ԵՐԵՎԱՆԻ ՊԵՏԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ

ՏԵՂԵԿԱՏՎԱԿԱՆ ՏԵԽՆՈԼՈԳԻԱՆԵՐԻ ԿՐԹԱԿԱՆ ԵՎ ՀԵՏԱԶՈՏԱԿԱՆ ԿԵՆՏՐՈՆ

ՏԵՂԵԿԱՏՎԱԿԱՆ ՀԱՄԱԿԱՐԳԵՐԻ ԱՄԲԻՈՆ

ՏԵՂԵԿԱՏՎԱԿԱՆ ՀԱՄԱԿԱՐԳԵՐԻ ԿԱՌԱՎԱՐՈՒՄ ԿՐԹԱԿԱՆ ԾՐԱԳԻՐ

Մկրտչյան Ալբերտ Կարենի

ՄԱԳԻՍՏՐՈՍԱԿԱՆ ԹԵԶ

Ծրագրային կոդի հատկությունների հարցումների համակարգ՝ հիմնված ղեկավարման կախվածությունների գրաֆի վրա

«Տեղեկատվական համակարգերի կառավարում» մասնագիտության ինֆորմատիկայի մագիստրոսի որակավորման աստիճանի հայցման համար

ԵՐԵՎԱՆ 2024

Ուսանող՝ _____ *Մկրտչյան Ալբերտ*
Ստորագրություն

Գիտական ղեկավար՝ _____ *ֆ.մ.գ.թ., Ասլանյան Հայկ*
Ստորագրություն

Գրախոս՝ _____ *պ.գ.թ., Տիգրան Թուխյան*
Ստորագրություն

«Թույլատրել պաշտպանության»

Ամբիոնի վարիչ՝ _____ *ՀՀ ԳԱԱ ակադեմիկոս, ֆ.մ.գ.դ.,*
Ստորագրություն
պրոֆեսոր, Սամվել Շուքուրյան

«_____»_____20__թ

Համառոտագիր

Ծրագրային կոդի հատկությունների հարցումների համակարգ հիմնված ղեկավարման կախվածությունների գրաֆի վրա

Система запроса свойств исходного кода на основе графа зависимостей управления

A system for querying source code properties based on a control dependency graph

Աշխատանքի շրջանակներում հետազոտվել են ծրագրային կոդի վերլուծություն կատարող ժամանակակից գործիքները: Գոյություն ունեցող գործիքների մեծ մասը կարողանում են հայտնաբերել միայն սահմանափակ քանակի խնդիրներ, և չեն ապահովում բավարար ֆունկցիոնալություն: Ֆունկցիոնալության ավելացման համար անհրաժեշտ են խորը գիտելիքներ ծրագրային վերլուծության բնագավառում, սակայն շատ ծրագրավորողներ չեն տիրապետում նմանատիպ վերլուծությունների: Աշխատանքում նախագծվել և իրականացվել է նոր գործիք, որը պահանջելով միայն համապատասխան ծրագրավորման լեզվի իմացություն թույլ է տալիս կատարել ծրագրային կոդի վերլուծություն:

Բովանդակություն

ՆԵՐԱԾՈՒԹՅՈՒՆ

Ծրագրային ապահովման ոլորտը շարունակաբար զարգանում և ընդլայնվում է, ինչի արդյունքում ծրագրային ապահովման ծավալներն անընդհատ աճում են: Անընդհատ աճող ծրագրային ապահովման ծավալների հետ մեկտեղ առաջանում է ծրագրային կոդի հատկությունների ավտոմատ վերլուծության գործիքների պահանջարկ: Ավտոմատացված վերլուծությունները թույլ են տալիս բացահայտել ծրագրի տարբեր կողմերը, ինչպիսիք են դասերի, ֆունկցիաների, փոփոխականների և հրահանգների կապերը, ինչպես նաև ստանալ տեղեկատվություն դրանց մասին: Նման վերլուծությունները կարևոր են ոչ միայն նոր ծրագրերի մշակման, այլև գոյություն ունեցող ծրագրերի պահպանման և զարգացման համար:

1. Արդիականություն

Աշխարհում առկա ծրագրային կոդի հատկությունների վերլուծության գործիքներն ունեն որոշ սահմանափակումներ և թերություններ: Այդ գործիքների մեծ մասը կարողանում են հայտնաբերել միայն սահմանափակ քանակի խնդիրներ և չեն ապահովում բավարար ֆունկցիոնալություն: Ֆունկցիոնալության ավելացման համար անհրաժեշտ են խորը գիտելիքներ ծրագրային վերլուծության բնագավառում, սակայն շատ ծրագրավորողներ չեն տիրապետում նմանատիպ վերլուծությունների, իսկ այդ գիտելիքները ձեռք բերելու համար անհրաժեշտ են ժամանակ և ռեսուրսներ: Այս խնդիրը լուծելու համար առաջանում է ծրագրային կոդի հատկությունների վերլուծության այնպիսի համակարգի անհրաժեշտություն, որը կարող է մատչելի լինել ծրագրավորողներին՝ առանց խորը մասնագիտական գիտելիքների պահանջի:

Այս աշխատանքի նպատակն է նախագծել և իրականացնել ծրագրային կոդի հատկությունների հարցումների համակարգ, որը թույլ կտա կատարել ծրագրային կոդի վերլուծություն՝ պահանջելով միայն համապատասխան ծրագրավորման լեզվի իմացություն:

2. Խնդրի դրվածքը

Նախագծել և իրականացնել ծրագրային կոդի հատկությունների հարցումների համակարգ, որը կվերլուծի և կտրամադրի տեղեկատվություն դասերի, ֆունկցիաների, հրահանգների, փոփոխականների և դրանց կապերի մասին:

3. Գոյություն ունեցող գործիքներ

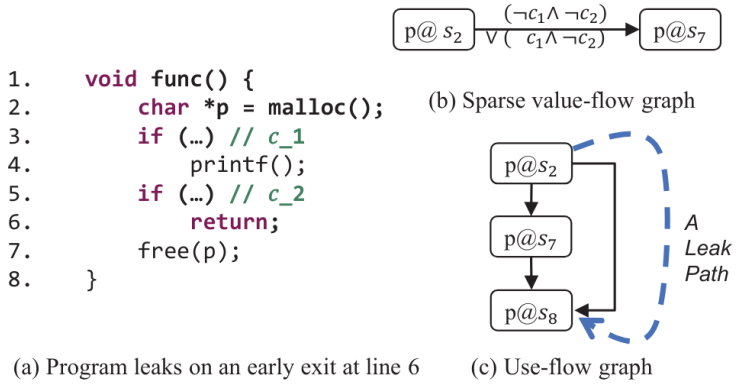
Աշխատանքի սկզբնական փուլում կատարվել է արդեն գոյություն ունեցող կոդի ստատիկ վերլուծություն կատարող գործիքների ուսումնասիրություն: Արդյունքում հետազոտվել են առաջատար գործիքների մշակված ալգորիթմների թերություններն ու առավելությունները, որոշ գործիքներ փորձարկվել են Ջուլիետ թեստային հավաքածույում՝ CWE-401 տիպի սխալներ[3] գտնելու համար:

3.1. SMOKE

SMOKE[18]-ի ալգորիթմը կազմված է երկու հիմնական փուլերից՝ բարձր ճշտության և մասշտաբայնության հասնելու համար: Առաջին փուլում այն օգտագործում է պարզ, բայց ոչ ճշգրիտ վերլուծություն՝ հիշողության արտահոսքի բոլոր հնարավոր ուղիները հայտնաբերելու համար և զտում է այն ուղիները, որոնք չեն կարող հանգեցնել արտահոսքի: Այս նպատակով նախ օգտագործվում է նոսր արժեքների կախվածության գրաֆը, ապա կառուցվում է օգտագործման կախվածության գրաֆը (UFG): UFG-ն պարունակում է վերլուծության համար բավարար ինֆորմացիա բոլոր դինամիկ հիշողության օբյեկտների մասին: UFG-ի յուրաքանչյուր կող համապատասխանեցվում է պայմանների հետ, որոնցից կախված ուղղորդվում է ծրագրի աշխատանքի ընթացքը: Նկար 1-ում (վերցված է[18]-ից) պատկերված է UFG-ի օրինակ: UFG-ն բացահայտ նկարագրում է ցուցիչի արժեքների հնարավոր ընթացքն ստեղծելով սահմաններից դուրս գտնվող գագաթ (p@s8): Դա նշանակում է, որ դինամիկ հիշողության օբյեկտը, որն օգտագործվում էր վերը նշված ցուցիչի միջոցով այլևս հասցեավորված չէ:

Երկրորդ փուլում այն օգտագործում է արդեն ստացված UFG-ի առավել ճշգրիտ վերլուծություն: Սկզբում որոնում է բոլոր ճանապարհները, որոնք չեն պարունակում դինամիկ հիշողություն օգտագործող օպերացիաներ: Այնուհետև հայտնաբերված յուրաքանչյուր ճանապարհի համար կիրառում է Z3 գործիքը[15]՝ նրանց իրագործելիությունը ստուգելու համար: Այս գործընթացն անհրաժեշտ է ճանապարհների զգայունությունն ապահովելու և կեղծ հայտնաբերված արտահոսքերը զտելու համար: Գործիքն ունի որոշ սահմանափակումներ՝

