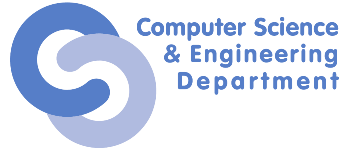
Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare



TEZĂ DE DOCTORAT

REZUMAT

Consilidarea Securității Arhitecturilor IoT

Conducător Științific:
Prof. Dr. Ing. Răzvan-Victor Rughiniș

Autor:
Ing. Constantin Eduard Stăniloiu

București, 2023

Cuprins

1 Introducere **1**
1.1 Contribuțiile Tezei 4

2 Îmbunătățirea securității kernelului Linux **5**
2.1 Module Linux Kernel mai sigure folosind limbajul de programare D 5
2.2 Integrarea automată a codului D cu kernelul Linux 11

3 Îmbunătățirea securității microserviciilor și aplicațiilor **18**
3.1 Un nou framework de colecții pentru biblioteca standard a limbajului de programare D 18
3.2 Extinderea suportului API client-server pentru limbaje de programare sigure pentru memorie 24

4 Auditarea securității arhitecturilor și aplicațiilor IoT **30**
4.1 IoT Fuzzing folosind AGAPIA și Framework-ul River 30
4.2 Execuție concolică scalabilă a binarelor COTS cu cadrul River 36

5 Concluzii **44**
5.1 Rezumatul tezei 44
5.2 Contribuții 45
5.3 Lista de publicații 46

Capitolul 1

Introducere

Securitatea software-ului este esențială pentru sistemele moderne datorită numărului tot mai mare de atacuri cibernetice din ultimii ani. Lumea a fost martoră la creșterea atacurilor de securitate cibernetică în ultimii ani, așa cum este prezentat în Figura 1.1. Exploatarea vulnerabilităților în sălbăticie a costat companiile de miliarde[1], cercetătorii de la IBM System Science Institute estimează că repararea unei vulnerabilități costă de 100 de ori mai mult decât costurile de dezvoltare[2].

Figura 1.1: Creșterea atacurilor cibernetice săptămânale globale per organizație[3]

Numărul tot mai mare de dispozitive IoT (și furnizorii acestora) [4] ridică preocupările legate de confidențialitate și securitate. Numărul de vulnerabilități în sistemele și dispozitivele computerizate, în special pentru dispozitivele IoT, este în creștere, iar această tendință este de așteptat să continue în viitor datorită creșterii numărului de dispozitive IoT[5]. Această creștere a vulnerabilităților reprezintă un risc semnificativ, deoarece extinde suprafața potențială de atac și face mai probabil ca aceste vulnerabilități să fie exploatate de actori rău intenționați[6, 7]. O modalitate de a înțelege această idee este să analizăm cele două dovezi furnizate: Figura 1.2 arată că numărul vulnerabilităților este în creștere, multe dintre ele fiind de severitate medie și mare. Figura 1.3 arată că numărul de dispozitive IoT este de așteptat să se dubleze până în 2030. Aceste două informații sunt legate, deoarece creșterea numărului de dispozitive IoT va duce probabil la o creștere a numărului de vul-

CVSS Severity Distribution Over Time

This visualization is a simple graph which shows the distribution of vulnerabilities by severity over time. The choice of LOW, MEDIUM and HIGH is based upon the CVSS V2 Base score. For more information on how this data was constructed please see the NVD CVSS page .

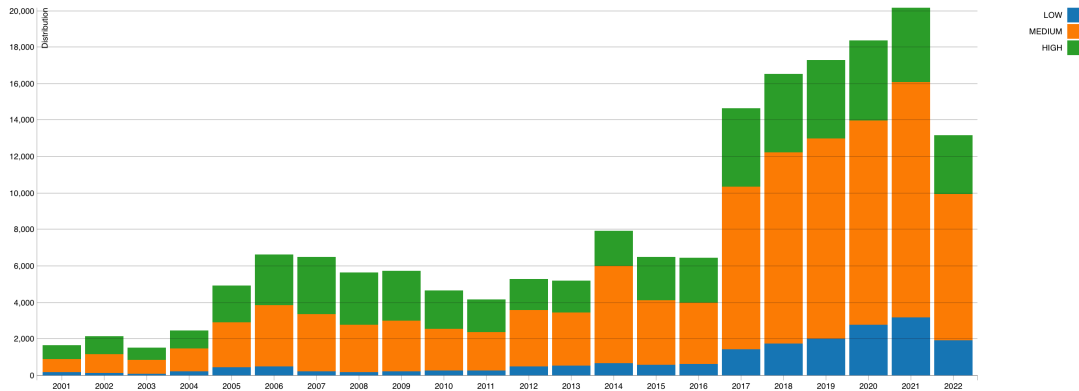


Figura 1.2: Distribuția severității sistemului comun de scorare a vulnerabilităților (CVSS) în timp[8]

nerabilități, deoarece există mai multe dispozitive care pot fi potențial exploatate.

