och förbättringar som kan göras. Evalueringen sker i form av en kvalitativ jämförelse av funktionalitet i AMBA och befintliga verktyg. Slutligen sammanfattas och diskuteras projektets bidrag i avsnitt 7.

2 Teori

I detta avsnitt förklaras bakgrund och problembeskrivning till projektet. Innan problemen kan beskrivas måste vi förstå några centrala idéer och teorier inom domänen. Först beskrivs olika metoder inom binäranalys, sedan ges en bakgrund till symbolisk exekvering, den underliggande tekniken inom symbolisk fuzzing, och vilka problem tekniken löser. Detta följs upp med en mer detaljerad beskrivning av symbolisk exekvering och fuzzing. Avslutningsvis, beskrivs problem som en symbolisk exekveringsmotor står inför och några möjliga lösningar till dessa problem.

2.1 Binäranalys

Inom skadeprogramanalys (jfr eng. *malware analysis*) och *reverse engeneering* är binäranalys nödvändig eftersom programmet som ska analyseras oftast endast tillgänglig som maskinkod [2].

Förutom när endast maskinkoden är tillgänglig är binäranalys också användbart för att upptäcka och undersöka minnessårbarheter. Det beror på det semantiska gapet mellan ett högnivå programspråk och maskinkod. Det är inte trivialt att argumentera för programmets beteende innan och efter kompilering till maskinkod och hur väl dessa motsvarar varandra. Genom att betrakta maskinkoden kan man "undersöka vad programmet faktiskt gör istället för vad man tror att det gör" [2]. Sammanfattningsvis kan sårbarheter introduceras i kompileringssteget på grund av fel i kompilatorer som är sällsynta men inte obefintliga [7].

Ett flertal metoder används för att analysera maskinkod:

1. disassemblera exekverbar binär och läsa dess funktioner för att förstå vad de gör [8].

2. dekompilera maskinkoden med ett verktyg som ger pseudokod, och sedan manuellt undersöka denna mer läsbara koden [8].

3. använda fuzzing på programmet, det vill säga automatiskt generera testfall tills ett orsakar en krasch eller annat oönskat beteende [9].

4. använda symbolisk fuzzing på programmet, där nya testfall genereras med hjälp av tidigare för att effektivt täcka fler vägar. [10].

För att bilda förståelse för ett program krävs att insikter är både korrekta och abstrakta. I detta avseende syftar korrekt på avsaknaden av felaktiga slutsatser och abstrakt på möjligheten att resonera om programmet generellt i motsats till att resonera om en specifik konkret indata i taget.

Metod 1-2, att läsa kod, kan ge en abstrakt förståelse av vad programmet gör, men för att verifiera huruvida resonemanget är korrekt krävs hypotestestning vilket kräver att programmet körs. Således går det inte att skaffa sig en korrekt förståelse genom att enbart läsa kod.

Metod 3-4, att köra programmet på testfall, ger framförallt en black-box-förståelse av programmet. Tillgången till exekverbar binär och exekveringsmiljön används endast som ett verktyg för att generera nya testfall. Fuzzing och symbolisk fuzzing kan köras helautomatiskt och är korrekta. Vid fuzzing är en täckande sökning av indatarummet oftast omöjlig, och då kan den automatiska analysen ha missat ett kvalitativt annorlunda beteende. Dessutom ger en omfattande uppsättning indata-utdata-par inte användaren samma information som källkoden ger. Därmed är helautomatiska analysmetoder inte abstrakta. Det kan finnas gömda beteenden som är omöjliga att hitta med en automatisk analys, som till exempel ett hoppvilkor som beror på en kryptografisk hash av indatan. Detta är en fundamental begränsning som inte kan lösas med bättre verktyg. En analysmetod borde kunna peka ut var dess förståelse tar slut, snarare än att utelämna detta fullständigt vilket är vad avsaknaden av testfall visar sig som.

2.1.1 Automatiska och manuella metoder för binäranalys

För att klargöra distinktionen mellan manuella och automatiska metoder för binäranalys kan vi betrakta följande exempel:

```
# Givet sträng-input från stdin
s = input()
if sha256(s) == saved_hash:
  allow_access()
else:
  deny_access()
```

Figur 2.1: Exempelprogram där det är fördelaktigt att att använda manuella metoder för binäranalys

```
# Givet sträng-input från stdin
s = input()
if s == "secret":
  allow_access()
else:
  deny_access()
```

Figur 2.2: Exempelprogram där det är fördelaktigt att att använda automatiska metoder för binäranalys

I fall där det existerar kända konstanter, något som är typiskt i fall som involverar kryptografi i olika utsträckning, är det rimligt att tillämpa manuella metoder för att bilda förståelse för programmet. Genom att inspektera maskinkoden för program motsvarande figur 2.1 kan det enkelt hittas en konstant relaterad till sha256-algoritmen for att beräkna hash funktionen och därmed bilda förståelse för programmet. I detta fall är det dessutom orimligt att tillämpa automatiska metoder eftersom dessa, såsom konkolisk testning, genererar alltför stora symboliska representationer och hade i det ovan exemplet krävt att det går att hitta inversen till en given sha256-hash vilket idag är omöjligt och leder därmed till att alla vägar i programflödet inte undersöks.

Ett motsatt fall är exemplet i figur 2.2 som passar att undersökas med automatiska metoder eftersom det är tidskrävande att manuellt välja slumpvalda värden på `s` för att hitta den korrekta vägen. Istället lämpar symbolisk fuzzing sig väl i detta fallet. Eftersom symbolisk fuzzing väljer olika konkreta värden samtidigt som den tillämpar symbolisk exekvering med symboliska värden som följer den givna vägen, till exempel om `s == "annat"` vilket motsvarar att programmet väljer else-vägen `deny_access()`. Detta upprepas med nya exekveringsvägar och till slut hittar den indata som ger `allow_access()`-vägen.

2.1.2 Statisk och dynamisk binäranalys

En annan typ av kategorisering av olika analysmetoder som fokuserar på hur analysen genomförs delar metoderna i två grupper: statisk och dynamisk analys [11].

Statisk analys syftar på analys som går att göra utan att exekvera programmet som analyseras. Exempel på statisk binäranalys är metod 1–2 beskrivet tidigare i avsnitt 2.1, alltså att disassemblera binären och/eller visualisera kod [11].

Dynamisk analys går ut på att analysera ett program under exekvering. Exempel på dynamisk binäranalys är metod 3–6 i avsnitt 2.1. I alla fall krävs någon typ

av injektion av kod eller data i programmet i syfte att kunna extrahera viktig information under programmets exekvering [11].

2.1.3 Minnessårbarheter

I praktiken uppgör minnessäkerhetsbrister ungefär 70 procent av alla sårbarheter [12]. Minnessårbarheter kan påverka programflödet (jfr. eng. *control flow*) i ett program och alltså dess beteende. Det kan vara allvarligt om en angripare kan läsa delar av minnet som inte ska vara läsbara men det mest allvarliga är skrivaccess utanför minnesgränser. En angripare kan använda det för att till exempel exekvera godtyckliga kommandon på datorn som kör programmet [13], alltså *remote code execution*.

I en perfekt värld hade det varit fördelaktigt att undvika alla sårbarheter men det är inte möjligt på grund av mänskliga fel och konsekvenser av de verktyg vi använder. För att undvika, exempelvis, 100% av minnessårbarheter krävs användning av minnessäkra programspråk (jfr. eng. *memory-safe language*). Minnessäkra programspråk tilltar en kombination av körtids- (jfr. eng. *runtime*) och kompileringstidskontroller (jfr. eng. *compile time checks*) för att undvika minnessårbarheter. Däremot finns det än idag många skäl för att fortsätta använda minnesosäkra programspråk som C, såsom prestandaskäl, hårdvaubegränsningar, men främst äldre kod som är mödosam att skriva om [13]. Dessutom är de nyare programspråken, exempelvis Rust eller Swift, som kan konkurrera med C relativt nya och det krävs tid för övergången mellan språken.

Två exempel på minnessårbarheter är buffertöverflöden (jfr. eng. *buffer overflow*) och formatsträngsbuggar (jfr. eng. *format string bug*). Den vanligaste av de två är buffertöverflöden vilka uppstår av otillräckliga gränskontroller (jfr. eng. *bounds checks*), och utlöser möjlighet till åtkomst bortom gränserna för den minnesregion som avsetts. Angripare kan utnyttja detta för att korrumpera programmets normala beteende genom att skriva bortom gränserna. Formatsträngsbuggar uppstår när någon typ av indata tolkas och används som indata till vanliga textkonverteringsfunktioner som `printf`, som i sin tur använder konverteringsfunktioner för konvertering av olika data till text. Om inte en noggrann och tillräcklig utvärdering av indatan görs innan den används i en funktion som kan tolka formatsträngar, kan funktionen använda stacken och den data som råkar ligga där som argument till konverteringsfunktioner. Med noggrann utformning av indata kan en angripare utnyttja formatsträngsbuggar för att exekvera godtyckliga kommandon [13].

2.2 Symbolisk exekvering

Symbolisk exekvering motiverades av behovet för automatiska kontroller av olika programegenskaper. “Aspekter av intresse kan vara att ingen division med noll någonsin utförs, att ingen NULL-pekare någonsin avrefereras (jfr eng. *dereferenced*), att det inte finns någon bakdörr som kan kringgå autentisering och så vidare” [14]. För att kontrollera dessa egenskaper krävs kontroll av flertal olika exekveringsvägar vilket är svårt med vanlig exekvering (konkret exekvering) men enkelt genom symbolisk exekvering. Symbolisk exekvering tillåter dynamisk binäranalys och både manuella och automatiska sådana. Som tidigare nämnts, är symbolisk fuzzing en metod som är baserad på symbolisk exekvering.

Metoder baserade på symbolisk exekvering har används för att upptäcka både buffertöverflöden [15] och formatsträngsbuggar [16]. Metoderna har olika begränsningar, såsom prestandabegränsningar och andra begränsningar som symbolisk exekvering innehar, men dessa sårbarheter är också allmänt svåra att upptäcka automatiskt.

Att exekvera ett program symboliskt innebär att representera värden utefter programflödet som symboliska villkor (jfr eng. *constraints*), vilka sedan lösas av en SMT-lösare (jfr. eng. *Satisfiability Modulo Theories solver*). SMT-lösare är programvara för att lösa problem som rör satisfierbarheten hos formler, och använder matematiska teorier så som modulär aritmetik, talteori, m.fl [17]. En symbolisk körning representerar flera konkreta körningar eftersom de (symboliska) värden som används representerar grupper av konkreta värden vilka har gemensamt hur de påverkar programmets flöde [18].

Vägar i programmets kontrollflöde utforskas med symboliska uttryck som kallas för vägvillkor (jfr. eng. *path constraint*) och uttrycker de begränsningar som finns på programmets data — vilka egenskaper datan måste uppfylla för att just denna väg ska kunna följas [18]. Huruvida villkorsblock av program är nårbara kan evalueras eftersom de egenskaper som måste uppfyllas för att följa vägen dit dokumenteras under den symboliska körningen, och resulterar i fullständiga symboliska uttryck som en SMT-lösare kan appliceras på för att finna kokreta lösningar.