1. Վերլուծությունն արվում է դաշտերի հանդեպ ոչ զգայուն:



Նկար 1: Դինամիկ հիշողության արտահոսքի օրինակ

2. Ցուցիչների վերլուծությունը ճշգրիտ չէ:
3. Որոշ ճանապարհներ անիրագործելի են բարդ թվաբանական և միջֆունկցիոնալ տվյալների կախվածությունների պատճառով:
4. Հաշվի չի առնվում թվաբանական գործողությունները ցուցիչների հետ, free (p + y) արտահայտությունը համարժեք է համարվում free (p)-ին, ինչն ակնհայտ սխալ է:
5. Հաշվի են առվում միայն այն դեպքերը, երբ դինամիկ հիշողության առանձնացման գործողությունը բարեհաջող է անցնում:

SMOKE-ը ծրագրավորված է LLVM-ի վերին մակարդակում և միայն բինար տարբերակն է հասանելի[14]:

3.2. PCA

CA[21]-ն առաջին հերթին թարգմանում է նախնական կոդը LLVM IR-ի: Այնուհետև, օգտագործելով LLVM-ի gold plugin-ը բոլոր IR ֆայլերը հավաքում է մեկում: Ապա այն օգտագործում է Անդերսենի ցուցիչների վերլուծության գործիքը[1] և հավաքում ցուցիչների մասին ինֆորմացիա, մասնավորապես, թե որ դինամիկ հիշողության օբյեկտի վրա է հղված այս կամ այն ցուցիչը: Այդ ինֆորմացիայի վրա հիմնվելով, կառուցվում են ֆունկցիաների կանչերի կախվածության և ղեկավարման կախվածության գրաֆները: Եւ վերջում, արդեն ունենալով համապատասխան գրաֆներն ու ինֆորմացիան, կառուցվում է միջֆունկցիոնալ տվյալների կախվածության գրաֆ (DDG):

Դինամիկ հիշողության արտահոսքի հայտնաբերման նպատակով PCA-ը

յուրաքանչյուր դինամիկ հիշողություն առանձնացնող ինստրուկցիայի (A) համար հավաքում է բոլոր նրանից հասանելի գագաթները (N) DDG-ում: Եթե N-ը առանձնացված հիշողությունն ազատող ինստրուկցիա չի ներառում, ապա պնդում է, որ տեղի ունի հիշողության արտահոսք:

Որոշ դինամիկ հիշողության օբյեկտների կարող են հետևել մեկից ավելի դրանք ազատող ինստրուկցիաներ DDG-ում: Նման դեպքերում, A-ն կհամարվի ազատված, եթե գոյություն ունի ղեկավարման կախվածության ճանապարհ A-ից դեպի այն ազատող ինստրուկցիան: Հակառակ դեպքում նույնպես պնդում է, որ տեղի ունի հիշողության արտահոսք: Գործիքն ունի հետևյալ սահմանափակումները՝

1. Վերլուծությունն արվում է հոսքի, դաշտի և համատեքստի հանդեպ ոչ զգայուն:
2. Այն օգտագործում է LLVM gold plugin-ը IR ֆայլերը մեկում հավաքելու համար, այնուհետև կառուցում է DDG, ինչը գործիքը դարձնում է ոչ մասշտաբային:

Գործիքի նախնական կոդը հասանելի է[10]:

3.3. SVF

SVF[24]-ն հիմնված է LLVM-ի վերին մակարդակում: Առաջին քայլում այն թարգմանում է նախնական կոդը LLVM IR-ի: Այնուհետև, օգտագործելով LLVM-ի gold plugin-ը բոլոր IR ֆայլերը հավաքում է մեկում: SVF-ն օգտագործում է որոշ ցուցիչների վերլուծիչներ, ցուցիչների համար համապատասխան ինֆորմացիա հավաքելու նպատակով, մասնավորապես, թե որ դինամիկ հիշողության օբյեկտի վրա է հղված այս կամ այն ցուցիչը: Վերլուծիչներցի մեկն Անդերսենի ցուցիչների վերլուծիչն է[1]: Օգտագործելով LLVM IR-ը և ցուցիչների մասին հավաքված ինֆորմացիան՝ գործիքը կառուցում է ղեկավարման կախվածության գրաֆ (CFG) և հավաքում է հիշողության ստատիկ առանձին վերագրման մասին ինֆորմացիան (SSA - Static Single Assignment): Յուրաքանչյուր VFG-ի գագաթ իրենի ներկայացնում է ծրագրի որևէ ինստրուկցիա, իսկ գագաթների միջև կողերը տեղադրվում են օգտվելով օգտագործման-հայտարարման և ցուցիչների մասին հավաքված ինֆորմացիայից:

Բացի այդ, SVF-ն ապահովում է հիշողության տարածքների տարանջատում, ինչը թույլ է տալիս օգտվողներին հիշողությունը բաժանել հավաքածուների: Սա օգտակար է մեծածավալ ծրագրերը վերլուծելու համար, եթե հաշվի է առնվում հիշողության որոշակի տարածք:

Տարբեր ստուգման գործիքներ կարող են իրականացվել VFG-ի հիման վրա, որը հասանելի է օգտվող ծրագրերի համար: Հիշողության արտահոսքի հայտնաբերումը համարվում է աղբյուր-ստացողի խնդիր (յուրաքանչյուր հիշողության հատկացում յուրաքանչյուր ուղու վրա պետք է հասնի իր ազատմանը): Ստորև նշված են գործիքի որոշ սահմանափակումներ՝

1. Այն օգտագործում է LLVM gold plugin-ը IR ֆայլերը մեկում հավաքելու համար, այնուհետև կառուցում է DDG, ինչը գործիքը դարձնում է ոչ մասշտաբային:
2. Վերլուծությունն արվում է ճանապարհների և դաշտերի հանդեպ ոչ զգայուն:

3.4. Pinpoint

Pinpoint[22]-ը փորձում է հաղթահարել արդեն գոյություն ունեցող ցուցիչների վերլուծիչների համար մասշտաբայնության խնդիրները: Այն օգտագործվում է մի շարք աղբյուր-ստացող խնդիրների լուծման համար, ինչպիսիք են դինամիկ հիշողության արտահոսքի հայտնաբերման, ազատումից հետո օգտագործված ցուցիչի հայտնաբերման, կրկնակի ազատման և այլն: Այս գործիքի հիմնական տրամաբանությունը կայանում է նրանում, որ ցուցիչների անալիզի փուլը բաժանված է երկու ենթափուլերի: Առաջինը պարզ ճշգրտությամբ լոկալ փոփոխականների տվյալային կախվածության հետազոտումն է: Երկրորդը բարդ և ռեսուրսատար միջֆունկցիոնալ տվյալների կախվածության և ճանապարհների անալիզն է: Բոլոր փոփոխականների համար վերլուծություն անելու փոխարեն այն ընտրում է միայն անհրաժեշտ փոփոխականները և կատարում է միայն այդ փոփոխականների միջֆունկցիոնալ վերլուծությունը: Ճանապարհների իրագործելիության վերլուծությունը կատարվում է ըստ պահանջի և օգտագործում է SMT վերլուծող համակարգ: Այս մոտեցումը նվազեցնում է կառավարման հոսքի ուղիների քանակը և գործարկում է SMT վերլուծող համակարգը միայն այդ ուղիների համար: Գործիքն իրականացվել է LLVM-ի վերին մակարդակում և առևտրայնացվել է[11]:

3.5. Sparrow

Sparrow[20]-ն ստատիկ վերլուծություն կատարող գործիք է, որը հայտնաբերում է դինամիկ հիշողության արտահոսքի խնդիրը C լեզվով գրված ծրագրերում: Այն հիմնված է վերացական մեկնաբանության վրա և կարգավորում է ցիկլերը, կամայական ֆունկցիայի կանչերի ցիկլերը և կեղծանունները: Ալգորիթմը հուսալի չէ: Այսպիսով, այն չի կարող երաշխավորել ծրագրում հնարավոր բոլոր հիշողության արտահոսքների հայտնաբերումը:

Sparrow-ն առանձին վերլուծում է յուրաքանչյուր ֆունկցիայի հիշողության օգտագործման վարքը և ստացված ինֆորմացիան պահպանում: Յուրաքանչյուր ֆունկցիայի նկարագիրն օգտագործվում է իր կանչերի ժամանակ: Այն վերլուծում է ֆունկցիաների կանչերի գրաֆը հակառակ տեղաբանական կարգով: Ֆունկցիայի կանչերի ցիկլերի դեպքում կանչերի ցիկլի բոլոր ֆունկցիաները միասին վերլուծվում են մեկ ֆիքսված կետի կրկնության շրջանակներում: Դինամիկ կանչի սահմանների դեպքում (օրինակ ՝ ֆունկցիայի ցուցիչների պատճառով) ֆունկցիան կանչողը սպասում է, մինչև կանչվող ֆունկցիայի վերլուծությունը պատրաստ լինի:

Ֆունկցիաների վերլուծության ամփոփումը կազմված է երկու հիմնական փուլերից: (1) Հիշողության էֆեկտների գնահատում և (2) գնահատման արդյունքների օգտագործում՝ հիշողության հնարավոր արտահոսքերը հայտնաբերելու համար օգտակար տեղեկատվությունից բաղկացած ամփոփում ստանալու նպատակով: Հիշողության էֆեկտի գնահատման փուլը հիմնված է վերացական մեկնաբանության վրա:

Յուրաքանչյուր ֆունկցիայի համար հիշողության էֆեկտը բաղկացած է տեղեկատվության երեք մասից՝ առանձնացված հասցեներ, ազատված հասցեներ և ֆունկցիայի ավարտին հիշողության կարգավիճակը: Գործիքը հավաքում է բոլոր այն հասցեները, որոնք ակնհայտ հասանելիություն ունեն դրսից (օրինակ գլոբալ փոփոխականներ, ցուցիչ արգումենտներ, վերադարձվող արժեք և այլն): Այնուհետև գործիքը գտնում է, թե դրանցից որ հասցեներն են առանձացված հիշողության հասցեներ, որոնք ազատված կամ վերանվանված: Արդյունքները պահպանվում են տվյալ ֆունկցիայի ամփոփման մեջ:

Յուրաքանչյուր ֆունկցիայի վերլուծության ամփոփում իր մեջ ներառում է տեղեկություն դրսից հասանելի բոլոր հասցեների մասին: Գործիքը տեղեկացնում է

դինամիկ հիշողության արտահոսքի սխալի հայտնաբերման մասին, երբ ֆունկցիայի վերլուծության արդյունքում գտնվում է դինամիկ հիշողության օբյեկտ, որի հասցեավորումը կորցնում է տվյալ ֆունկցիան և դրսից ոչ մի տեղ չի պահպանվում: Ստորև ներկայացված են գործիքի որոշ թերություններ՝

1. Բոլոր գլոբալ փոփոխականների մասին ինֆորմացիան պահվում է գրաֆի մեկ հանգույցում և այդպիսով գործիքը չի բացահայտում միևնույն գլոբալ փոփոխականի մեջ տարբեր հիշողության օբյեկտների հասցեներ պահպանելու խնդիրը:
2. Վերլուծությունն անզգայուն է ճանապարհների նկատմամբ: Ֆունկցիայի արգումենտների համար այն հավաքում է բոլոր ազատված հասցեները՝ առանց ճանապարհները հաշվի առնելու: Այսպիսով, գործիքը կարող է չհայտնաբերել հիշողության արտահոսքը, եթե հիշողության բլոկի արտահոսքը ազատվում է այլ ճանապարհով, քան օգտագործվածը: Նմանապես, գլոբալ փոփոխականների նշանակումները հավաքվում են անկախ առանց ճանապարհները հաշվի առնելու:
3. Գործիքը վերլուծության ընթացքում ենթադրում է, որ բոլոր ցիկլերը ծրագրի աշխատանքի ընթացքում աշխատելու են ամենաքիչը մեկ անգամ, ինչը կարող է հանգեցնել կեղծ բացասական արդյունքների:

Փաստաթղթերի համաձայն SPEC2000 հենանիշի և այլ բաց կոդով ծրագրերի վերլուծության արդյունքում գործիքի կողմից հայտնաբերել է հիշողության 332 արտահոսք, և միայն 47-ը կեղծ դրական են:

3.6. Fastcheck

Fastcheck[17]-ն իրականացնում է միջֆունկցիոնալ վերլուծության ալգորիթմ C լեզվով գրված ծրագրերում դինամիկ հիշողության արտահոսքի հայտնաբերման համար: Առաջին քայլով Fastcheck-ը շարահյուսական վերլուծություն է իրականացնում նախնական կոդի համար ներքին թարգմանիչի միջոցով (այն չի օգտագործում արտաքին այլ թարգմանիչներ) և վերլուծության հիման վրա կառուցում է ղեկավարման կախվածության գրաֆ (CFG): Երկրորդ քայլով այն կատարում է հայտարարման-օգտագործման անալիզ CFG-ի հիման վրա: Արդեն հավաքված

ինֆորմացիայի հիման վրա այն կատարում է ցուցիչների վերլուծություն (հոսքից անկախ, միավորման վրա հիմնված) և կառուցում է արժեքների կախվածության գրաֆը (VFG)։ Այս քայլում VFG-ի կողերը դեռ համապատասխանեցված չեն ճյուղավորման օպերատորների պայմանների հետ։

Ալգորիթմը հետևում է VFG-ի միջոցով հիշողության բաշխման կետերից դեպի ազատման կետեր արժեքների հոսքին։ Եթե այդպիսի ճանապարհներ չկան, ապա հաղորդում է դինամիկ հիշողության արտահոսքի մասին։ Հակառակ դեպքում, տեղաբաշխմանը վերաբերող ենթագրաֆը առանձնացվում է, և այդ ենթագրաֆի կողերը նշվում են համապատասխան ճյուղավորման պայմաններով՝ ստեղծելով պաշտպանված գրաֆիկ։ Հիշողության արտահոսքի հայտնաբերման խնդիրը համապատասխանեցվում է պաշտպանված արժեքների հոսքի գծապատկերում գրաֆի հասանելիության խնդրին։ Եթե կա թույլատրելի ուղի՝ առանց դրանում ազատման ինստրուկցիա կանչելու հրահանգի, ապա հաղորդվում է հիշողության արտահոսքի մասին։ Ճանապարհնեի ստուգումը կատարվում է SMT վերլուծիչի միջոցով։ Ստորև նշված են որոշ սահմանափակումներ՝

1. Ցուցիչների վերլուծությունը համատեքստի հանդեպ զգայուն չէ։
2. Շարահյուսական վերլուծությունն իրականացվում է ոչ լիարժեք և փորձերի ընթացքում պարզվել է, որ շատ դեպքերում նախնական կոդը չի հաջողվում վերլուծել։

Fastcheck-ը գրված է Java լեզվով և նախնական կոդը հասանելի է[6]։

3.7. Clang Static Analyzer

Clang static analyzer[5]-ը բաց նախնական կոդով գործիք է, որը հայտնաբերում է C, C++ լեզուներով գրված ծրագրեում առկա դինամիկ հիշողության արտահոսքի խնդիրները։ Այն իրականացնում է ճանապարհների հանդեպ զգայուն միջֆունկցիոնալ վերլուծություն։ Գործիքն իրականացնում է ծրագրի սիմվոլիկ կատարում և բոլոր մուտքային տվյալներին տալիս է սիմվոլիկ արժեքներ։

Clang static analyzer-ը Clang[4]-ի մի մասն է կազմում և օգտագործում է կառուցվածքներ, վերացական շարահյուսական ծառեր (AST) և ղեկավարման կախվածության գրաֆ (CFG) մասնատված գրաֆ (EG) կառուցելու նպատակով։ EG-ի գագաթները ծրագրի կարգավիճակներն են, իսկ կողերը ծրագրի կետերն ու

պայմաններն են (ProgramPoint, ProgramState): ProgramState-ը նկարագրում է ծրագրի աշխատանքի վերացական պահը: (Point1, State1) գագաթից դեպի (Point2, State2) գագաթ կողը պայման է Point1-ի և Point2-ի միջև, որը State1-ը դարձնում է State2:

Ֆունկցիաների կանչերը հնարավորինս տեղադրված են, քանի որ նրանց համատեքստերն ու ճանապարհները տեսանելի են ֆունկցիան կանչողի համատեքստում: Այսպիսով, ուսումնասիրվում են ծրագրի բոլոր հնարավոր ճանապարհները:

Խնդիրները հայտնաբերելու նպատակով, գործիքն օգտագործում է ստուգիչներ, որոնցից յուրաքանչյուրը գեներացնում է համապատասխան վերացական կարգավիճակներ ծրագրի յուրաքանչյուր կետի համար: Վերջում դիրարկելով բոլոր կարգավիճակները, ստուգիչը հայտնաբերում է գոյություն ունեցող խնդիրները: Գործիքն ունի հետևյալ թերությունները՝

1. Ցիկլերի իտերացիաների քանակը հնարավոր է փոխել ‘maxloop’ արգումենտի միջոցով: Յուրաքանչյուր ցիկլ աշխատում է maxloop անգամ և հաստատուն իտերացիաների քանակով ցիկլեր անալիզ անելու ժամանակ կարող է մեծ ինֆորմացիայի կորուստ լինել, երբ այդ հաստատունը մեծ լինի maxloop-ից:
2. Ճանապարհների արագ աճ կարող է տեղի ունենալ մեծաքանակ if-երի դեպքում: Եւ այդպիսով, էքսպոնենցիալ արագությամբ կարող է աճել կարգավիճակների քանակը:

Clang static analyzer-ն LLVM-ի մասն է կազմում և նախնական կոդը հասանելի է[8]:

3.8. Infer

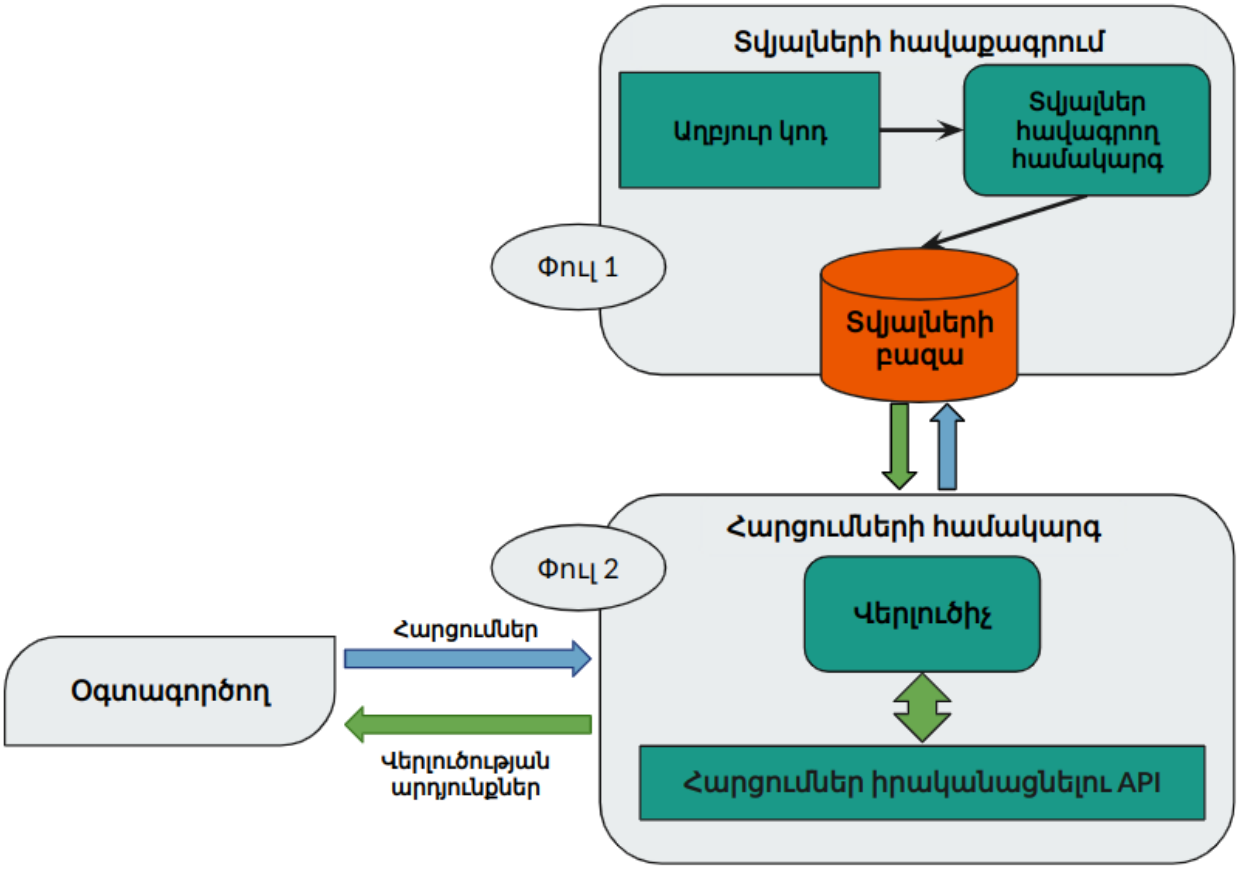
Infer[7]-ը ստատիկ վերլուծիչ է C/C++, Objective C և Java լեզուներով գրված ծրագրերի համար: Վերլուծության արդյունքում այն նախնական կոդում հայտնաբերում է գոյություն ունեցող մի շարք խնդիրներ, որոնց շարքում նաև դինամիկ հիշողության արտահոսքի խնդիրը: Գործիքը վերլուծության համար օգտագործում է SIL[13] միջանկյալ ներկայացումը և ալգորիթմը հիմնված է Hoare logic[19]-ի վրա: Առաջին քայլում այն կառուցում է ֆունկցիաների կանչերի գրաֆ (CG) և ղեկավարման կախվածության գրաֆ (CFG): Այնուհետև կանչերի գրաֆի գագաթները կարգավորում է տոպոլոգիական հերթականությամբ ու յուրաքանչյուր

ֆունկցիայի համար կառուցում է Hoare եռյակներ: Տոպոլոգիական հերթականությամբ կարգավորելն օգնում է կանչվող ֆունկցիայի համար ինֆորմացիան նախապես ունենալով օգտագործել կանչող ֆունկցիայի վերլուծության ժամանակ: Հաջորդ քայլով կատարվում է միջֆունկցիոնալ արժեքների վերլուծությունը[23] բոլոր փոփոխականների հնարավոր սիմվոլիկ արժեքները ստանալու նպատակով: Այսպիսով, յուրաքանչյուր ֆունկցիայի համար ստացված Hoare եռյակների ճշտությունը ստուգվում է ըստ վերլուծության արդյունքում հավաքված ինֆորմացիայի և նրանց ոչ ճշգրիտ լինելու դեպքում գործիքը տեղեկացնում է համապատասխան սխալի մասին:

4. Ծրագրային կոդի հատկությունների հարցումների համակարգի նախագծում

Ծրագրային կոդի հատկությունների հարցումների համակարգի նախագծումը բաղկացած է երկու հիմնական փուլերից (Նկար 2)՝

1. տվյալների հավաքագրում,
2. հարցումների համակարգ:



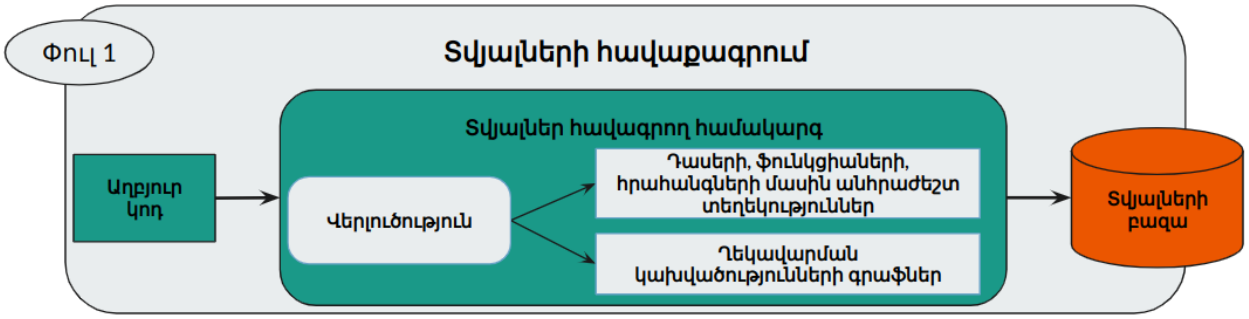
Նկար 2: Հարցումների համակարգի փուլերը

Առաջին փուլը տվյալների հավաքագրման փուլն է, որի նպատակն է ստեղծել տվյալների բազա, որը կպահի աղբյուր կոդի վերաբերյալ անհրաժեշտ տեղեկությունները: Բազմաթիվ բաց կոդով նախագծերից իրականացվում է տվյալների հավաքագրում. աղբյուր կոդից ստանալով անհրաժեշտ տեղեկություններ պահպանում ենք դրանք տվյալների բազայում:

Երկրորդ փուլը բուն հարցումների համակարգի նախագծումն է: Այս փուլում մշակվում է հարցումների համակարգ, որի միջոցով օգտագործողները կատարելով հարցումներ կարող են վերլուծել իրենց ծրագրերի հատկությունները:

4.1. Տվյալների հավաքագրում

Ծրագրային կոդի հատկությունների հարցումների համակարգի համար էական դեր ունի տվյալների հավաքագրման փուլը (Նկար 3)։ Այս փուլում կարևոր է տվյալներ հավաքագրող համակարգի օգտագործումը։ Տվյալներ հավաքագրող համակարգի նպատակն է վերլուծել ծրագրային կոդը և հավաքագրել անհրաժեշտ տեղեկություններ՝ ներառյալ ծրագրի կառուցվածքի, ղեկավարման հոսքի և ներքին փոխկապակցվածությունների վերաբերյալ։



Նկար 3: Փուլ 1, տվյալների հավաքագրում

Տվյալներ հավաքագրող համակարգը մուտքում ստանում է աղբյուր կոդի ֆայլերը և իրականացնում վերլուծություններ՝ ընդգրկելով հետևյալ տեղեկությունների հավաքագրումը.