Pentru a rezolva această problemă, designerii de limbaje de programare au dezvoltat limbaje de programare a sistemelor, cum ar fi Rust și D, care se concentrează pe furnizarea de siguranță a memoriei și creșterea productivității dezvoltatorului fără a sacrifica performanța, toate păstrând dimensiunea binară rezultată mică. Aceasta înseamnă că siguranța memoriei și prototiparea rapidă nu mai sunt o caracteristică a limbajelor de programare de nivel înalt, avide de resurse. Aceste proprietăți cheie permit dezvoltarea dispozitivelor IoT în alternative mai sigure (față de C) care pot preveni vulnerabilitățile încă din momentul compilației.

Sistemele de operare sunt omniprezente. Orice smartphone, laptop, cloud mașina de calcul, dispozitivul purtabil sau Internet of Things (IoT) are un sistem de operare în culise [10]. Datorită faptului că este o componentă atât de fundamentală a fiecărui dispozitiv, funcționarea sistemul trebuie să fie sigur și securizat, așa cum orice vulnerabilitate, oricât de mică, are potențialul compromite întregul sistem [11].

Nucleele celor mai populare sisteme de operare, inclusiv Linux, Android, MacOS, iOS și Windows [12], sunt implementate folosind limbajul de programare *C*. Practic orice mobil, computerul desktop sau laptop folosește limbajul *C* în centrul său. În Linux, majoritatea codului sursă este reprezentată de implementări de drivere de dispozitiv [13]. Driverul de dispozitiv este o componentă critică a sistemului de operare și, ca atare, este importantă securitatea sa nu poate fi supraevaluată [14]. Este foarte bine cunoscut faptul că C este în prezent limbajul de bază pentru programarea sistemelor. Dar, la fel de cunoscute sunt și potențiala securitate problemele pe care le expune, inclusiv, dar fără a se limita la, erori legate de memorie, cum ar fi depășirile de buffer.

Un astfel de exemplu este cazul driverului *RTLWIFI*, construit pentru cipurile Wi-Fi Real-

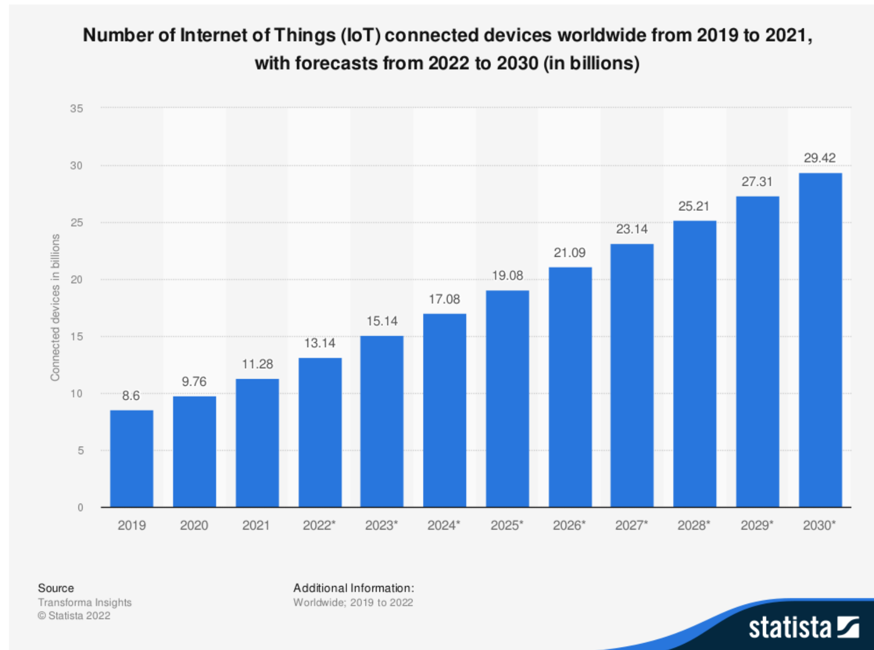


Figura 1.3: Numărul de dispozitive conectate la Internet of Things (loT) la nivel mondial din 2019 până în 2021, cu previziuni din 2022 până în 2030 (în miliarde)[9]