Eftersom de symboliska värdena har kapacitet att representera grupperingar av konkreta värden istället för enskilda sådana, utförs en generaliserad testning av programmet, som ger insikt i hur programmet beter sig givet en grupp av parametrar som alla på grund av någon eller några gemensamma egenskaper, orsakar gemensamma beteenden i programmet [19].

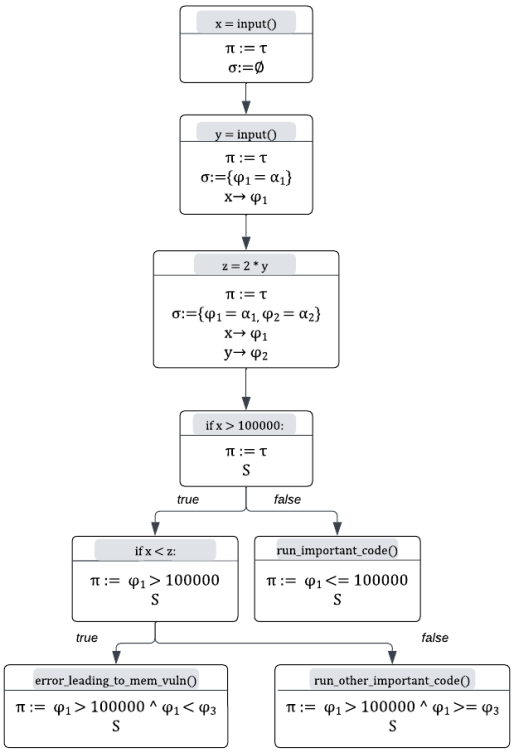
En symbolisk exekveringsmotor arbetar genom att först representera programmets indata som symboliska variabler, vilka vid starten inte har några begränsningar. När programflödet når en förgrening som baseras på någon av de symboliska variablerna, väljer motorn en gren och tillsätter villkoren avgjorde grenvalet på den symboliska variabeln för alla vägar som fortsätter utefter grenen. Även de operationer som utförs på indatan under körningen översätts till symboliska operationer på motsvarande symboliska variabler. När körning utefter grenen är slutförd repeterar motorn metodiken vid förgreningen för att utforska de andra alternativa grenarna. De tillståndsvillkor som en viss väg visas ha byggs därför successivt upp genom att motorn utökar de symboliska variablerna till villkorliga uttryck allt eftersom vägar följs [18].

Symbolisk exekvering kan leda till vägexplosionsproblemet, vilket uppstår i program vars förgreningar växer exponentiellt och resulterar i att en symbolisk exekvering aldrig terminerar [20]. Därför är det inte effektivt att alltid undersöka alla grenar i ett program. Exempel på metoder för att undvika vägexplosionsproblemet är tillståndssammanslagning (jfr. eng. *state merging*) och heuristik. Tillståndssammanslagning går ut på att slå samman ett antal exekveringsvägar genom att upptäcka exekveringsvägar som liknar varandra och slå samman dessa genom att kombinera deras villkor. Det krävs heuristik för att upptäcka fall där tillståndssammanslagning är applicerbart [14]. För att illustrera symbolisk exekvering används följande pseudokod:

```
x = input()
y = input()
z = 2 * y

if x == 100000:
  if x < z:
    # fabricated scenario of
    # memory vulnerability
    error_leading_to_mem_vuln()
  else:
    run_other_important_code()
else:
  run_important_code()
```

(a)



(b)

Figur 2.3: Exempel för att visa symbolisk exekvering: pseudokod (a) och vägvillkor och symboliskt tillstånd för alla vägar i pseudokoden angett i (b). $\mathcal{S} = \{\sigma := \{\phi_1 = \alpha_1, \phi_2 = \alpha_2, \phi_3 = 2\phi_2\}, x \to \phi_1, y \to \phi_2, z \to \phi_3\}$

Exempelprogrammet i figur 2.3a använder symbolisk exekvering för att hitta vilken indata som leder programexekveringen till de olika grenarna i programflödet. I många fall är det intressant att göra en uttömmande sökning och hitta alla möjliga vägar i ett program, något som är möjligt i detta program men inte i alla. Ett motexempel är komplexa program som på grund av vägexplosionsproblemet inte hittar alla möjliga vägar. Variablerna x och y sätts till symboliska värden som sedan används för att beräkna vägvillkoret och de symboliska uttryck som variablerna utvecklas till för en vald gren. Därefter används dessa uttryck och vägvillkor tillsammans för att bilda en ekvation som sedan kan lösas med hjälp av en SMT-lösare. Figur 2.3b visar hur det symboliska tillståndet förändras för alla möjliga grenar i programmet.

I 2.3b används π för att ange vägvillkoret vilket är initialt satt till $\top$ eftersom

villkoret är sant från början och σ används för att visa avbildningen för symboliska värden. π och σ populeras längs med exekveringen, och x och y representeras med symboliska värden. Beroende på vilken väg som väljs i exekveringen, uppdateras vägvillkoret. I andra noden sett uppifrån finns det två skillnader i jämförelse med den första noden: x tilldelas ϕ_1 som är en symbolisk representation av α_1. Efter den fjärde noden sett uppifrån görs ett val och vägvillkoret förändras beroende på vilken väg som tas – vägvillkoret uppdateras till $\phi_1 > 100000$ om $x < z$ och annars uppdateras det till $\phi_1 \leq 100000$. På samma sätt uppdateras π och ϕ längs andra exekveringsvägar och vilket till slut leder till ett komplett programflödesdiagram som beror på *x, y, z*. I varje nod kan ett konkret värde som uppfyller vägvillkoren evalueras.

2.3 Fuzzing

Fuzzing är en användbar automatiskt metod för att testa program i syfte att finna svårupptäckta problem såsom buggar och krascher.

Grundprincipen i fuzzing är att undersöka programmets beteende på mer varierad indata genom att generera oväntad, godtycklig eller felaktig data. Denna typ av genererad indata resulterar ofta i syntaktiskt eller semantiskt felaktig indata som inte kan hanteras av målprogrammet. Det är däremot inte trivialt att generera indata som täcker ett programs möjliga beteenden vilket har lett till bland annat mutationsbaserad fuzzing (jfr. eng. *mutation-based fuzzing*) och genereringsbaserad fuzzing (jfr. eng. *generation-based fuzzing*). Mutationsbaserad fuzzing muterar känd giltig indata, till exempel om strängen ‘fuzz’ är giltig indata kan detta muteras till ‘fuzzZZZZZ’. Om en användare exempelvis vill utföra fuzzing på bildhanterings-biblioteket libjpeg skulle detta innebära att skicka giltiga jpeg-bilder till fuzzern för att användas som seeds. Seeds i detta sammanhanget innebär de initiala värdena som sedan modifieras. Detta skiljer sig från genereringsbaserad fuzzing som genererar indata givet en modell för domänen — en fördel i jämförelse med mutationsbaserad fuzzing som kräver känd kvalitativ indata [21].

2.3.1 Symbolisk fuzzing

Symbolisk fuzzing är en white-box fuzzingmetod som nyttjar symbolisk exekvering för att maximera kodtäckning (jfr. eng. *code coverage*), det vill säga fuzzerns förmåga att traversera över samtliga kanter och noder i programmets kontrollflödesgraf. I motsats till grey-box-fuzzing så möjliggör symbolisk exekvering att en gren som inte tidigare tagits alltid väljs och ökar således kodtäckning [22]. Som beskrivet i avsnittet Symbolisk exekvering sker detta genom att emulera programmet

och ersätta indata med symboliska representationer, vilka enklast liknas med matematiska formler i form av algebraiska uttryck. Dessa uttryck byggs tillsammans med vägvillkor och kan skickas till en SMT-lösare för att evalueras till konkreta värden.

2.3.2 Problem med fuzzing

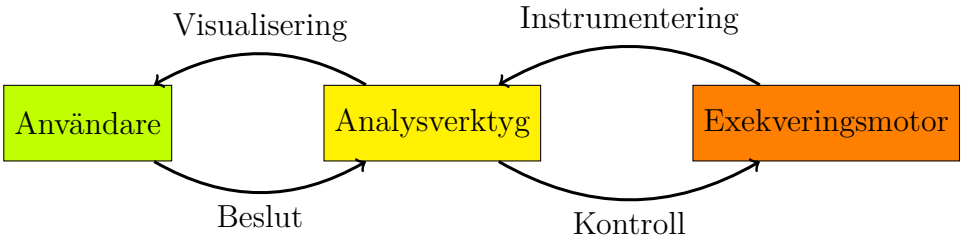
Ett problem är insikt om den underliggande kodstrukturen. En viktig egenskap som fuzzing har och som används för att beskriva dess effektivitet är kodtäckning. Black-box-fuzzing är exempel på en typ av fuzzing som saknar vetskap om den underliggande kodstrukturen och endast genererar slumpmässig indata, något som leder till ytlig testning av målprogrammet. I kontrast till black-box fuzzing finns det grey-box fuzzing, exempelvis AFL [23] som tillämpar binärinstrumentering, en teknik för att observera eller manipulera en binär genom att modifiera källkoden i binären. Genom binärinstrumentering kan information om kodtäckning för en viss indata utvinnas för att sedan försöka maximera kodtäckning med ny indata och därmed öka testytan. [22].

Fuzzing kräver ofta protokoll- eller domänkännedom för att kunna generera indata. Detta är svårare med komplexa kodbaser eller bibliotek som saknar trivial eller uppenbar indata vilket därmed leder till lägre kodtäckning.

White-box fuzzing är inte en allmän lösning till problemen med fuzzing, utan har också problem relaterade till prestanda, vägexplosion, och falskt positiva resultat. Det finns en stark korrelation mellan kodtäckning och bugg täckning [24] men eftersom white-box fuzzing är ett prestandakrävande verktyg kan detta leda till falskt positiva resultat eftersom den symboliska exekveringen även guidar fuzzingen längs med grenar som inte nödvändigtvis leder till buggar, eller är praktiskt omöjliga. Trots den starka korrelationen mellan kodtäckning och buggtäckning, kan buggar inte uteslutas vid testning baserat på hög kodtäckning.

2.4 Symbolisk exekveringsmotor

Kärnan i ett korrekt dynamisk binäranalysverktyg är en exekveringsmotor, en komponent som på ett korrekt vis kan exekvera programmet. Figur 2.4 visar förhållandet mellan användaren, analysverktyget och dess exekveringsmotor. Att köra ett program innebär att ladda binären och dess bibliotek, hoppa till startadressen och sedan köra enskilda instruktioner. Om binäranalysverktyget ska kunna använda metoder som använder symbolisk exekvering behöver denna exekvering av enskilda instruktioner också stödja symboliska variabler.



Figur 2.4: Schematisk bild av ett binäranalysverktyg byggt kring en exekveringsmotor

Symbolisk exekvering innehar, i praktiken, begränsningar i skalbarhet. En symbolisk exekveringsmotor som utför symbolisk exekvering behöver ta hänsyn till ett antal frågor gällande [14]:

Minne Motorn behöver spara komplexa datatyper och representera de på ett sätt som tillåter villkorslösning. Exekvering av större program kräver mer minne för att bokföra symboler och uttryck och därmed mer tid för exekvering [14].