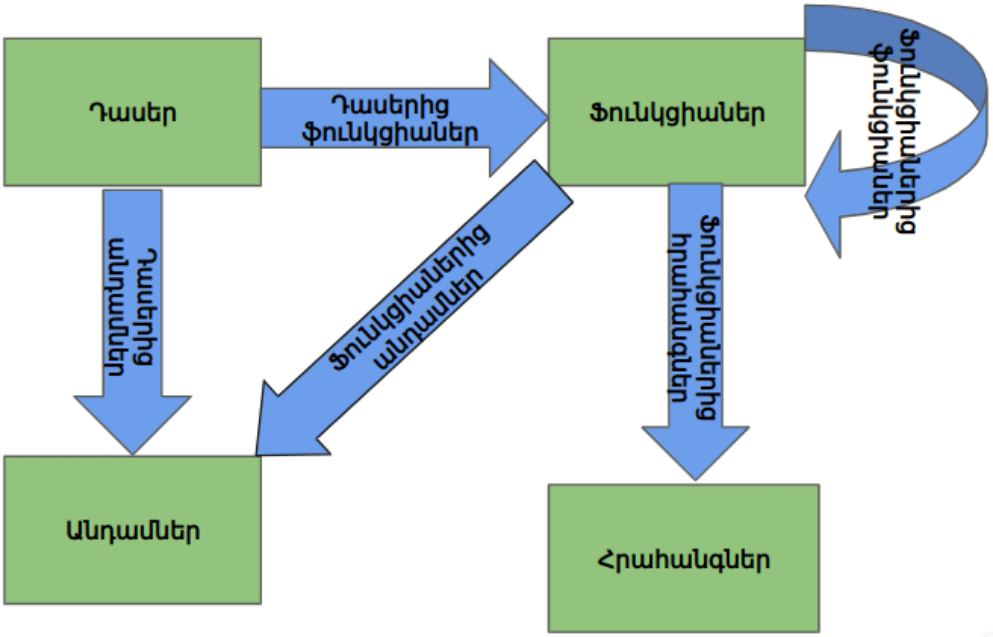
1. Դասերի մասին տեղեկություններ՝ ներառյալ յուրաքանչյուր դասի անունը, դասի դաշտերը, աղբյուր կոդի տողերի համարները, ժառանգականության կապերը, ֆունկցիաները և այլն,
2. Ֆունկցիաների մասին տեղեկություններ՝ ներառյալ յուրաքանչյուր ֆունկցիայի անունը, վերասահմանված ֆունկցիա լինելը, աղբյուր կոդի տողերի համարները, հայտարարող դասի անունը, կանչվող ֆունկցիաները, արգումենտները, լոկալ փոփոխականները, օգտագործվող և սահմանվող դաշտերը, տեսանելիությունը և այլն,
3. Հրահանգների մասին տեղեկություններ՝ ներառյալ յուրաքանչյուր հրահանգների տիպը, աղբյուր կոդի տողերը, կանչվող ֆունկցիաները, օպերանդները, օգտագործվող և սահմանվող փոփոխականները և այլն։

Այս տվյալները պահվում են կառուցված տվյալների բազայում՝ հետագա վերլուծությունների և հարցումների համար օգտագործելու նպատակով։ Դրանք հիմք

են հանդիսանում ծրագրային կոդի հատկությունների հարցումների համակարգի համար:

Տվյալների բազայի նախագծում

Հարցումները արդյունավետ կատարելու նպատակով նախագծվել է տվյալների բազա՝ հաշվի առնելով ռելացիոն ու ոչ ռելացիոն բազաների հատկությունները: Տվյալների բազան ներառում է աղբյուր կոդի հիմնական տարրերի՝ դասերի, դասերի անդամ դաշտերի և հրահանգների մասին տեղեկություններ (Նկար 4):



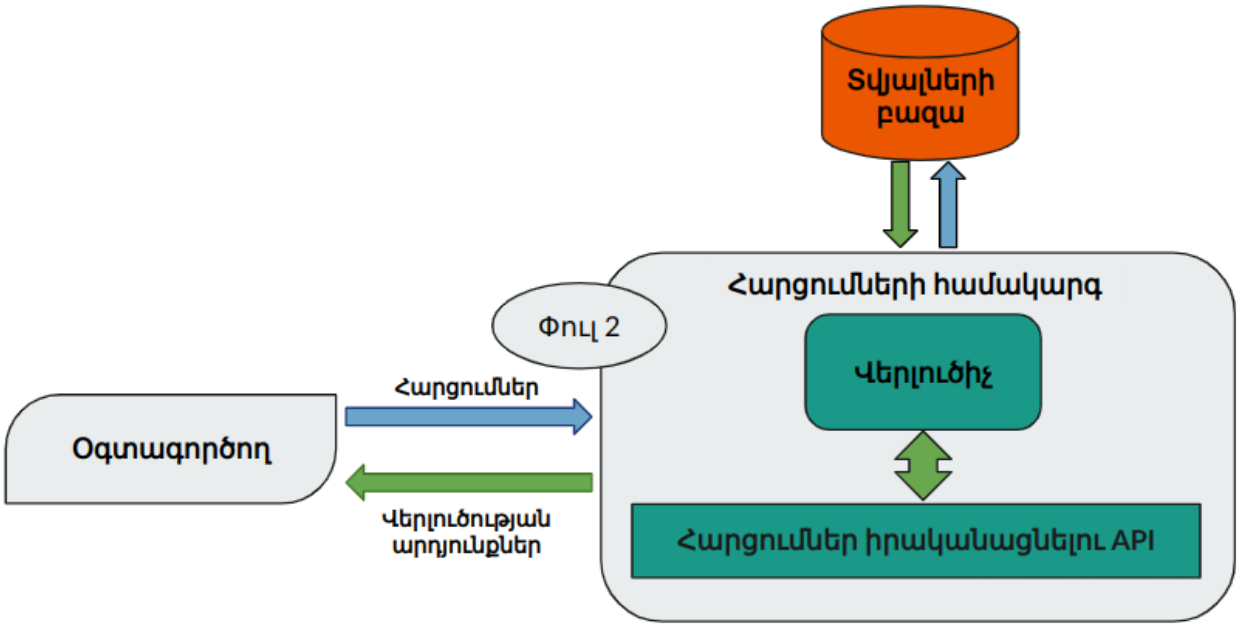
Նկար 4: Տվյալների բազայի կառուցվածքը

4.2. Հարցումների համակարգ

Երկրորդ փուլում նախագծվել է բուն հարցումների համակարգը: Նախագծված համակարգը հանդիսանում է ծրագրային կոդի հատկությունների վերլուծության կարևոր բաղկացուցիչ մասը: Հարցումների համակարգը ճկուն է, ընդլայնելի և հեշտ օգտագործվող: Այս համակարգը թույլ է տալիս ծրագրավորողներին, նույնիսկ առանց խորը վերլուծական գիտելիքների, արագ և ճշգրիտ ստանալ տեղեկատվություն իրենց ծրագրերի կառուցվածքի և հատկությունների մասին:

Նախագծված API-ի միջոցով օգտագործողները կարող են կատարել հարցումներ, որի արդյունքում համակարգը տվյալների բազայից վերցնում է

անհրաժեշտ տվյալներ, կատարում համապատասխան վերլուծություններ և տալիս է հարցմումների պատասխանները (Նկար 5)։



Նկար 5: Փուլ 2, Հարցումների համակարգ

Հարցումների խմբեր

Հարցումների համակարգը տրամադրում է մի շարք հարցումների խմբեր՝ ծրագրի տարբեր հատկությունների վերլուծության համար.

1. Դասերի համար հարցումներ
2. Ֆունկցիաների համար հարցումներ
3. Հրահանգների համար հարցումներ
4. Օգտագործողի կողմից կառավարվող տվյալների վերլուծության հարցումներ

Հարցումների խմբերի բաժանման հիմնական նպատակն օգտագործողներին առավել պարզեցված API տրամադրելն է։

Դասերի համար հարցումներ

Այս հարցումները բաժանվում են երկու խմբի՝

1. որոնման հարցումներ,
2. դասի վերլուծության հարցումներ։

Որոնման հարցումների միջոցով օգտագործողները կարող են որոնել և գտնել ծրագրի դասերն՝ օգտագործելով տարբեր ֆիլտրեր: Նրանք կարող են որոնել դասերն ըստ՝

1. անվանման,
2. պարունակող կամ չպարունակող ֆունկցիաների,
3. անդամ դաշտերի և այլն:

Սա թույլ կտա օգտագործողներին արագ և թիրախավորված կերպով գտնել իրենց հետաքրքրող դասերը:

Երբ օգտագործողը ընտրել է կոնկրետ դաս, նա կարող է անցնել դրա մանրամասն վերլուծությանը: Այդ դասի վերլուծության հարցումների միջոցով հնարավոր կլինի ստանալ տեղեկություններ՝

1. դասի ծնող և ժառանգ դասերի մասին,
2. անդամ ֆունկցիաների մասին,
3. անդամ դաշտերի մասին և այլն:

Նկար 6-ում դասերի հարցումների օրինակ է: Այս օրինակում օգտագործողը որոնման հարցումների միջոցով գտնուն է "_class" անվանման վերջավորություն ունեցող դասերը (class1), այնուհետև գտնում է "d" անունով ֆունկցիա պարունակող դասերը (class2) և գտնում է այդ երկու բազմությունների տարբերությունը (diff1): Այս ամենը կատարվում է տվյալների բազային մեկ անգամ դիմելով (exec()):

```
class1 = Classes().name_suffix("_class")
# {C_class,  D_class, …}
class2 = Classes().with_function_name("d")
# {D_class, …}

diff1 = class1 - class2 # {C_class}
diff1.exec()
```

Նկար 6: Դասերի համար հարցումների օրինակ

Այսպիսով, դասերի համար նախատեսված որոնման և վերլուծության հարցումները թույլ կտան օգտագործողներին տեղեկանալ դասերի և դրանց կապերի մասին:

Ֆունկցիաների համար հարցումներ

Այս հարցումները նույնպես բաժանվում են երկու խմբի՝

1. ֆունկցիաների որոնման հարցումներ,
2. ֆունկցիայի վերլուծության հարցումներ:

Որոնման հարցումների միջոցով օգտագործողները կարող են որոնել և գտնել ֆունկցիաներն՝ օգտագործելով տարբեր ֆիլտրեր: Նրանք կարող են որոնել ֆունկցիաներն ըստ՝

1. անվանման,
2. հայտարարող դասի,
3. արգումենտների,
4. կանչված ֆունկցիաների և այլն:

Սա թույլ կտա օգտագործողներին արագ և թիրախավորված կերպով գտնել իրենց հետաքրքրող ֆունկցիաները:

Կոնկրետ ընտրված ֆունկցիայի համար օգտագործողը կարող է կատարել վերլուծության հարցումներ: Այդ հարցումների միջոցով հնարավոր կլինի ստանալ տեղեկություններ՝

1. տվյալ ֆունկցիան պարունակող դասերի մասին,
2. հրահանգների մասին,
3. այլ ֆունկցիաների հետ կապերի մասին և այլն:

Նկար 7-ում ֆունկցիաների հարցումների օրինակ է: Այս օրինակում օգտագործողը որոնման հարցումների միջոցով գտնուն է "f" անունով ֆունկցիաները (functions), այնուհետև գտնում է բոլոր այն ֆունցիաները, որոնք կանչում են "f" անունով ֆունկցիա (callers):

```
functions = Functions().with_name("f").exec()
# {f}
callers = functions[0].caller_functions().exec()
# {g}
```

Նկար 7: Ֆունկցիաների համար հարցումների օրինակ

Այսպիսով, ֆունկցիաների համար նախատեսված որոնման և վերլուծության հարցումները թույլ կտան օգտագործողներին տեղեկանալ ֆունկցիաների և դրանց փոխկապակցվածության մասին:

Հրահանգների համար հարցումներ

Այս հարցումները ևս բաժանվում են երկու խմբի՝

1. հրահանգների որոնման հարցումներ,
2. հրահանգների վերլուծության հարցումներ:

Օգտագործողները կարող են որոնել հրահանգներն ըստ՝

1. տիպի (օրինակ՝ IF, FOR, CALL և այլն),
2. դասերի,
3. կանչած ֆունկցիաների և այլն:

Սա թույլ կտա օգտագործողներին արագ և թիրախավորված կերպով գտնել իրենց հետաքրքրող հրահանգները:

Երբ օգտագործողն ընտրել է կոնկրետ հրահանգը, նա կարող է անցնել դրա մանրամասն վերլուծությանը: Հրահանգների վերլուծության հարցումների միջոցով հնարավոր կլինի ստանալ տեղեկություններ՝

1. նախորդող և հաջորդող հրահանգների մասին,
2. օպերանդների մասին,
3. տվյալ հրահանգը պարունակող ֆունկցիայի մասին,
4. կանչվող ֆունկցիաների մասին և այլն:

Նկար 8-ում հրահանգների հարցումների օրինակ է: Այս օրինակում օգտագործողը որոնման հարցումների միջոցով գտնուն է "g" անունով ֆունկցիաների վերադարձի հրահանգները (instructions), այնուհետև վերլուծության հարցումների միջոցով որևէ հրահանգի համար գտնում է բոլոր այն հրահանգները, որոնցից կախված է վերադարձվող հրահանգը (backward_df_instructions):

Այսպիսով, հրահանգների համար նախատեսված հարցումները թույլ կտան օգտագործողներին արագ գտնել և մանրամասն վերլուծել ծրագրի ղեկավարման և տվյալների հոսքերը: Սա կարևոր է ծրագրի արդյունավետության և անվտանգության վերլուծման համար:

```
g = Functions().with_name("g").exec()[0]
# {g}
instruction = g.return_instructions().exec()[0]
# {return r}
backward_df_instructions = instruction.backward_df()
# {int a, int x = a, int r = g(f(x), f(10)) + 10)}
```

Նկար 8: Հրահանգների համար հարցումների օրինակ

Օգտագործողի կողմից կառավարվող տվյալների վերլուծության հարցումներ

Ծրագրային կոդի հատկությունների հարցումների համակարգի կարևոր բաղկացուցիչն է օգտագործողի կողմից կառավարվող տվյալների վերլուծության հարցումները: Այս հարցումների նպատակն է օգտագործողներին հնարավորություն տալ վերլուծել, թե ինչպես են իրենց ծրագրի մեջ օգտագործվում օգտագործողի կողմից մուտքագրված տվյալները: Այս հարցումների միջոցով օգտագործողները կկարողանան ստանալ տեղեկություններ ծրագրի կատարման ընթացքում օգտագործողի կողմից կառավարվող տվյալների հոսքի մասին, իմանալ, թե ծրագրի ինչ հատվածներում են այդ տվյալները օգտագործվում, ինչպիսի փոփոխականների հետ են փոխազդում, ինչպես նաև կկարողանան արձանագրել սխալներ կամ անվտանգության խոցելիություններ, որոնք կարող են առաջանալ օգտագործողի մուտքագրած տվյալների հետ փոխազդեցության արդյունքում:

```
1 int main(int argc, char **argv)
2 {
3     char buf[20];
4     strcpy(buf, argv[1]);
  }
```

Նկար 9: Բուֆերի գերբեռնում պարունակող աղբյուր կոդ

Նկար 9-ում ներկայացված է բուֆերի գերբեռնում պարունակող աղբյուր, 4-րդ տողի strcpy ֆունկցիայի երկրորդ արգումենտը (argv[1]) կառավարվում է օգտագործողի կողմից, և հնարավոր է հանգեցնի բուֆերի գերբեռնման: Նկար 10-ում նկարագրված հարցումների միջոցով օգտագործողը գտնում է strcpy ֆունկցիայի երկրորդ արգումենտի վրա ազդող, ագտագործողի կողմից կառավարվող հրահանգները (tainted_paths), և եթե այդ բազմությունը դատարկ չէ, տեղեկացնում է, որ առկա է բուֆերի գերբեռնում:

```
srt_copy = Instructions().with_callee_function_name("strcpy").exec()[0]
dest = srt_copy.get_callee_values()[0].get_arg(1)
tainted_paths = TaintEngine.get_all_tainted_paths_affecting_point(dest)
# [char **argv]
if tainted_paths:
   print("Buffer overflow")
```

Նկար 10: Օգտագործողի կողմից կառավարվող տվյալների վերլուծության հարցումների օրինակ

5. Ծրագրային կոդի հատկությունների հարցումների համակարգի իրականացում

Այս բաժնում ներկայացվում է ծրագրային կոդի հատկությունների հարցումների համակարգի իրականացման գործընթացը: Այն ներառում է հարցումների ավտոմատ գեներացման համակարգի մշակումը, ծրագրի ներքին կառուցվածքը նկարագրող բաղադրյալ գրաֆի կառուցումը, ինչպես նաև տվյալների հոսքի վերլուծությունն ակտիվ փոփոխականների վերլուծությամբ: Այս բաժինը ներկայացնում է համակարգի իրականացման հիմնական մոտեցումներն ու մեթոդները՝ արդյունավետ և հուսալի հարցումների համակարգ ապահովելու համար:

5.1. Հարցումների ավտոմատ գեներացման համակարգ

Ծրագրային կոդի հատկությունների հարցումների համակարգի նպատակն է օգտագործողների համար հեշտացնել ծրագրերի վերլուծության գործընթացը: Այս նպատակին հասնելու համար մշակվել է հարցումների ավտոմատ գեներացման համակարգ:

Ավտոմատ գեներացման համակարգի նպատակն է ավտոմատացնել հարցումների ստեղծման գործընթացը: Այն թույլ է տալիս օգտագործողներին առանց ինքնուրույն տվյալների բազայի հարցումներ գրելու՝ ստանալ անհրաժեշտ տեղեկություններ տվյալների բազայից:

Այսպիսով, օգտագործողներին չի պահանջվի մշակել բարդ հարցումներ տվյալների բազայի համար: Փոխարենը, նրանք կկարողանան օգտագործով մշակված API-ը, ստանալ անհրաժեշտ տեղեկատվություն ծրագրի վերաբերյալ:

Նկար 11a-ում ներկայացված է հարցումների համակարգի API-ի օգտագործմամբ մշակված հարցման օրինակ, իսկ Նկար 11b-ում հարցումների ավտոմատ գեներացման համակարգի կողմից գեներացված տվյալների բազայի հարցումը:

```
classes = (
    Classes()
    .with_name("C")
    .with_function_name("f")
    .with_function_name_not("g")
)
classes.exec()
```

(a)

```
FOR v_1 IN classes
   FILTER v_1.name == "C"
   LET connected_functions = (
       FOR v_4 IN classes_to_functions
           FILTER v_4._from == v_1._id
           FOR v_5 IN functions
               FILTER v_5.name == "g"
               FILTER v_4._to == v_5._id
           RETURN DISTINCT v_5.name
   )
   FILTER LENGTH(connected_functions) == 0
   FOR v_2 IN classes_to_functions
   FOR v_3 IN functions
       FILTER v_3.name == "f"
       FILTER v_2._from == v_1._id
       AND v_2._to == v_3._id
   RETURN DISTINCT v_1
```

(b)