tek, care, în 2019, a permis declanșarea depășirii tamponului în interiorul nucleului Linux, fără niciun utilizator orice interacțiune [15]. Evidențiatul aici este pe **fără interacțiune cu utilizatorul**, sugerând defectul era intrinsec șoferului însuși.

Deoarece codul driverelor de dispozitiv reprezintă cea mai mare parte a nucleului Linux, acesta reprezintă, de asemenea, o suprafață uriașă de atac. Credem că limbajul de programare D poate fi un candidat excelent pentru scrierea driverelor de dispozitiv kernel într-un limbaj de programare sigur. D este pe deplin compatibil cu C și verifică mecanic, în timpul compilării, codul utilizatorului pentru comportament nesigur. Limbajul are, de asemenea, funcții puternice de compilare care fac posibilă scrierea unui cod foarte specific, de înaltă performanță. De asemenea, are o curbă de învățare lină și nu forțează caracteristicile de limbă asupra utilizatorilor, permițându-le să-și îmbunătățească treptat baza de cod, pe măsură ce devin mai confortabili cu limba.

Creșterea securității nucleului Linux crește automat securitatea tuturor aplicațiilor care rulează pe acesta. Dispozitivele ușoare, cum ar fi senzorii, au un firmware flash pe ele și nu rulează deasupra unui nucleu. Deși acesta este cazul, aceștia sunt de obicei conectați la un dispozitiv gateway care rulează pe Linux, astfel încât să poată trimite datele colectate, prin Internet, pentru ca utilizatorul final să le acceseze. Din această cauză, considerăm că o vulnerabilitate în nucleul dispozitivului gateway ar afecta întregul sistem. Astfel, creșterea securității nucleului Linux va aduce beneficii tuturor dispozitivelor.

După cum sa menționat anterior, o mulțime de aplicații IoT comunică cu serviciile cloud. Modul standard de acces la servicii este printr-un API RESTful: punctele finale disponibile

ale unui serviciu sunt publicate folosind specificația OpenAPI[16] - un fișier JSON care descrie URI-ul punctului final, metodele acceptate de URI, argumentele acceptate și valorile returnate așteptate. Pentru a întări securitatea aplicațiilor dezvoltate, sunt necesare biblioteci securizate și implementări API. Ne propunem să oferim o modalitate de a genera automat biblioteci securizate pe partea clientului pe baza unei date API JSON descriere.

Testarea și validarea securității sistemelor IoT este dificilă, din cauza faptului că majoritatea software-ului este proprietar (COTS - custom off-the-shelf - binare) și natura încorporată a sistemului face dificilă colectarea datelor și a dispozitivului de audit. Securitate. Utilizatorul este forțat să aibă încredere într-o terță parte și nu este capabil să valideze software-ul, deoarece nu are acces la codul sursă. Testarea Fuzz este o procedură de testare automatizată stabilită care ajută echipele să descopere căi și vulnerabilități netestate în produsele lor. Scopul este de a implementa un fuzzer care poate testa binare COTS, oferind utilizatorilor IoT un instrument de audit pentru dispozitivele lor.

1.1 Contribuțiile Tezei

Restul acestui rezumat este structurat după cum urmează:

Capitolul 2 prezintă contribuțiile noastre privind îmbunătățirea securității nucleului Linux. Acesta detaliază modul în care limbajul de programare D poate fi utilizat pentru a scrie drivere securizate în ecosistemul kernelului. Arătăm că codul C poate fi portat cu ușurință în D fără a suporta penalități de performanță. De asemenea, prezentăm munca noastră care ușurează și mai mult procesul, prin traducerea automată a definițiilor structurii de date ale nucleului în cele compatibile cu D.