Miljö Program i verkligheten kommunicerar på många sätt med sin omgivning. För program är detta en virtuell miljö bestående av filer, biblioteksanrop och IPC (*Inter Process Communication*). För att en exekveringsmotor ska vara så brett tillämpbar som möjligt behöver den också stödja flera sorters omgivningskommunikation och representera omgivningen på ett så komplett sätt som möjligt [14].

Vägexplosion Verkliga program innehåller loopar, rekursion, undantag och kombinationer av dessa som kan orsaka ett exponentiellt ökande antal exekveringsvägar i programflödet. Det är alltså osannolikt att en motor kan uttömmande utforska alla möjliga exekveringsvägar inom rimlig tid. Tillståndssammanslagning kan appliceras i vissa fall för att reducera antalet exekveringsvägar.

Villkorslösning SMT-lösare kan skala till komplexa kombinationer av villkor över hundratals variabler. Däremot kan icke-linjär aritmetik utgöra ett stort hinder för effektivitet. Dessutom finns det exempel på olösliga problem där SMT-lösare är inte tillämpbara [14].

3 Befintliga verktyg

All form av binäranalys kan delas in i statisk; dynamisk; eller en kombination av dem [2]. Statiska metoder bygger på att man översätter maskinkod till assembler utan att att exekvera binären, medan dynamiska metoder loggar exekverade instruktioner [2]. Dessa analysmetoder implementeras av en rad olika verktyg som idag används inom akademin och industrin. Listans syfte är att ge läsaren en översikt över fältet, och målet är inte att vara komplett.

3.1 Statisk disassemblering

En vanlig statisk binäranalys är statisk disassemblering (ofta bara kallat disassembering) [2]. Disassembering innebär att man försöker återskapa assemblerinstruktioner från maskinkoden i binären, utan att exekvera maskinkoden [2]. En av svårigheterna med att översätta maskinkod till assembler är att särskilja instruktioner från data [2]. Till exempel kan man inte veta om maskinkoden `0x8E` är decimalvärdet 142 eller x86-instruktionen `mov` [2].

Ett vanligt disassemblerverktyg är Ghidra, ett program utvecklat av NSA (National Security Agency) [8]. Ghidra har också en debugger och funktionsgraf. Debuggern ska underlätta binär debugging genom att integrera med andra funktioner i Ghidra. Funktionsgrafen låter användaren se hur programmet är uppbyggt visuellt och hur olika funktioner interagerar med varandra. Funktionaliteter kan utökas eller andra funktioner utvecklas genom plugins till Ghidra [25]. Ghidra tillåter även automatisering genom att skriva skript. Ett exempel är ett skript som hittar exempelvis sårbarhet i form av funktionsanrop till potentiellt osäkra API-anrop genom statisk analys [26]. Andra vanliga disassemblerverktyg är IDA Pro från Hex rays [27] och Binary Ninja från Vector 35 [28].

3.2 Dynamiska binäranalysramverk för symbolisk exekvering

En metod för dynamisk binäranalys är symbolisk exekvering. Eftersom exekveringen är symbolisk är det möjligt att utforska alla möjliga exekveringsvägar i

programmet, även de som inte är möjliga med konkreta indata. Både S^2E och SymQEMU är kraftfulla verktyg för att analysera binära program med symbolisk exekvering.

SymQEMU är byggt som en förlängning av QEMU, och använder en kombination av dynamisk binäröversättning och symbolisk exekvering för att analysera binära program som körs inuti emulatorn. SymQEMU utför kompileringsbaserad symbolisk exekvering, där den mellanliggande representationen först blir modifierad innan den översätts till värdarkitekturen, och är därför arkitekturoberoende utan att påverka prestandan [17]. Dessutom använder SymQEMU Linux *user-mode* emulering, vilket innebär att endast användarutrymmet (jfr. eng. *user space*) emuleras. Användarutrymmet innefattar alla program som inte körs av operativsystemets kärna, och genom emulering nås högre prestanda i kontrast med emulering av hela system [17].

S^2E är byggt ovanpå QEMU och utökar virtuella maskiner med stöd för symbolisk exekvering. S^2E erbjuder redskap för att fokusera utforskningen på delkomponenter av systemet och gör det även möjligt för användare att sätta in kod i målsystemet vid specifika punkter under exekvering. Det gör att användare kan anpassa analysprocessen efter sina behov [29]. S^2E kan analysera kod för de flesta processorarkitekturer men betalar för det med ökad komplexitet och prestanda [17].

I kontrast till SymQEMUs Linux user-mode emulering, emulerar S^2E hela målsystemet, vilket innebär att S^2E kan göra analys på ett större antal system. Inbyggda system är ett typexempel där det krävs helsystemsemulering då bland annat inbyggda system ofta har modifierade operativsystem.

Ett annat verktyg för att identifiera buggar och sårbarheter med hjälp av dynamisk symbolisk exekvering är SAGE (*Scalable Automated Guided Execution*). SAGE var det första verktyget som utförde dynamisk symbolisk exekvering på x86-binärnivå och använder flera avgörande optimeringar för att hantera stora exekveringsspår (jfr. eng. *execution traces*). För att skala upp till stora exekveringsspår använder SAGE flera tekniker för att förbättra hastigheten och minnesanvändningen för villkorsgenerering, exempelvis så kartläggs ekvivalenta symboliska uttryck till samma objekt och villkor som redan är tillagda hoppas över [10].

Binäranalysverktyget angr stödjer både statiska analyser såsom dekompilering till pseudo-C-kod och dynamiska analyser med hjälp av symbolisk exekvering. Användarens analyser utförs genom Python-skript som interagerar med angrs API för att kontrollera en symbolisk emulator skriven i Python för angr. Skript som använder angr kan användas för *reverse engineering*, sårbarhetssökning och kan även vara

del av exploateringsverktyg [30]. Tävlingen Cyber Grand Challenge organiserades 2016 av DARPA. I tävlingen skulle lag skriva helautomatiska system som hittar, korrigerar och angriper sårbarheter i CTF-liknande uppgifter. Det vinnande systemet, Mayhem, använde angr som symbolisk exekveringsmotor.

För att accelerera den symboliska exekveringen kan angr användas i kombination med Unicorn CPU-emulatorn, ett lättviktigt ramverk för emulering som stödjer många arkitekturer och baseras på QEMU [31]. Detta är möjligt med hjälp av komponenten *angr.engines.unicorn*. Genom att använda komponenten kan man exekvera med konkreta indata när det är möjligt och fördelaktigt men understött av symbolisk exekvering då flera exekveringsvägar behöver utforskas eller då programmets beteende inte är deterministiskt [32]. I angr kan det emulerade programmet manipulera filer och nätverksströmmar som representeras som Python-objekt. Dessa objekt läses från och skrivs till när det emulerade programmet utför systemanrop.

3.3 Automatiska fuzzers

Ytterligare en metod för binäranalys är fuzzing. Fuzzing innebär att testa ett program med en stor mängd olika indata. Målet är att simulera oväntade beteenden eller krashar om potentialen för dessa existerar. Fuzzing är därför ett effektivt medel för att undersöka om ett program innehåller sårbarheter [9].

AFL++ är en utveckling av den ursprungliga AFL (*American Fuzzy Lop*) fuzzern och har förbättrats med fler funktioner och bättre prestanda. AFL använder täckningsstyrd fuzzing (jfr. eng. *coverage-guided fuzzing*) för att generera testfall som är troliga att utlösa buggar och sårbarheter. AFL spårar kodtäckningen vid varje testfall och använder denna informationen för att styra hur tidigare testfall mutateras till nya. Även AFL++ muterar indata för att generera nya indata från befintliga, och använder feedback från programvaran som fuzzas för att styra fuzzing-processen [23].

3.4 Visualiserad symbolisk fuzzing

De i avsnitt 3.1, 3.2 samt 3.3 presenterade kategorierna av binäranalysverktyg är alla etablerade kategorier som AMBA inte faller under. Kategorin som AMBA tillhör kallar vi ”visualiserad symbolisk fuzzing” och detta avsnitt diskuterar det fåtal befintliga verktyg som vi anser faller inom kategorin.

Verktyget Symbolic Execution Debugger (SED) [33] är en debugger för Java 1.4 utan multitrådning, reflektion och flyttalsaritmetik. Användaren använder SED som ett Eclipse-plugin, i vilket valfri metod kan väljas och analyseras med symbolisk exekvering. Exekveringsträdet visas för användaren tillsammans med källkodsrader och de vägvillkor som tar exekveringen dit.

Att SED är utformat som en debugger som kan starta exekveringen vid valfri Javametod gör verktyget väl utformat till formell verifiering. Verktyget förlitar sig på tillgång till källkod och att användaren vet vilken indata som anses giltig för metoden. Begränsningen till enskilda metoder gör samtidigt den symboliska exekveringen mer överblicklig för användaren, då användaren i mindre grad själv behöver exkludera det som analysen inte bör lägga fokus på.

Att SED inriktar sig på en delmängd av Java innebär att det är mindre applicerbart för datasäkerhet och binäranalys då dessa sammanhang ofta kräver analys av exekverbara binärer, ofta utan tillgång till källkod. Eftersom Java är ett språk på högre nivå än maskinkod innebär samtidigt en inriktning mot Java att många implementationsdetaljer i hur koden skulle göras på en riktig dator kan ignoreras. Dynamisk minneshantering är ett exempel på detta, där analys av Javakod kan abstrahera bort exekveringsmiljöns anrop till `malloc()` och `free()` medan analys av maskinkod nödvändigtvis behöver hantera komplexiteten i deras implementationer. Symbolisk exekvering i en högnivåmiljö exkluderar därmed också många detaljer som verktyget eller användaren annars på annat vis hade behövt filtrera bort.

Verktyget SymNav [34] kan visualisera och kontrollera analysen av en exekverbar binär inuti angr:s symboliska exekveringsmotor. angr beskrivs i avsnitt 3.2.

Arbetsflödet består av att starta SymNav med en via kommandorad angiven binär och sedan vänta tills programmet forkar. Då visar det grafiska gränssnittet det symboliska trädet, delar av kontrollflödesgrafen, etc. Användaren kan då navigera den visualiserade informationen, och sedan antingen fortsätta eller starta om körningen med en angiven tidsbudget och minnesbudget. Till exempel kan användaren starta en sökning med standardparametrar i 10 sekunder och med 1 GB RAM. När söktiden löpt ut uppdateras informationen i det grafiska gränssnittet.

SymNavs grafiska gränssnitt består av tre paneler: en trädpanel som visar trädet av symboliska tillstånd, en kontrollflödespanel som visar kontrollflödesgrafen för enskilda funktioner och en styrpanel för att filtrera vilken information som ska visas.

Trädpanelen visar en kompakt representation av alla vägar av symboliska tillstånd som tagits. Dessutom visas en parallellkoordinatgraf där diverse nyckeltal för de symboliska tillstånden visas. Genom att interagera med denna graf kan användaren filtrera bort vägar efter nyckeltal, till exempel genom att endast visa vägar som någon gång anropar `malloc()`.

Kontrollflödespanelen visar en graf över alla grundblock inom en funktion. Noder innehåller den assemblykod som deras grundblock innehåller och kanter färgläggs efter hur många olika symboliska tillstånd som vandrat kanten.