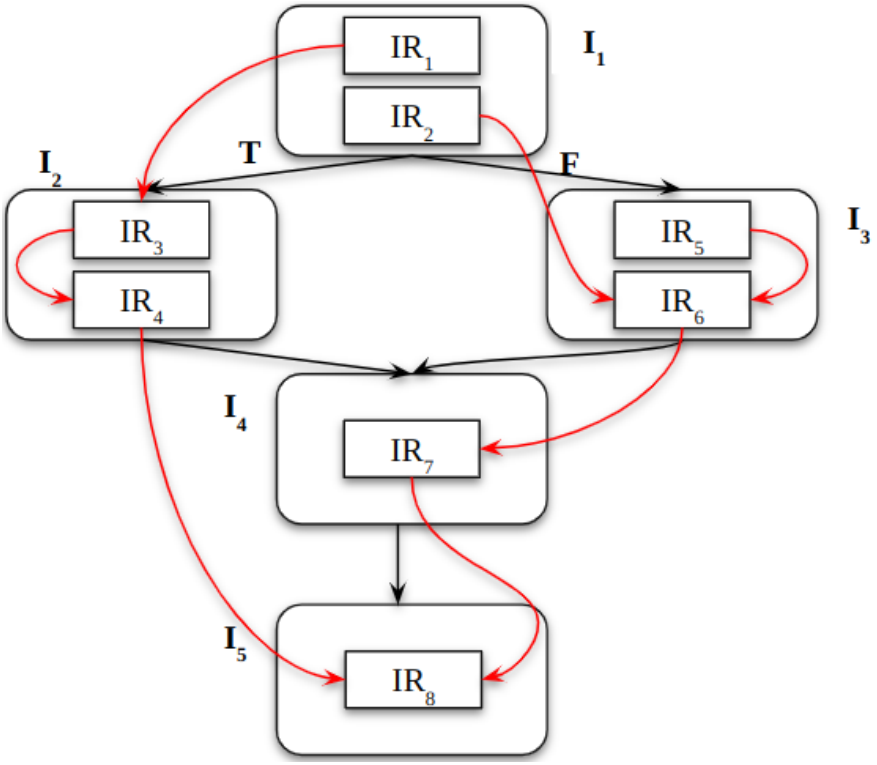
Նկար 11: Հարցումների համակարգի հարցումների (a) և ավտոմատ կերպով գեներացված տվյալների բազայի հարցումների (b) օրինակ

5.2. Բաղադրյալ գրաֆ

Ծրագրային կոդի հատկությունների վերլուծություններն իրականացնելու համար օգտագործվել է բաղադրյալ գրաֆ կառուցվածքը: Բաղադրյալ գրաֆն ուղորդված գրաֆ է, որի գագաթները ներկայացնում են միջանկյալ ներկայացման հրամաններ, ֆունկցիայի պարամետրեր, գլոբալ փոփոխականներ, դասի անդամ փոփոխականներ, իսկ կողերը ցույց են տալիս գրաֆի գագաթների միջև ղեկավարման և տվյալային կախվածությունները (Նկար 12):

Բաղադրյալ գրաֆը կառուցվում է ղեկավարման հոսքի գրաֆի հիման վրա, նրան ավելացնելով հավելյալ գագաթներ և կողեր: Ավելացվող գագաթներն իրենցից ներկայացնում են այն գլոբալ փոփոխականների կամ դասերի դաշտերի հայտարարությունները, որոնից գրաֆի գագաթներին համապատասխանող հրահանգներում առկա է տվյալային կախվածություն: Ավելացվող կողերը ցույց են տալիս գրաֆի գագաթներին համապատասխանող հրահանգների միջև առկա տվյալային կախվածությունները:

Ծրագրի հրահանգների փոխարեն, որպես բաղադրյալ գրաֆի գագաթներ են օգտագործվել միջանկյալ ներկայացման հրամանները: Այդ կերպ բարձրացվել է կատարվող վերլուծությունների ճշտությունը: Նկար 13-ում ներկայացված է աղբյուր կոդը և այդ կոդի բաղադրյալ գրաֆը: Ինչպես երևում է բաղադրյալ գրաֆից,



Նկար 12: Բաղադրյալ գրաֆ

ֆունկցիայի կողմից վերադրաձվող v2 փոփոխականի արժեքը հաստատուն է և կախված չէ ֆունկցիայի x արգումենտից:

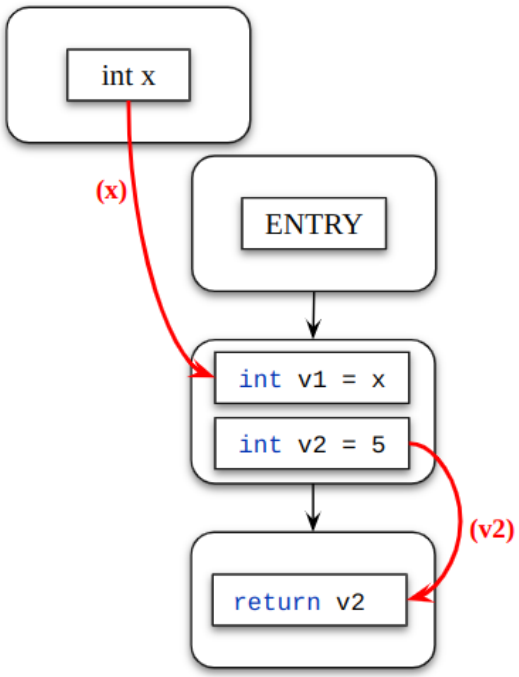
Տվյալների հոսքի վերլուծություն

Տվյալների հոսքի վերլուծությունը ծրագրի կատարման ճանապարհներին տվյալների հոսքերի մասին ինֆորմացիայի հավաքագրումն է: Այն կատարվում է ղեկավարման հոսքի գրաֆի հիման վրա: Տվյալների հոսքի վերլուծության երկու հիմնական մեթոդներն են ակտիվ փոփոխականների և հասնող սահմանումների[16] վերլուծությունները: Գործիքում տվյալների հոսքի վերլուծությունը կատարվել է ակտիվ փոփոխականների վերլուծությամբ:

Ակտիվ փոփոխականների վերլուծություն

Փոփոխականն ակտիվ է ծրագրի որոշակի կետում, եթե իրեն վերագրված արժեքը ղեկավարման հոսքի գրաֆում իր ժառանգների կողմից օգտագործվում է: Վերլուծություն կատարելու համար օգտագործում ենք հետևյալ չորս բազմությունները՝ *def*, *use*, *in* և *out*: Ղեկավարման հոսքի V գագաթի def բազմությունը պարունակում է այն փոփոխականները, որոնք արժեքավորվել են V գագաթում, իսկ use բազմությունը՝ որոնք օգտագործվել են V գագաթում: Ունենալով այս երկու

```
1 int foo(int x) {
2    int v1 = x, v2 = 5;
3    return v2;
4 }
```



Նկար 13: Աղբյուր կոդ օրինակ և համապատասխանող բաղադրյալ գրաֆը

բազմությունները, կարող ենք հաշվել in և out բազմությունները (այսինքն ակտիվ փոփոխականների բազմությունը գագաթ մուտք գործելիս և գագաթից դուրս գալիս), հետևյալ հավասարումների միջոցով[16].

$$\text{IN}[exit] = \varnothing$$
$$\text{IN}[V] = \text{use}[V] \cup (\text{OUT}[V] \setminus \text{def}[V])$$
$$\text{OUT}[V] = \bigcup_{s \in \text{V} successors} \text{IN}[s]$$

Նկատենք, որ in և out բազմությունները հաշվելու հավասարումները ռեկուրսիվ են և փոխկապակցված: Առաջին հավասարումը սահմանում է սահմանային պայմանը: Սա նշանակում է, որ ծրագրի ավարտին ակտիվ փոփոխականներ չկան: Երկրորդ հավասարումով ասվում է, որ փոփոխականը գագաթ մուտք գործելիս ակտիվ է, եթե այն օգտագործվում է գագաթում, կամ դուրս է գալիս գագաթից առանց վերասահմանվելու: Երրորդ հավասարումով ասվում է, որ գագաթից դուրս գալիս փոփոխականն ակտիվ է այն և միայն այն դեպքում, երբ այն ակտիվ է իր ժառանգ գագաթներից գոնե մեկում:

def և use բազմություններ

Յուրաքանչյուր գագաթում առկա փոփոխականները դիտարկվում են կամ որպես սահմանվող(def) կամ որպես օգտագործվող(use) փոփոխականներ: Որոշ փոփոխականներ դիտարկվում են և որպես սահմանվող, և որպես օգտագործվող փոփոխականներ: Օրինակ, եթե դիտարկենք “++” post increment օպերատորը, “var++” արտահայտությունում var փոփոխականը և օգտագործվում է և նորից սահամանվում: Դիտարկված փոփոխականներն ավելացվում են def և use բազմություններում:

Այսպիսով կատարվում է def և use փոփոխականների վերլուծություն և ղեկավարման հոսքի գրաֆի յուրաքանչյուր գագաթի համար տրվում է այդ գագաթի def և use բազմությունները: Նկար 14-ում ցուցադրված է Նկար 13-ի օրինակին համապատասխանող def և use բազմությունները:

```
1  int foo(int x) {
2     int v1 = x, v2 = 5;
3     return v2;
4  }
```

0	`int x`	`def{x} : use{}`
1	`int foo(int x) {`	`def{} : use{}`
2	`int v1 = x;` `int v2 = 5;`	`def{v1} : use{x}` `def{v2} : use{}`
3 4	`return v2;` `}`	`def{} : use{v2}`

Նկար 14: def և use բազմություններ

6. Դինամիկ հիշողության արտահոսքի սխալների հայտնաբերում

Ծրագրային կոդի հատկությունների հարցումների համակարգը օգտագործվել է դինամիկ հիշողության արտահոսքեր (memory leaks)[9] հայտնաբերելու համար: Այդ նպատակով, համակարգի API-ի օգտագործմամբ իրականացվել է համապատասխան ստուգիչ (checker) (Տես Նկար 15): Ստուգիչի աշխատանքը հիմնված է հետևյալ ալգորիթմի վրա.