Capitolul 3 prezintă contribuțiile noastre în ceea ce privește îmbunătățirea securității aplicațiilor. În primul rând, prezintă munca noastră în dezvoltarea unui nou cadru de colecții pentru biblioteca standard D. Colecțiile noastre sunt capabile să deducă siguranța operațiunilor din tipul conținut, furnizat de utilizator. Arătăm că implementarea noastră poate oferi beneficii de performanță de până la 2x în comparație cu implementările existente ale bibliotecii standard. În continuare, vă prezentăm contribuția noastră în construirea unui generator de biblioteci bazat pe Specificațiile OpenAPI.

Capitolul 4 prezintă munca noastră privind îmbunătățirea cadrului de fuzzing River. În primul rând, detaliază completările care permit cadrului să fuzz rețelele IoT. În continuare, prezintă provocările efectuării execuției simbolice pe binare COTS și limitările motoarelor de execuție simbolice actuale, de ultimă generație și modul în care soluția noastră le depășește.

Capitolul 5 se încheie, discută lucrările viitoare și prezintă lista publicațiilor.

Capitolul 2

Îmbunătățirea securității kernelului Linux

2.1 Module Linux Kernel mai sigure folosind limbajul de programare D

Nucleul Linux este utilizat pe o gamă largă de dispozitive: servere, supercalculatoare, dispozitive inteligente și sisteme încorporate. Având în vedere popularitatea sa, securitatea nucleului a devenit un subiect critic de cercetare. În consecință, au fost create o gamă largă de instrumente terțe pentru a detecta erori în implementarea sa. Cu toate acestea, noi vulnerabilități sunt descoperite și exploatate în fiecare an. Explicația acestui fenomen constă în faptul că limbajul de programare care este utilizat pentru implementarea nucleului, C, este conceput pentru a permite operațiuni de memorie nesigure. În acest capitol, arătăm că este posibilă tranziția incrementală a codului kernelului de la C la un limbaj de programare sigur pentru memorie, D, prin portarea și integrarea unui driver de dispozitiv. În plus, propunem o serie de transformări de cod care permit compilatorului D să raționeze despre siguranța anumitor operațiuni de memorie. Implementarea noastră mărește garanțiile de securitate ale nucleului fără a suporta penalități de performanță.

2.1.1 Introducere

Nucleul sistemului de operare Linux este utilizat pe scară largă de o gamă largă de hardware, inclusiv supercalculatoare [17], servere [18], dispozitive portabile [19] și sisteme încorporate [20], ceea ce îl face cel mai popular sistem de operare.

Linux rulează într-un mod de procesor privilegiat (numit *modul kernel* sau *modul supervizor*) cu acces complet la memoria sistemului și dispozitivele. Un atac de succes pe Linux va oferi atacatorului controlul deplin asupra întregului sistem, făcându-l o țintă căutată. Astfel de atacuri reprezintă o întâmplare comună. Figura 2.1 evidențiază numărul de vulnerabilități descoperite în Linux pe baza rapoartelor Common Vulnerability and Exposure (CVE) [21], care reprezintă o medie de aproximativ 250 de rapoarte pe an. Nu există nicio modalitate de a ști câte vulnerabilități nedescoperite există și sunt exploatate activ.

Pentru a se proteja de potențiale atacuri de securitate, nucleul Linux folosește o varietate de mecanisme de auto-protecție [22, 23], cum ar fi Kernel Address Space Layout Randomization (KASLR), Kernel Page Izolarea mesei (KPTI), protectorul stivei etc. Cu toate acestea,