Styrpanelen visar för användaren vilka filter de hittills applicerat och låter användaren filtrera ytterligare vilka symboliska tillstånd som ska visas i det grafiska gränssnittet. Dessutom låter styrpanelen användaren definera resursbudgeten, starta om körningen eller fortsätta körningen från nuvarande symboliska tillstånd. När en körning startas kan användaren ange att nuvarande filter i det grafiska gränssnittet ska omvandlas till villkor på den symboliska indatan samt prioriteringsregler för fortsatt sökning.

SymNav är därmed fasbaserad, med alternerande söknings- och presentationsfaser. SymNav ger användaren kraftfulla verktyg att filtrera ut intressanta tillstånd efter metriker såsom kodtäckningsgrad och minnesallokeringsmönster. Samtidigt ärver SymNav flera nackdelar från angr, såsom suboptimal prestanda [35] och begränsad miljömodellering då analysgränsen går vid processen med emulerade systemanrop snarare än vid en virtuell maskin. SymNav är skrivet för att kunna anpassas till andra exekveringsmotorer än angr, men detta är inte implementerat i dess nuvarande form.

4 AMBA

Detta arbete presenterar AMBA, ett binäranalysverktyg för att visualisera och styra symbolisk fuzzing interaktivt. I detta avsnitt kommer vi att beskriva funktionalitet i AMBA, AMBAs begräsningar samt hur man kan köra och använda AMBA.

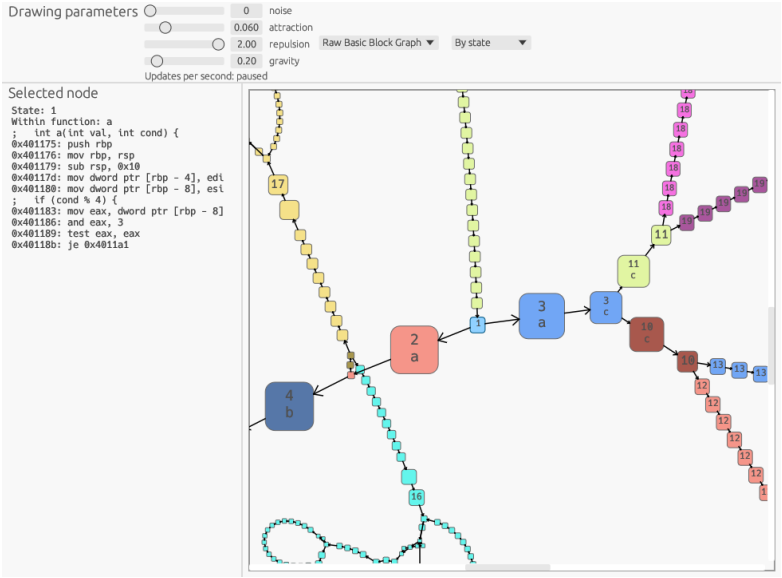
AMBA exekverar program med symbolisk indata och visar ett antal grafer över exekveringen. Dessa grafer uppdateras kontinuerligt under programmets exekvering. Graferna består av ett antal noder med metadata sammankopplade med riktade kanter. När en användare klickar på en nod visas den nodens metadata i en separat panel. Metadatan visas även om man zoomar in på en nod.

Den första grafen är en basic-block-graf. Figur 4.1 visar ett exempel. Här representerar varje tillstånd i grafen ett *basic block* i maskinkoden, alltså en sekvens maskinkod som exekveras utan hopp. Det grafiska gränssnittet visar blockets adress, det symboliska tillståndet blocket exekveras i och den disassemblerade maskinkoden. Om binären innehåller debugdata visas även funktionsnamn och om både debugdata och källkoden är tillgänglig visas de källkodsrader som gett upphov till maskinkoden.

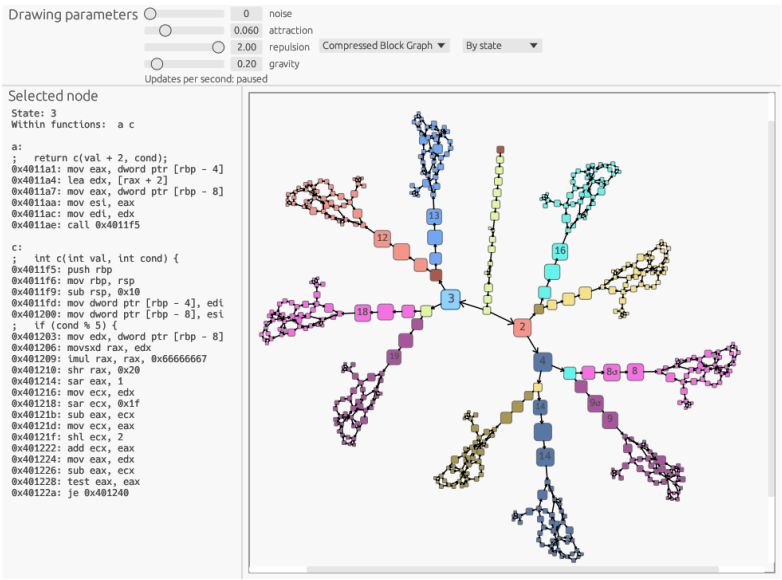
Den andra grafen är en komprimerad basic-block-graf. Figur 4.2 visar ett exempel. Den visar samma graf som den första, men med linjesubgrafer sammanslagna till samma nod. Precist uttryckt kantkomprimeras alla kanter till noder som har ingrad och utgrad exakt ett. Sidopanelen visar samma information som för basic-block-grafen, alltså assemblerkod och debugdata, men för hela spannet av sammanslagna noder.

Den tredje grafen visar symboliska tillstånd. Figur 4.3 visar ett exempel. Ett tillstånd utgörs av exekveringsspannet uppdelat vid varje gren som görs baserat på ett symboliskt uttryck. Sidopanelen visar tillståndets namn samt det grennummer som identiferar det föränderliga tillståndet i S^2E. Då föräldrar inte kan samexistera i tid med sina barn använder S^2E grennummer för att hänvisa till nu körande tillstånd.

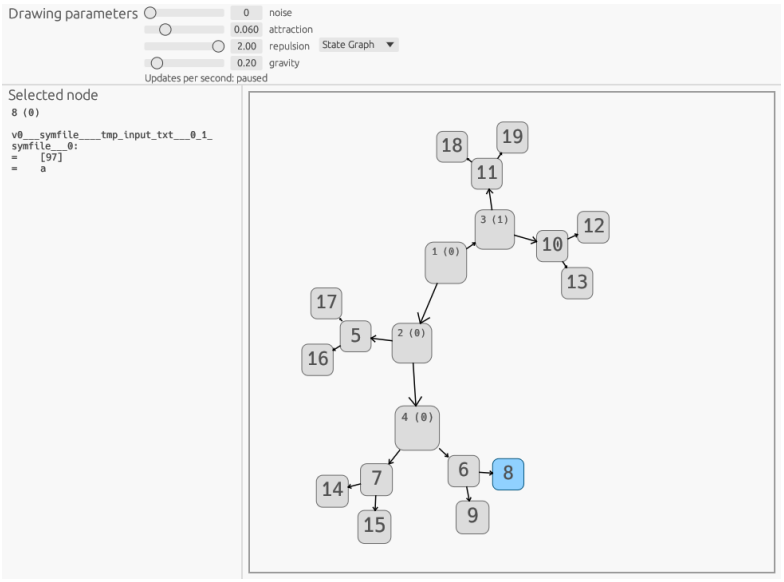
Den fjärde grafen är en sammanslagen basic-block-graf, som visar basic-block ut-



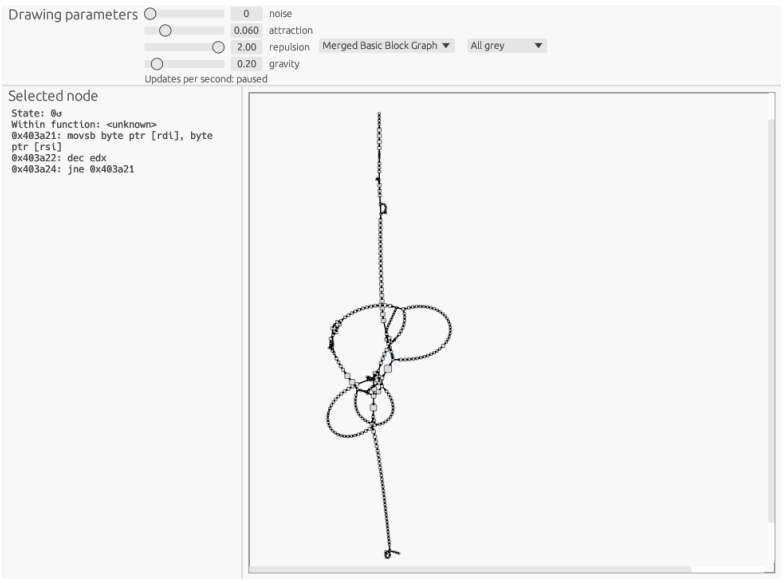
Figur 4.1: AMBAs basic-block-graf för testprogrammet control-flow



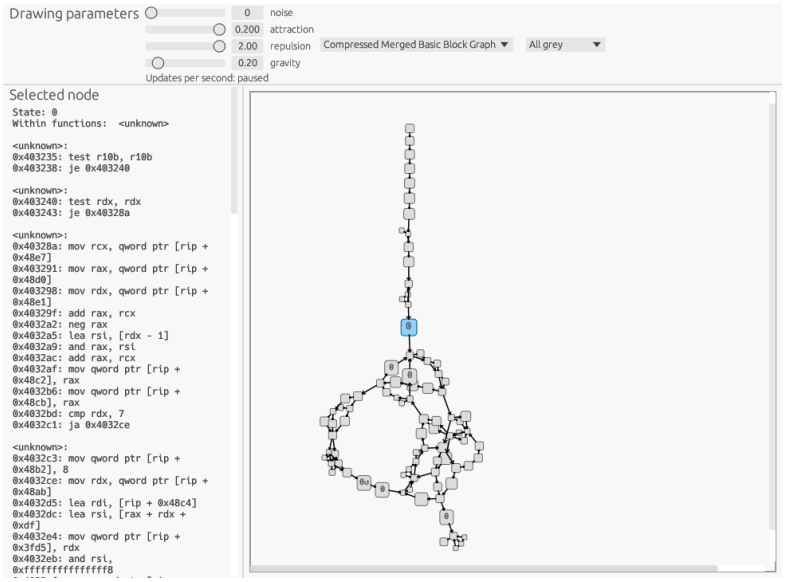
Figur 4.2: AMBAs komprimerade basic-block-graf för testprogrammet control-flow



Figur 4.3: AMBAs graf över symboliska tillstånd för testprogrammet control-flow



Figur 4.4: AMBAs sammanslagna basic-block-graf för testprogrammet control-flow



Figur 4.5: AMBAs komprimerade sammanslagna basic-block-graf för testprogrammet control-flow

an att delas vid tillståndsdelning. Precist uttryckt är det grafen som bildas från den första grafen, efter att varje par av noder vars metadata endast skiljer sig i det symboliska tillståndet har ersatts med en nod kopplad till unionen av de ursprungliga nodernas kanter. Figur 4.4 visar ett exempel.