1. Գտնել malloc, calloc ֆունկցիաների կանչերի հրահանգները,
2. Յուրաքանչյուր malloc, calloc կանչի համար որոշել նպատակային օպերանդը (destination),
3. Յուրաքանչյուր նպատակային օպերանդից գտնել կախվածություն ունեցող և free կանչող հրահանգները (forward_df_free_instructions),
4. Յուրաքանչյուր ճանապարհի համար, որ սկսվում է malloc, calloc կանչից և ավարտվում է ծրագրի վերջում, ստուգել, արդյոք այդ ճանապարհը պարունակում է forward_df_free_instructions հրահանգներից որևէ մեկը:
5. Եթե ոչ, արձանագրել դինամիկ հիշողության արտահոսք, եթե ճանապարհն իրական է:

Հարցումների համակարգի օգտագործմամբ ստեղծված ստուգիչը թույլ է տալիս ավտոմատ կերպով հայտնաբերել այնպիսի դեպքեր, երբ ծրագրի աշխատանքի ընթացքում հիշողությունը չի ազատվում ամբողջությամբ: Սա կարևոր խնդիր է ծրագրային ապահովման որակի և արդյունավետության տեսանկյունից, քանի որ չազատված հիշողությունը կարող է հանգեցնել ծրագրի կայունության և կատարողականության նվազեցմանը:

6.1. Թեստավորում

Մշակված գործիքը թեստավորվել և համեմատվել է աշխարհում արդեն գոյություն ունեցող այլ գործիքների հետ: Ինչպես նաև գործիքի միջոցով թեստավորվել են բաց կոդով հասանելի ավելի քան 100 պրոեկտներ, որոնք իրականացված են մեծամասամբ C ծրագրավորման լեզվով:

```
malloc_instructions = Instructions().with_callee_function_names(
    ["malloc", "calloc"]
).exec()
for malloc_instruction in malloc_instructions:
    check_free_for_alloc(malloc_instruction)


def check_free_for_alloc(malloc_instruction):
    buf = malloc_instruction.get_dest()
    forward_df_free_instructions = {
        forward_df_instruction
        for forward_df_instruction in buf.forward_df()
        if "free" in forward_df_instruction.callee_names()
    }

    for path in malloc_instruction.all_next_instructions_paths():
        if not forward_df_free_instructions.intersection(path):
            if sym_executor.path_exists(path):
                print("Memory leak")
```

Նկար 15: Հարցումների համակարգի API-ի օգտագործմամբ գրված դինամիկ հիշողության արտահոսքի ստուգիչ

Արդյունքների համեմատումը Juliet թեստերի հավաքածույի վրա

MLH-ը թեստավորվել է Juliet թեստերի հավաքածույի վրա, որը նախատեսված է ծրագրային ապահովման գործիքների թեստավորման և արդյունքների գնահատման համար: Այն Software Assurance Metrics and Tool Evaluation(SAMATE)[12] նախագծի մասն է կազմում, որը մշակվել է National Institue of Standards and Technology(NIST)-ի կողմից: Ջուլիետ թեստերի հավաքածուն ներառում է ավելի քան 120,000 թեստերի օրինակներ առանձնացված զանազան խնդիրների համար ինչպիսիք են դինամիկ հիշողության արտահոսքը, բուֆերի գերհագեցումը և այլն: Այն ներառում է թեստեր C/C++, Java և Ada լեզուներով: Թեստավորման համար հավաքածուից առանձնացվել են միայն դինամիկ հիշողության արտահոսքի համար նախատեսված օրինակները (CWE401_Memory_Leak[3]): Մշակվել է թեստավորման համակարգ, որը գնահատում է ունեցած գործիքների արդյունավետությունը վերը նշված հավաքածուի համար:

6.2. Արդյունքներ

Ստորև ներկայացված է առաջատար այլ գործիքների և նոր մշակված գործիքի արդյունավետության աղյուսակը (Աղյուսակ 1)։

Name	True Positives	True Negatives	False Positives	False Negatives	F1 score
CSA	536	4481	125	332	0.7011
Infer	262	4392	214	606	0.3899
SMOKE	496	4510	96	372	0.6795
PCA	486	4342	264	382	0.6007
SVF	452	4168	438	416	0.5142
MLH	868	4606	0	0	1

Աղյուսակ 1: Juliet թեստերի հավաքածույի վրա համեմատման արդյունքների

Դինամիկ հիշողության արտահոսքի հայտնաբերումը առաջատար բաց կոդով գործիքներում

Մի շարք հատուկ առանձնացված բաց կոդով գործիքներ դասակարգված են ըստ իրենց ակտիվության և կարևորության։ Այդ գործիքներից ավելի քան հարյուրը թեստավորվել են նոր մշակված գործիքի միջոցով՝ դինամիկ հիշողության արտահոսքի խնդիրները հայտնաբերելու նպատակով։ Ստացված ճիշտ արդյունքները հրապարակվել և հաստատվել են, սխալ ստացված արդյունքների համար կատարվել է վերլուծություն և մշակվել հետագա պլան գործիքում առկա թերությունները վերացնելու նպատակով։ Նկար 16-ում ներկայացված է ֆունկցիայի օրինակ coturn[2]-ից, որում հարցումների համակարգի API-ի օգտագործմամբ գրված սստուգիչի կողմից հայտնաբերվել է դինամիկ հիշողության արտահոսքի դեպք։

```
1412  void admin_server_receive_message(struct bufferevent *bev, void *ptr) {
1413    UNUSED_ARG(ptr);
1414
1415    struct turn_session_info *tsi = (struct turn_session_info *)malloc(sizeof(struct turn_session_info));
1416    turn_session_info_init(tsi);
1417    int n = 0;
1418    struct evbuffer *input = bufferevent_get_input(bev);
1419
1420    while ((n = evbuffer_remove(input, tsi, sizeof(struct turn_session_info))) > 0) {
1421      if (n != sizeof(struct turn_session_info)) {
1422        fprintf(stderr, "%s: Weird CLI buffer error: size=%d\n", __FUNCTION__, n);
1423        continue;
1424      }
1425
1426      ur_map_value_type t = 0;
1427      if (ur_map_get(adminserver.sessions, (ur_map_key_type)tsi->id, &t) && t) {
1428        struct turn_session_info *old = (struct turn_session_info *)t;
1429        turn_session_info_clean(old);
1430        free(old);
1431        ur_map_del(adminserver.sessions, (ur_map_key_type)tsi->id, NULL);
1432      }
1433
1434      if (tsi->valid) {
1435        ur_map_put(adminserver.sessions, (ur_map_key_type)tsi->id, (ur_map_value_type)tsi);
1436        tsi = (struct turn_session_info *)malloc(sizeof(struct turn_session_info));
1437        turn_session_info_init(tsi);
1438      } else {
1439        turn_session_info_clean(tsi);
1440      }
1441    }
```

Նկար 16: Դինամիկ հիշողության արտահոսքի իրական օրինակ

Եզրակացություն

1. Նախագծվել և իրականացվել է ծրագրային կոդի հատկությունների հարցումների համակարգ, որը պահանջում է միայն համապատասխան ծրագրավորման լեզվի իմացություն
2. Օգտագործելով հարցումների համակարգը՝ ստեղծվել է դինամիկ հիշողության արտահոսքի սխալների հայտնաբերման ալգորիթմ

    - Թեստավորվել են C լեզվով գրված բաց կոդով հասանելի լավագույն 100 նախագծերի վրա (ըստ github star-երի), որի արդյունքում հայտնաբերվել են սխալներ:

Գրականություն

[1] Andersen's pointer analysis. https://github.com/grievejia/andersen.

[2] Coturn. https://github.com/coturn/coturn.

[3] Cwe-401. https://cwe.mitre.org/data/definitions/401.html.

[4] Clang. https://clang.llvm.org/, .

[5] Clang-analyzer. https://clang-analyzer.llvm.org/, .

[6] Fastcheck. https://www.cs.cornell.edu/projects/crystal/fastcheck/.

[7] Infer. https://fbinfer.com/.

[8] The llvm compiler infrastructure. https://github.com/llvm/llvm-project.

[9] Memory leak. https://en.wikipedia.org/wiki/Memory_leak.

[10] Pca. https://github.com/awen-li/PCA.

[11] Pinpoint. https://whichbug.github.io/artifact.html.

[12] Software assurance metrics and tool evaluation(samate). https://www.nist.gov/programs-projects.

[13] Sil. https://fbinfer.com/odoc/next/infer/IR/Sil/index.html.

[14] Smoke. https://smokeml.github.io.

[15] Z3. https://github.com/Z3Prover/z3.

[16] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 2 edition, 2006.

[17] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design Implementation*, San Diego, California, USA, June 10-13 2007.

[18] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576--580, 1969.

[20] Y. Jung and K. Yi. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008*, Tucson, AZ, USA, June 2008.

[21] W. Li, H. Cai, Y. Sui, and D. Manz. Pca: memory leak detection using partial call-path analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2020.

[22] S. Qingkai, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *PLDI 2018: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.

[23] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, January 1995.

[24] Y. Sui and J. Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, March 2016.