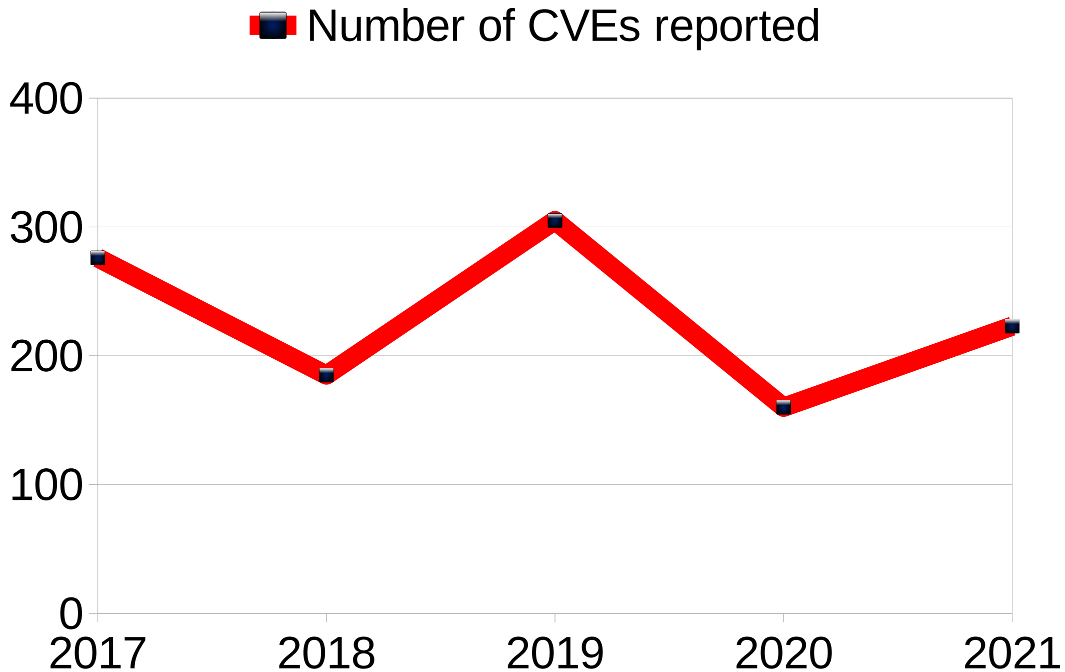


Figura 2.1: Numărul de rapoarte comune de vulnerabilitate și expunere (CVE).

aceste vulnerabilități apar din cauza unei combinații de greșeli ale programatorului și lipsei suportului de siguranță din limbajul de programare. Nucleul Linux este scris în mare parte în C, un limbaj rapid, dar cu caracteristici de siguranță minime. Sintaxa C permite accesul ușor la memoria programului, dar este, de asemenea, principala sursă de vulnerabilități, cum ar fi depășirile de buffer, pointerii către date expirate, pointeri către memoria neinițializată.

În acest capitol propunem o abordare complementară pentru securizarea nucleului Linux: utilizarea unui limbaj de programare sigur, adică un limbaj cu caracteristici care să ajute dezvoltatorul în scrierea codului securizat.

Alegerea noastră este limbajul de programare D [24], care are o sintaxă similară cu C/C++ și oferă programare modernă și caracteristici de siguranță. D își propune să ofere cât mai multe dintre beneficiile de performanță ale limbajului de programare C, cu cât mai puține dezavantaje de securitate. Astfel, răspundem la întrebarea generală de cercetare: *Pot fi rescrise componentele software critice (driverele sistemului de operare) într-un limbaj de programare sigur cu un efort rezonabil, menținând în același timp performanța?*.

Rescrierea unei componente software dintr-o limbă mai veche într-una mai nouă oferă posibilitatea de a utiliza funcții de programare mai moderne. În cazul nostru există beneficii de siguranță, cum ar fi: verificarea limitelor matricei, variabile imuabile, funcții sigure, inițializare garantată, șabloane și multe altele.

Pe scurt, în acest capitol aducem următoarele contribuții:

- Am arătat că este posibil să folosim un limbaj sigur în nucleu prin portarea cu succes a unui driver de rețea Linux la D. Mai exact, am portat `virtio_net` din cadrul virtio [25].
- Proiectăm și implementăm tehnici care se bazează pe caracteristici specifice limbajului D pentru a îmbunătăți driverele kernel-ului Linux. Costurile de performanță sunt neglijabile, beneficiile de securitate fiind oferite de limbajul de programare D.
- Oferim o metodologie pentru portarea modulelor kernel-ului Linux în limbajul de programare D. Demonstrată de portul nostru de succes, metodologia poate fi folosită pentru a porta alte module de kernel Linux.

2.1.2 Metodologia de portare a modulelor la D

Introducerea codului D în Kernel-ul Linux

Există două opțiuni pentru extinderea funcționalității nucleului Linux: (1) legarea statică a unui fișier obiect nou direct la nucleul de bază sau (2) construirea unui modul încărcat și conectarea dinamică a acestuia după cum este necesar.