Den femte och sista grafen är en komprimerad sammanslagen basic-block-graf, vilken konstrueras från den sammanslagna basic-block-grafen genom samma kompression som i den komprimerade block-block-grafen. Figur 4.5 visar ett exempel.

De två första graferna, alltså de komprimerade och okomprimerade basic-block-graferna, kan färgläggas utifrån symboliska tillstånd och starkt anslutna komponenter.

Utöver att visa dessa grafer går det också att välja noder i den symboliska tillståndsgrafen för att prioritera deras fortsatta evaluering. Detta gör det möjligt att undersöka intressanta subträd i grafer som är större än vad AMBA realistiskt sett kan besöka och visa. Detta kan användas för att kringgå vägexplosionsproblemet och undvika att enorma grafer skapas av program som inte terminerar. Genom att dubbelklicka på en nod i tillståndsgrafen kommer exekveringen fortsätta från motsvarande tillstånd.

Grafnodernas placering i 2D hittas genom iterativ lösning för lägst energi i ett

system av attraktion, repulsion och externa krafter. Utplaceringsalgoritmens parametrar kan konfigureras i realtid vid körning av användaren i toppanelen. Den här panelen låter även användaren välja vilken graf de vill visa.

Genom S^2E kör AMBA det analyserade programmet virtualiserat med KVM i QEMU. Detta innebär att den analyserade binärens beteende i hög grad inte påverkas av miljön på datorn som kör AMBA, vilket gör analysen mer reproducibel. Det innebär också att skadlig kod borde kunna analyseras dynamiskt i AMBA med låg risk för skadlig påverkan på användarens dator. Dock är AMBA omogen programvara som ej granskats för felkonfiguration eller liknande kopplat till QEMU eller S^2E, så vi rekommenderar inte att använda AMBA för analys av skadlig kod i dess nuvarande skick.

4.1 Begränsningar

Alla program kan inte hanteras av AMBA. Programmen som AMBA kör måste vara byggda för Linux x86-64 och kunna köras på Ubuntu-22.04-x86_64. Detta kan göras genom att bygga direkt för Ubuntu eller med full statisk länkning mot t.ex. libc-implementationen musl istället för dynamisk länkning mot glibc.

Den symboliska indatan är också begränsad till att komma ifrån stdin samt filer på disk. Symbolisk indata till AMBA i dess nuvarande skick kan inte skickas som programargument (`argc`, `argv`), miljövariabler (`envp`) eller i form av symbolisk hårdvara (t.ex. registret TSC/Time Stamp Counter). Detta är inte en fundamental begränsning utan endast oimplementerad funktionalitet som lämnas som utvecklingsmöjlighet.

En stor begränsning vid användandet av AMBA och symbolisk fuzzing i allmänhet är hanterandet av program med väldigt många symboliska tillstånd. Detta problem kallas vägexplosionsproblemet och beskrivs i inledningen, Sektion 1. AMBAs enda funktionalitet för att hantera detta är prioritering av exekveringstillstånd, vilket är ett trubbigt verktyg. Detta är till stor del en fundamental begränsning i symbolisk fuzzing och kräver yttligare metoder för att försöka angripa problemet.

4.2 Hur du kör AMBA

AMBA utvecklades i GitHub-repot `https://github.com/lokegustafsson/amba-bsc-thesis`. Dokumentationen i repot innehåller mer detaljerade installationsinstruktioner och allmän dokumentation.

AMBA kan köras i alla Linux-x86-64-miljöer med stöd för KVM med tillräckligt mycket diskutrymme (ca 20 GB) och RAM-minne (16 GB för att bygga AMBA, ej nödvändigt vid nedladdning av cachad binär). De tre installationsstegen består av att först installera pakethanteraren Nix, följt utav av att köra kommandot `nix build`. När detta kommandot körs byggs alla programvarukomponenter från LLVM och Linux till S^{2}E och AMBA självt. Detta bygget är deterministiskt reproducibelt från källkod och Nix möjliggör även korrekt cachelagring och servering över nätverk vilket vi använt för att minimiera ombyggnation under AMBAs utvecklingsprocess. Det sista installationssteget är att köra kommandot `nix run . -- init --download` som laddar ned en Ubuntu-22.04-x86_64-diskavbild byggd av S^{2}E-projektet . Bygget av diskavbilder för S^{2}E är inte bit-för-bit reproducibelt och därför är detta bygget inte packeterat i Nix.

AMBA körs med kommandot

```
./result/bin/amba-wrapped run path/to/a/recipe.json"
```

efter `nix build` eller ekvivalent

```
nix run . -- run path/to/a/recipe.json"
```

alltså genom att peka AMBA på en receptfil. En giltig receptfil följer ett schema varav en delmängd av fälten har implementerad funktionalitet. Figur 4.6 visar en receptfil för vilken alla fält är implementerade.

I receptfilen specifieras först ett antal filer som ska placeras i QEMU-gästen. Filerna specifieras med deras relativa sökväg i gästen, med konkreta filer angivna relativt `~/` och symboliska filer angivna relativt `/tmp/`. Filers innehåll kan specifieras som en sökväg relativt mappen receptfilen ligger i, vilket kopierar in en konkret fil. Filers innehåll kan också specifieras som symboliskt genom att ange en lista av byteadresser (t.ex. `0`) samt byteadressintervall (t.ex. `[42, 57]`) mellan vilka filen ska innehålla symboliska bytes. Symboliska filer behöver också konkreta värden som får förtur i tillståndsutforskningen. Dessa kan anges med antingen `"seed": "<förtursvärden>"` eller `"host_path": "<värddatorsökväg till förtursvärden>"`.

När AMBA körs med `nix run . -- run path/to/a/recipe.json` öppnas det grafiska gränssnittet samtidigt som QEMU/S^{2}E startas i bakgrunden. Grafuppdateringar skickas regelbundet tillbaka tills alla tillstånd utforskats eller programmet stänger ned. Både före och efter alla tillstånd utforskats kan graferna undersökas av användaren.

```
{
  "files": {
    "control-flow": "./control-flow",
    "input.txt": {
      "seed": "a",
      "symbolic": [0]
    }
  },
  "executable_path": "./control-flow",
  "stdin_path": "/tmp/input.txt"
}
```

Figur 4.6: Receptfilen för testprogrammet control-flow. Binären `control-flow` ligger bredvid receptfilen och kopieras till QEMU-gästens hemmapp. Dessutom placeras en 1 byte lång fil innehållande en symbolisk variabel i `/tmp/input.txt`. I gästen körs binären med den symboliska filen som `stdin`.

4.3 Varför S^{2}E

AMBA använder och är byggt ovanpå exekveringsmotorn S^{2}E. Två andra exekveringsmotorer som skulle kunna ha använts i detta arbete är SymQEMU och angr. Alla dessa verktyg har beskrivits översiktligt i avsnitt 3.2.

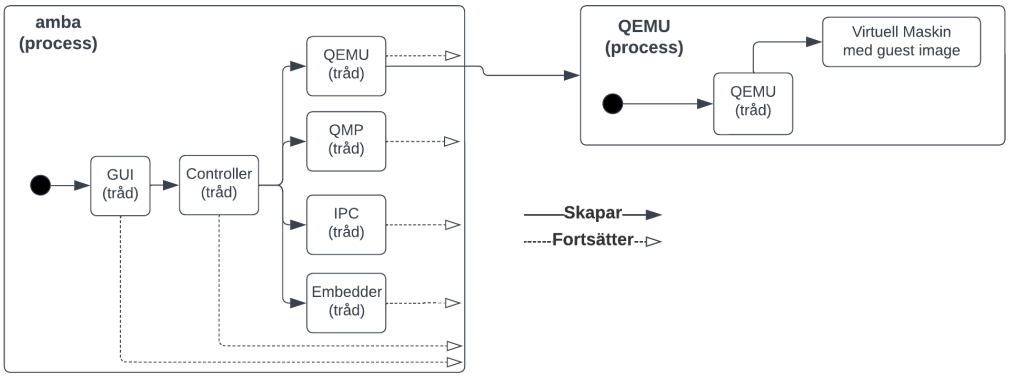
AMBA bygger inte på SymQEMU av tre anledningar. För det första har S^{2}E mer publikt tillgänglig dokumentation än SymQEMU, vilket bedömdes underlätta utvecklingsarbetet. För det andra genomför S^{2}Es utvecklare ett mer aktivt underhållsarbete jämfört med SymQEMUs, vilket bedömdes underlätta byggprocessen och minska sannolikheten att stöta på problemen i den första delen av projektet. Slutligen kan SymQEMU endast emulera enskilda program och genomför alltså inte fullsystemsemulation. Trots att AMBA endast analyserar enskilda program ansågs denna begränsning gynna S^{2}E genom att möjliggöra vidareutveckling till mer generell analys. SymQEMUs stora fördel relativt S^{2}E är dess bättre prestanda [35]. Detta bedömdes inte väga upp för dess nackdelar relativt S^{2}E.

AMBA bygger inte på angr av tre anledningar. För det första har S^{2}E bättre prestanda än angr [35]. För det andra stödjer inte heller angr fullsystemsemulation. För det tredje finns redan interaktiv visualiserad symbolisk fuzzing byggd ovanpå angr i form av SymNav, vilket innebär att AMBA genom att bygga på S^{2}E kan särskilja sig mer från befintliga verktyg.

5 Implementation

Avsnitt 4.2 beskriver hur AMBA installeras och körs. Detta avsnitt diskuterar utvalda delar av AMBAs implementationsdetaljer.

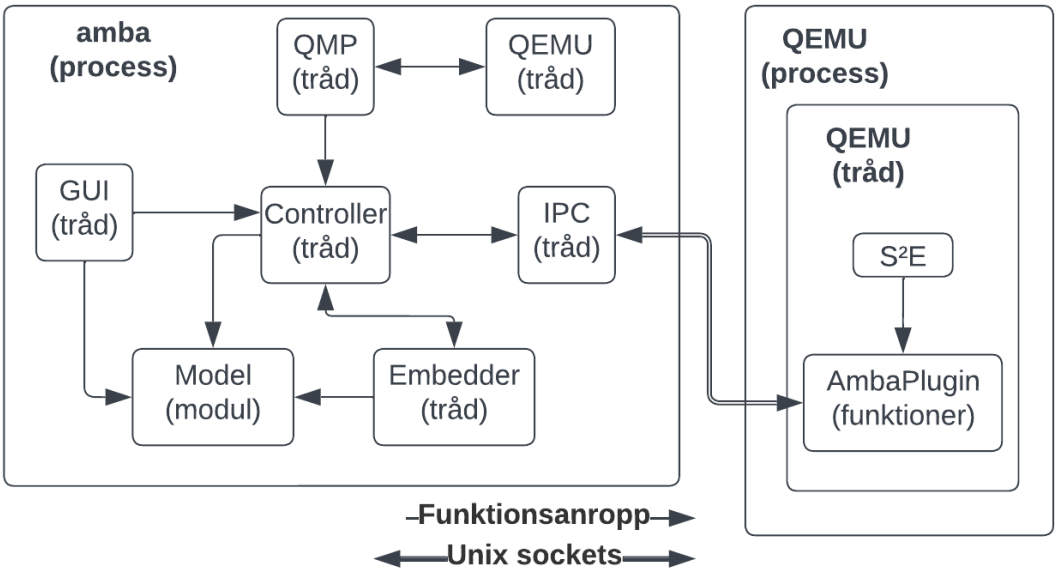
AMBA ska betraktas som ett system bestående av en huvudprocess och en bakgrundsprocess. Huvudprocessen är ett grafiskt Rust-program bestående av flera beständiga trådar som kommunicerar med varandra genom delat minne och meddelanden. Bakgrundsprocessen är en S^2E-instans startad av huvudprocessen med S^2E-pluginet AmbaPlugin som möjliggör kommunikation mellan huvudprocessen och S^2E självt. IPC mellan huvudprocessen och bakgrundsprocessen sker över två Unix-socketar. Figur 5.1 visar en översikt av AMBAs arkitektur.



Figur 5.1: AMBA: skapande av processer och trådar

5.1 Huvudprocess

Huvudprocessen kör 6 beständiga trådar. Huvudtråden eller GUI-tråden, som äger och hanterar det grafiska gränssnittet. Kontrolltråden (controller), som är mellanhand i all meddelandebaserad kommunikation mellan trådar och även är avsändare för IPC till AmbaPlugin. Utplaceringstråden (embedder), som tar emot instruktioner från kontrolltråden, kör den beräkningsmässigt tunga nodutplaceringen på de fem graferna och skriver denna till minne tillgängligt från GUI-tråden. IPC-tråden,



Figur 5.2: AMBA: kommunikation mellan processer och trådar

som tar emot meddelanden över IPC från AmbaPlugin. QMP-tråden, som tar emot och skickar meddelanden till QEMU. QMP står för QEMU Machine Protocol och möjliggör programmatisk kontroll av QEMU-instansen direkt. QMP-tråden har en väldigt begränsad funktion och är inte nödvändig för AMBAs nuvarande funktionalitet. QEMU-tråden, som startar QEMU/S^{2}E-instansen och inväntar dess avslutande. Detta inväntande är bland annat nödvändigt för att hantera en krasch i S^{2}E eller AmbaPlugin på ett bra sätt.

5.1.1 Uppstart

Som beskrivet i avsnitt 4.2 startas AMBA av användaren med en angiven receptfil. Huvudprocessen tolkar receptfilen och genererar den konfiguration som S^{2}E behöver för att köra den analyserade binären så som receptfilen specifierat. Huvudprocessen startar alla beständiga trådar och QEMU-tråden startar QEMU. QEMU startas från en minnesavbild av en virtuell maskin anpassad till S^{2}E, och en process kallad bootstrap körs som förbereder den virtuella maskinen inför den symboliska exekveringen. Denna förberedelse innebär till exempel att `/tmp/` som innehåller de symboliska filerna placeras i ett RAM-uppbackat tmpfs och att de symboliska filerna görs symboliska. Binären som ska analyseras skickas in i QEMU, startas och *hooks* till AmbaPlugin börjar sampla data som ska skickas till huvudprocessen. Datan skickas till huvudprocessen som börjar visa grafen när den byggs och samtidigt kontinuerligt utplaceras.

5.1.2 GUI

Huvudprocessen använder immediate-mode-gui-biblioteket egui [36] för att rita sitt gränssnitt. Detta innebär att hela fönstret ritas om varje bildruta, mer likt typisk spelgrafik än klassiska grafiska gränssnitt. egui är enkelt att utveckla för vilket var kritiskt för att utveckla prototypen AMBA under detta kandidatarbete. Grafvykomponenten är byggd från grundläggande egui-komponenter vilket gjorde bibliotekets utvecklarergonomi viktig.

5.2 S^{2}E

Den symboliska exekveringsmotorn S^{2}E beskrivs mer översiktligt i avsnitt 3.2. Detta avsnitt diskuterar dess användning i AMBA och den funktionalitet som är relevant för detta.

S^{2}E tillhandahåller en hel virtuell maskin med stöd för symbolisk exekvering genom att bygga ovanpå på QEMU. S^{2}E är ett dynamisk bibliotek som `LD_PRELOAD`:as i en modifierad QEMU 3.0.0 och lägger till stöd för symboliska värden i register och RAM.

S^{2}E kan fristående agera som automatisk symbolisk fuzzer och producera konkret indata motsvarande olika symboliska lövtillstånd. Men S^{2}E är komplext, har mycket som kan konfigureras och dess fulla kraft är tillgänglig endast för S^{2}E-plugin [37].

Plugin har möjlighet att följa S^{2}E:s exekvering genom återanropsfunktioner och undersöka egenskaper för att spåra och forma analyser. Plugin kan även påverka den symboliska exekveringen, till exempel genom att i realtid ange vilka tillstånd som ska exekveras och vilka som ska pausas.

Konceptuellt består ett S2E plugin av ett tillstånd och en initieringsfuktion. Tillståndet ärvs från ett kärn-plugin som innehåller information om bl.a. programtillstånd, symbolisk exekveringstillstånd och information om andra plugin. Tillståndet kan utökas för att samla information, kontrollera egenskaper och utföra binäranalys. Initieringsfunktionen består av tillståndsinitiering och registrering av återanropsfunktioner till olika *hooks* som tillgängligörs av antingen S2E:s kärnplugin eller andra plugin. Dessa återanropsfunktioner anropas under exekvering när motsvarande villkor för en *hook* uppfylls.

AmbaPlugin är ett S2E plugin som har utvecklats för att samla information under

exekvering av en binär ämnat för att visualisera exekveringen i AMBA. AmbaPlugin kan även bestämma ett subträd av tillstånd vars exekvering ska prioriteras före andra.

5.2.1 AmbaPlugin

AmbaPlugin registrerar återanropsfunktioner för ett antal *hooks*, inklusive bland annat `onStateFork`, `onTranslateBlockComplete` och `onBlockStart`. När dessa *hooks* anropas samlas information om S2E:s tillståndsid, generation för ett block i fallet av självmodifierande kod, virtuell adress, maskinkod etc.

Utifrån detta framställs kantinformation för kontrollflödesgrafen och tillståndsgrafen som kontinuerligt skickas över IPC till huvudprocessen. Denna iterativa informationsuppdateringen är nödvändigt för att uppdatera användargränssnittet i realtid, alltså flera gånger per sekund. Få verkliga binärer har tillståndsgrafer som kan konstrueras fullständigt.

5.2.2 Paketering i pakethanteraren Nix

S^2E är ett komplicerat system som innehåller bland annat QEMU, Linux, KLEE och LLVM. Dess bygge var trasigt när vi påbörjade detta kandidatarbete (t.ex. felaktiga git-remote-url:er för QEMU-projektet), flera andra småsaker hindrade byggande och utöver allt detta rekommenderades en Docker-miljö för att hantera bygget med dess alla beroende komponenter. För att förbättra bygget, uppnå bättre reproducerbarhet samt bättre samspel med en grafisk applikation paketeras S^2E i Nix.

Nix-paketet S^2E har framställts för utveckling av AMBA och är inte lämpligt för upstreaming till S^2E i nuvarande form men detta kan vara möjligt efter utökningar och modifieringar.

5.3 Grafbehandling

Huvudprocessen tar emot en ström av kanter representerade som ordnade par av nodmetadata, där mottagandet av en kant innebär att den kanten precis bevandrats. Olika symboliska tillstånd har olika noder, vilket innebär att denna kantströmmen innehåller all information som huvudprocessen behöver.

Nya noder placeras i en hashtabell och får ett sekventiellt allokerat id. Detta id används sedan för att identifiera noden i olika representationer av grafen. Grafen

representeras med en grannlista för linjegrafskompression och nedbrytning i starkt ansluta komponenter och med en kantlista för nodutplacering genom kraftsimulation. I båda dessa sammanhang är det praktiskt att noder representeras som registerstora tal.

5.3.1 Linjegrafskompression

AMBA presenterar både en komprimerad och en okomprimerad graf för användaren. Grafen komprimeras genom kantkontraktion av alla kanter anslutna till en nod av ingrad exakt ett och utgrad exakt ett. Detta är ekvivalent med att se alla par av basic blocks som alltid förekommer efter varandra som ett enda basic block. Detta ger komprimerade basic blocks som kan spänna flera funktioner och dynamiskt länkade bibliotek.

Grafkompressionen sker online, alltså genom en datastruktur som samtidigt tillåter tilläggning av kanter i grafen och kan svara på frågor om vilka kanter som kommer in samt går ut från en (komprimerad) nod. Detta implementeras genom att lagra både en komprimerad och okomprimerad graf. En pekare till den okomprimerade grafen delas ut till läsare. När en kant ska läggas till kontrolleras först om denna redan finns i den okomprimerade grafen. Om inte återställs de kantkontraktioner som skapat de komprimerade noder som innehåller kantens ändnoder. Därefter läggs kanten till och grafen komprimeras lokalt. Tidskomplexiteten för kanttilläggningar är då lika med summan av kardinaliteten på de komprimerade noder som innehåller kantens slutnoder. Detta är kvadratiskt i värstafallet en enda linjegraf men fungerar i praktiken då kontrollflödesgrafen inte innehåller långa linjegrafer. Författarna är medvetna om att en fullt linjär algoritm existerar.

5.3.2 Starkt anslutna komponenter

AMBA kan färglägga noder i de komprimerade och ickekomprimerade kontrollflödesgraferna efter deras så kallade starkt anslutna komponent (jrf. eng. strongly connected component). En riktad grafs starkt anslutna komponenter är ekvivalensklasserna under relationen att ett par av noder är både nedströms och uppströms ifrån varandra. Den starkt anslutna komponenten som en nod V tillhör är alltså mängden av de noder som både kan nå V och som kan nås av V.

AMBA bryter ned grafer i deras starkt anslutna komponenter med Tarjans algoritm [38]. Tarjans algoritm bygger på en rekursiv DFS, och AMBA hanterar dess linjära stackdjup genom att köra algoritmen på en tråd med en tillräckligt stor stack.

5.3.3 Nodutplacering genom kraftsimulation

Nodutplaceringen styrs av en attraktionskraft mellan kopplade noder som är proportionelig mot $D^{1.2}$ på avståndet D, en repulsionskraft mellan alla par av noder som är proportionelig mot $\frac{D}{1+D^3}$ och en gravitationskraft som påverkar alla noder utom rotnoden i negativ y-led.

Hastighet och position integreras med Euler framåt, men magnituden på hastigheten ändras dessutom direkt: den ökas exponentiellt när accelerationen och hastigheten är likriktade och minskas exponentiellt annars. Detta ger snabbare konvergens. Positionen kan även uppdateras med uniformt brus om användaren har ställt in detta som nollskilt.

Slutligen gäller tvång som placerar rotnoden i origo och masscentrum på axeln vertikalt under. Därmed går exekvering framförallt uppifrån ned.

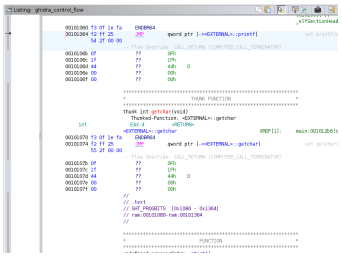
Den beräkningstunga delen av nodutplaceringen är beräkningen av repulsionskraften. Naiv summation mellan varje par av noder tar $O(n^2)$ tid per tidssteg eller $O(n^3)$ tid totalt under antagande att konvergens nås efter linjärt många steg. Därför implementerar AMBA även Barnes-Hut kraftberäkningsalgoritm på ett R-träd. Ett R-träd är ett balanserat binärt träd av punktmängder där varje lövnod är en enskild punkt och varje grennod lagrar begränsningslådan för punkterna den innehåller. AMBA bygger trädet genom att rekursivt hugga den axel som är geometriskt längst och använder en parameter för att bestämma vid kvot mellan lådstorlek och kraftavstånd där lådan anses tillräckligt liten för att dess noder kan approximeras ligga i lådans masscentrum. Barnes-Hut har en tidskomplexitet på $O(n \log n)$ per tidssteg eller för oss $O(n^2 \log n)$ totalt vilket möjliggör utplacering av större grafer.

6 Evaluering

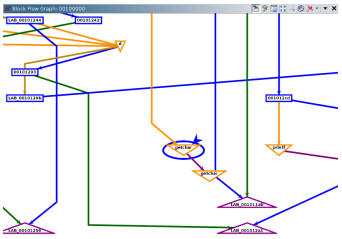
För att evaluera verktyget används befintliga verktyg som jämförelse i kombination med att deras fördelar och nackdelar vägs mot AMBA. Dessutom presenteras en diskussion kring utvecklingsprocessens gång och vidareutvecklingspotential.

6.1 Jämförelse mellan AMBA och Ghidra

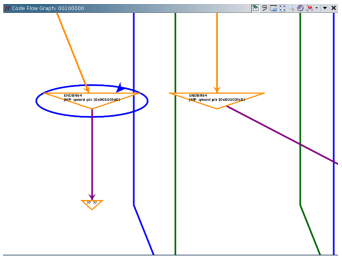
Ghidra är ett avancerat verktyg som gör statisk analys bortom syftet med AMBA, men AMBA har ett par likheter. Ghidra kan representera en kontrollflödesgraf för en given binär på två olika sätt: *Flow Graph* och *Code Flow Graph* [8].



(a) Ghidras code listing.



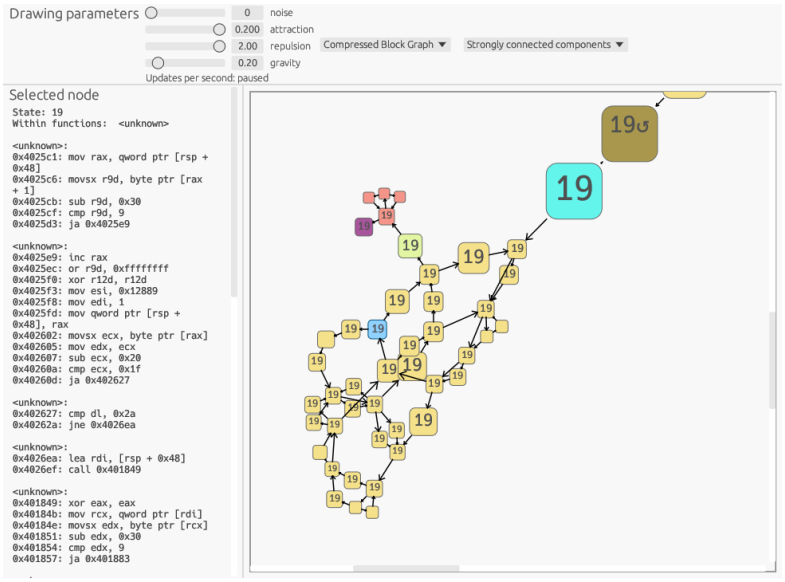
(b) Ghidras block-flow-graph.



(c) Ghidras code-flow-graph.

Figur 6.1: Tre primära vyer i verktyget Ghidra för att representera ett programs kontrollflödesgraf.

Ghidra kompletterar dessa vyer med en *code listing*, en primär vy där binärens disassemblerade kod listas. Användaren kan fritt välja att bilda en graf från ett markerat *code block*, vilket är Ghidras utökade specifikation på ett *basic block*, från Ghidras code listing [8]. AMBA har liknande funktionalitet, se figur 4.1, och kan visualisera binärens fullständiga disassemblerade kod för en given nod, om debugdata finns tillgänglig, men saknar större kontext likt Ghidras code listing. En fördel mot Ghidras representation är AMBAs fem olika sätt att representera en binärs kontrollflöde på: *Raw Basic Block Graph*, *Compressed Block Graph*, *State Graph*, *Merged Block Graph* och *Compressed Merged Block Graph*. Genom att använda dessa fem olika vyer kan en användare bilda en större förståelse om en



Figur 6.2: En del av AMBAs komprimerade basic-block-graf, färglagd efter starkt anslutna komponenter för testprogrammet control-flow

given binär, dels genom att exempelvis visualisera alla symboliska tillstånd som S^{2}E skapar i kombination med färgade noder som indikerar på starkt kopplade komponenter (jfr. eng. *strongly connected components*).

Kännedom om starkt kopplade komponenter i en kontrollflödesgraf är i synnerhet intressant vid analys av binärer utan källkod, eftersom dessa delgrafer speglar en starkare koppling mellan noderna och indikerar på en sektion i binären som troligtvis har större relevans än en sektion som saknar eller har betydligt färre starkt kopplade komponenter. Ett typiskt mönster man kan se med starkt kopplade komponenter är icke-nestlade loopar.

En användare skulle exempelvis kunna använda AMBAs nodfärgläggning för att bilda en större uppfattning om en viss del i binären och sedan komplettera med nodernas minnesaddresser för att visualisera samma sektion i ett mer avancerat verktyg som Ghidra. Ett typiskt användningsfall skulle kunna vara att i Ghidra söka efter strängar som ger mer information i den givna kodsektionen från AMBA eller använda Ghidras analysmetoder för att dekompilera binärens instruktioner till ett mer läsbart format i form av pseudokod.

6.2 Jämförelse mellan S^{2}E och angr som exekveringsmotor

S^{2}E har bättre prestanda än angr, vilket kan avläsas från figurerna i [35, Figur 1-5]. Eftersom AMBA använder S^{2}E som symbolisk exekveringsmotor har AMBA därmed en teoretisk fördel i prestanda gentemot motsvarande mjukvara byggd ovanpå angr. AMBA har i nuläget inte nödvändigtvis uppnått denna bättre prestanda, men exekveringsmotorns prestandafördel innebär att AMBA teoretiskt borde kunna uppnå en högre prestanda efter medveten optimering.

angr är mer väldokumenterat och populärt än S^{2}E. Dessutom är angr redan paketerad i Nix och hade varit enklare att påbörja utveckling ovanpå. Om AMBA byggdes ovanpå angr hade möjligvis mer funktionalitet kunnat implementeras under projektets gång, men då hade samtidigt inte fördelarna med S^{2}E kunnat utnyttjas.

6.3 Jämförelse mellan AMBA och SymNav

SymNav är ett mer moget verktyg än AMBA. Den största skillnaden mellan verktygen är att SymNav presenterar informationen från den symboliska fuzzingen på ett mer användarvänligt vis och att SymNav låter användaren filtrera bort de exekveringsvägar användaren inte vill fokusera på.

Att AMBA saknar tillståndsfiltrering gör verktyget opraktiskt för icke-triviala program, då visualiseringen blir överväldigande för användaren och AMBAs tillståndsfiltrering otymplig. AMBA kan därför inte konkurrera med SymNav i praktisk användbarhet, utan endast i det akademiska intresset i dess särskiljande funktionalitet samt potentiellt i den vidareutveckling som möjliggörs av AMBAs arktitektur.

En stor skillnad mellan AMBA och SymNav är att medan AMBA uppdaterar det grafiska gränssnittet i realtid, eller mer precist en gång per sekund, drivs SymNav genom att användaren startar körningar som varar så länge som specifierat av användaren. I ett typiskt arbetsflöde i SymNav undersöker användaren programmet genom det grafiska gränssnittet, startar en körning på en minut och väntar tills den är klar och återgår först därefter till att undersöka programmet genom det nu uppdaterade grafiska gränssnittet. En realtidsarkitektur möjliggör mer interaktivitet i hur användaren påverkar den symboliska fuzzingen.

En annan stor skillnad är vilken symbolisk exekveringsmotor som används. SymNav använder angr och AMBA använder S^{2}E. Den största konsekvensen av denna skill-

nad är att AMBA har potentiellt bättre prestanda i den symboliska exekveringen, vilket diskuterades i det förra avsnittet. Denna fördel kan, om AMBA utökas med tillståndsfiltrering, vara en faktor som gör AMBA mer passande än SymNav i ett sökningstungt användningsområde.

6.4 Arbetsprocess

Eftersom S^{2}E är ett komplext system med många komplicerade biblioteksberoenden beslutades det tidigt i arbetsprocessen att göra AMBA mer tillgängligt till slutanvändaren genom att paketera alla bibliotek som krävs för att köra S^{2}E. Detta var en komplicerad och tidskrävande uppgift eftersom mycket av S^{2}Es byggsystem var tvunget att återimplementeras eller kringgås när tvetydiga och icke-triviala problem uppstod.

Ett sådant problem var byggningen av diskavbilder till de virtuella maskiner som S^{2}E körs inuti QEMU. Detta problem löstes aldrig fullständigt utan fick arbetas runt genom att ladda ned färdiga diskavbilder byggda och uppladdade av S^{2}E-projektet i Google Drive.

Ett annat problem, detta orsakat av utdaterade bygginstruktioner i S^{2}E, var att QEMU-projektets repon pekades på genom utdaterade, nu trasiga länkar under `git.qemu-project.org` istället för uppdaterade fungerande länkar under `gitlab.com/qemu-project`.

Ett tredje problem var att biblioteket `libs2e` byggdes med ett implicit beroende på biblioteket `libgomp` som distribueras tillsammans med kompilatorn `gcc`. Specifikt hade `libs2e`-komponenten KLEE ett beroende på vissa symboler i statisk länkning som inte var tillgängliga från `libgomp` från Nix-paketerade `gcc`. Detta problem arbetades runt genom att dynamiskt länka en Ubuntu-byggd `libgomp` till `libs2e`.

Vi försökte även minimera mängden C++-kod som behövde skrivas genom att använda Autocxx för att generera kopplingar mellan C++ och Rust. Tanken var att vi med hjälp av Autocxx endast skulle behöva ett fåtal rader C++ och att merparten av källkoden skulle kunna skrivas i Rust. Tyvärr stötte vi på en bug i Autocxx där den genererade syntaktiskt inkorrekt kod. Detta, i kombination med att Autocxx saknade viktiga funktioner, ledde oss till att efter några veckor överge tanken att generera kopplingar och istället skriva och underhålla även C++-kod.

Dessa problem, tillsammans med väldigt många fler av liknande natur, ledde till mycket arbete som försköt planeringen. Således fanns det mindre tid till att ut-

veckla all funktionalitet som var tänkt i AMBA.

6.5 Vidareutveckling

En särskilt eftertraktad funktionalitet är tillståndssammanslagning. Syftet med tillståndssammanslagning är att minska antalet exekveringsvägar genom att förena symboliska tillstånd som är nästan ekvivalenta. Detta kan till exempel implementeras genom att ge uppgiften till användaren att interaktivt välja vilka tillstånd som ska sammanslås. Således ökar prestandan för den symboliska exekveringen som gör det möjligt att skicka fler förfrågningar (jfr. eng. *queries*) till SMT-lösaren.

Övervakning av systemanrop ger användaren en större insikt i vad som sker i en specifik del av grafen, exempelvis om det sker inmatning eller utmatning, om en process signaleras att stängas av, om binären kör ett `exec`-systemanrop för att köra ett externt program, etc. Att det är intressant att känna till programmets agerande i dess miljö gör det intressant att övervaka systemanrop. Detta skulle kunna implementeras i AMBA genom att lyssna efter exekvering av `sysmtemanrop`-instruktioner och när dessa exekveras läsa de register som definerar vilket systemanrop som anropas och med vilka parametrar det anropas.

Under AMBAs utvecklingsprocess har väldigt begränsad testning på icke-triviala program genomförts. Detta innebär att AMBA är ytterst svåranvänt när programmets kontrollflödesgraf är stor, både sett till att det grafiska gränssnittet saktar ned men speciellt sett till att användaren inte har några verktyg att filtrera bland informationen som presenteras. AMBA kan därmed vidareutvecklas genom att implementera filter liknande SymNavs filtreringsstöd eller annan funktionalitet lämpad för utforskning av icke-triviala program.

7 Slutsats

I avsnitt 1.4 beskrivs projektets mål om att utveckla ett binäranalysverktyg för att interaktivt visualisera symbolisk fuzzing. Visualiseringen skulle presenteras i ett grafiskt gränssnitt där användaren kan interagera med den symboliska fuzzingen av det analyserade programmet. Detta mål har uppnåtts i och med att AMBA implementerats med nuvarande funktionalitet.

Projektet motiverades av hypotesen att visuella representationer av exekveringsvägar gör det enklare för binäranalytikern att förstå binären, jämfört med ett skriptdrivet användargränssnitt som är standard vid direkt användning av S^2E, angr, SymQEMU och liknande. Vi anser att AMBA idag stödjer denna hypotesen, men under väldigt begränsade förhållanden. För att vetenskapligt undersöka hypotesen är det rimligt att först utöka AMBA eller ett liknande program såsom SymNav med funktionalitet som gör det praktiskt användbart, och/eller därefter genomföra en empirisk användarvänlighetsstudie. Att genomföra en sådan var inte ett mål och har inte heller genomförts i detta kandidatarbete.

Som nämnt i avsnitt 6.3 anser vi inte att AMBA är praktiskt användbart. Vi anser dock att AMBA, efter vidareutvecklingen beskriven i avsnitt 6.5, kan fylla en nisch i binäranalys som inte fylls av befintliga etablerade kategorier av verktyg och som inte heller fylls av SymNav i nuvarande form.

Som diskuterat i avsnitt 6.4 var ett av våra mål att paketera AMBA tillsammans med dess komplexa beroenden. Att bygga AMBAs beroenden visade sig vara mycket svårt då S^2Es bygginstruktioner är ofullständiga och delvis utdaterade. Att mycket tid behövde dedikeras till att korrigera byggsystemet innebar att mindre tid kunde allokeras till utvecklandet av slutprodukten.

Våra förhoppningar är att AMBA antingen kommer att vidareutvecklas, eller inspirera till ett nytt verktyg, med funktionerna som diskuterats som vidareutveckling. För närvarande anser vi inte att AMBA är praktiskt användbart, men anser att det finns en stor potential i vidareutveckling av AMBA.

Referenser

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan och S. Li, "Measuring Program Comprehension: A Large-Scale Field Study with Professionals," *IEEE Transactions on Software Engineering*, årg. 44, nr 10, s. 951–976, 2018. DOI: 10.1109/TSE.2017.2734091.

[2] D. Andriesse, *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly*. No Starch Press, dec. 2018, ISBN: 9781593279127.

[3] F. Hermans, *The Programmer's Brain, What every programmer needs to know about cognition*, med förord av J. Skeet. Manning Publications Co., aug. 2021, ISBN: 9781617298677.

[4] M. Boehme, C. Cadar och A. Roychoudhury, "Fuzzing: Challenges and Reflections," *IEEE Software*, årg. 38, nr 3, 2021. DOI: 10.1109/MS.2020.3016773.

[5] A. Zeller, R. Gopinath, M. Böhme, G. Fraser och C. Holler, "Symbolic Fuzzing," i *The Fuzzing Book*, Retrieved 2023-01-07 15:17:16+01:00, CISPA Helmholtz Center for Information Security, 2023. URL: https://www.fuzzingbook.org/html/SymbolicFuzzer.html.

[6] M. G. Rekoff, "On reverse engineering," *IEEE Transactions on Systems, Man, and Cybernetics*, årg. SMC-15, nr 2, 1985. DOI: 10.1109/TSMC.1985.6313354.

[7] J. Xu m. fl., "Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs," i *The 32nd USENIX Security Symposium (Security'23)*, 2023.

[8] "Gidra - A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission." (2023), URL: https://ghidra-sre.org/ (hämtad 2023-03-23).

[9] H. Liang, X. Pei, X. Jia, W. Shen och J. Zhang, "Fuzzing: State of the Art," *IEEE Transactions on Reliability*, årg. 67, nr 3, s. 1199–1218, 2018. DOI: 10.1109/TR.2018.2834476.

[10] P. Godefroid, M. Y. Levin och D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft.," *Queue*, årg. 10, nr 1, s. 20–27, jan. 2012, ISSN: 1542-7730. DOI: 10.1145/2090147.2094081. URL: https://doi.org/10.1145/2090147.2094081.

[11] N. Nethercote, "Dynamic binary analysis and instrumentation," University of Cambridge, Computer Laboratory, tekn. rapport UCAM-CL-TR-606, nov. 2004. DOI: 10.48456/tr-606. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf.

[12] M. Miller. "Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape," MSRC. (7 febr. 2019), URL: https://www.youtube.com/watch?v=PjbGojjnBZQ (hämtad 2023-03-01).

[13] D. Wagner, N. Weaver, P. Kao, F. Shakir, A. Law och N. Ngai. "Computer Security: CS 161 Course Textbook." (2023), URL: https://textbook.cs161.org/ (hämtad 2023-05-08).

[14] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu och I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, årg. 51, nr 3, maj 2018, ISSN: 0360-0300. DOI: 10.1145/3182657.

[15] S. Xu och Y. Wang, "BofAEG: Automated Stack Buffer Overflow Vulnerability Detection and Exploit Generation Based on Symbolic Execution and Dynamic Analysis," *Security and Communication Networks*, årg. 2022, 2022. DOI: 10.1155/2022/1251987.

[16] I. A. Vakhrushev, V. V. Kaushan, V. A. Padaryan och A. N. Fedotov, "Search method for format string vulnerabilities," *Proceedings of ISP RAS*, årg. 27, 4 2015. DOI: 10.15514/ISPRAS-2015-27(4)-2.

[17] S. Poeplau och A. Francillon, "SymQEMU: Compilation-based symbolic execution for binaries," *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[18] C. Cadar, D. Dunbar och D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," i *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, s. 209–224.

[19] C. Cadar, D. Dunbar och D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," i *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008, s. 209–224. URL: https://www.usenix.org/conference/osdi08/technical-sessions/presentation/cadar.

[20] P. Boonstoppel, C. Cadar och D. Engler, "RWset: Attacking Path Explosion in Constraint-Based Test Generation," i *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan och J. Rehof, utg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, s. 351–366, ISBN: 978-3-540-78800-3.

[21] J. Li, B. Zhao och C. Zhang, "Fuzzing: a survey," *Cybersecurity*, årg. 1, dec. 2018. DOI: 10.1186/s42400-018-0002-y.

[22] M. Boehme, C. Cadar och A. Roychoudhury, "Fuzzing: Challenges and Reflections," *IEEE Software*, årg. 38, nr 03, s. 79–86, maj 2021, ISSN: 1937-4194. DOI: 10.1109/MS.2020.3016773.

[23] A. Fioraldi, D. Maier, H. Eißfeldt och M. Heuse, "AFL++ : Combining Incremental Steps of Fuzzing Research," i *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[24] P. Wang, X. Zhou, K. Lu, T. Yue och Y. Liu, "Sok: The progress, challenges, and perspectives of directed greybox fuzzing," *Challenges, and Perspectives of Directed Greybox Fuzzing*, 2020.

[25] R. Messier och M. Berninger, "Getting Started with Ghidra by Ric Messier, Matthew Berninger," *O'Reilly Media, Inc*, 2023. URL: https://www.oreilly.com/library/view/getting-started-with/9781098115265/ch01.html (hämtad 2023-03-23).

[26] M. Ivaldi. "Automating binary vulnerability discovery with Ghidra and Semgrep." (juli 2022), URL: https://security.humanativaspa.it/automating-binary-vulnerability-discovery-with-ghidra-and-semgrep/ (hämtad 2023-03-23).

[27] Hex Rays. URL: https://hex-rays.com/ (hämtad 2023-04-27).

[28] Vector 35, *Binary Ninja*. URL: https://binary.ninja/ (hämtad 2023-04-27).

[29] V. Chipounov, V. Kuznetsov och G. Candea, "S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems," i *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, Newport Beach, California, USA: Association for Computing Machinery, 2011, s. 265–278, ISBN: 9781450302661. DOI: 10.1145/1950365.1950396. URL: https://doi.org/10.1145/1950365.1950396.

[30] Y. Shoshitaishvili m. fl. "angr Documentation." (2022), URL: https://docs.angr.io/ (hämtad 2023-03-23).

[31] "Unicorn CPU Emulator." (), URL: https://www.unicorn-engine.org/ (hämtad 2022-11-13).

[32] *angr.engines.unicorn - angr Documentation*. URL: https://docs.angr.io/en/latest/_modules/angr/engines/unicorn.html (hämtad 2023-05-15).

[33] M. Hentschel, R. Bubel och R. Hähnle, "The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more," *International Journal on Software Tools for Technology Transfer*, årg. 21, s. 485–513, 2019.

[34] M. Angelini m. fl., "SymNav: Visually assisting symbolic execution," i *2019 IEEE Symposium on Visualization for Cyber Security (VizSec)*, IEEE, 2019, s. 1–11.

[35] S. Poeplau och A. Francillon, "Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and Its Generation," i *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19, San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019, s. 163–176, ISBN: 9781450376280. DOI: 10.1145/3359789.3359796. URL: https://doi.org/10.1145/3359789.3359796.

[36] E. Ernerfeldt. "egui: an easy-to-use GUI in pure Rust." (18 april 2023), URL: https://github.com/emilk/egui/blob/3d6a15f442929ae14b942e2d5248f236e7ef9b2f/README.md (hämtad 2023-04-27).

[37] V. Chipounov, V. Kuznetsov och G. Candea, "The S2E Platform: Design, Implementation, and Applications," *ACM Trans. Comput. Syst.*, årg. 30, nr 1, febr. 2012, ISSN: 0734-2071. DOI: 10.1145/2110356.2110358. URL: https://doi.org/10.1145/2110356.2110358.

[38] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, årg. 1, nr 2, s. 146–160, 1972.