Cea mai bună practică recomandată este îmbunătățirea funcționalității nucleului Linux prin încărcare dinamică. Acest lucru ajută la păstrarea bazei de cod a nucleului neaglomerate și ușor de întreținut. În plus, utilizarea modulelor încărcate permite personalizarea timpului de execuție și ajută la menținerea suprafeței de atac reduse, reduce riscurile de vulnerabilitate și îmbunătățește securitatea.

Portarea unui modul în limbajul de programare D necesită:

- scrierea codului sursă corespunzător în D
- oferind puncte de intrare în modul ca funcții de interfață C
- actualizarea fișierelor de sistem de compilare pentru a construi legătura noului modul

Pentru a 2-a cerință, pentru a permite nucleului să comunice cu un modul scris în D, trebuie creată o interfață C. Pentru aceasta, scopul este de a include minimumul strict: macro-urile de înregistrare și stub-urile corespunzătoare care apelează la implementarea D. În cele din urmă, configurația de compilare a nucleului Linux trebuie să se adapteze la a treia cerință.

Sistemul de construire a nucleului presupune că are de-a face cu fișierele sursă C și încearcă să construiască fișierele obiect în consecință. Din fericire, sistemul de compilare acceptă și binare de obiecte pre-construite, ca dependențe, pe care le va lega cu fișierele obiect pe care le-a construit pentru a crea modulul kernel. Acest lucru se face prin schimbarea numelui dependenței din *module-file.o* în *module-file.o_shipped*. Pentru a lega fișierele obiect D într-un modul kernel, fișierele sursă D trebuie să fie compilate în prealabil cu comutatorul `-betterC` și să aibă numele atașat cu sufixul *.o_shipped*. Odată construite, acestea pot fi integrate în sistemul de construcție și conectate, alături de obiecte obișnuite, pentru a forma un modul (obiect kernel `.ko`).

Portarea modulelor la D

Portând modulul kernel, am urmat cinci pași, inclusiv testarea și evaluarea comparativă:

1. Portați tipurile de date, cum ar fi `structs` și `typedefs`, cerute de implementarea modulului, astfel încât aspectul să fie consecvent între obiectele D și C. Acest lucru asigură că modulul va funcționa corect după procesul de portare.
2. Portează implementarea modulului în mod incremental: port o rutină apoi rulează teste simple care verifică funcționalitatea modulului. Repetați procesul până când toate funcțiile au fost portate.
3. După finalizarea procesului de portare, primul set de benchmark-uri ar trebui rulat pentru a evalua comportamentul modulului. Aceasta implică compararea versiunii D a modulului cu versiunea originală C pentru a evalua orice diferențe de corectitudine.
4. Îmbunătățiți implementarea prin adăugarea de caracteristici de limbaj: `@safe`, înlocuiți macrocomenzi și modele cu metaprogramare și alte caracteristici de siguranță a memoriei.
5. Benchmark-ul este executat pentru a doua oară pentru a evalua performanța de rulare. Comparați implementarea modulului original cu variantele D brute și îmbunătățite.

Îmbunătățiri de siguranță

Acestea sunt o serie de îmbunătățiri de securitate oferite de limbajul de programare D. Ele sunt folosite pentru a implementa și a construi modulul kernel nou implementat în D.

Variabilele sunt inițializate la o valoare implicită de tipul lor, eliminând erorile de inițializare. În plus, variabilele locale marcate cu cuvântul cheie **scope** sunt limitate la domeniul de aplicare al funcției, reducând prezența pointerilor atârnați.

În D, spre deosebire de C, sistemul de tip nu permite **transformări implicite** de la tipul `void*` la orice alt tip de pointer, deoarece acest lucru ar putea duce la erori de corupere a datelor silențioase. D necesită o distribuție explicită pentru conversia indicatoarelor de diferite tipuri.

D nu permite **implicit switch fall-through**. D folosește, de asemenea, instrucțiunea `final switch` în cazul în care cazul implicit nu este necesar și nici nu este permis, util atunci când instrucțiunea `default` este inutilă. Instrucțiunea `final switch` este utilă în special atunci când este aplicată pe un tip `enum`, deoarece va impune utilizarea tuturor membrilor enum din instrucțiunile `case`.

Vectorii statici sunt în mod implicit verificați pentru accese asupra elementelor dincolo de limite.

Slices specifică o parte a unui tablou, printr-o referință și informații despre lungime. Sunt

folosite pentru a verifica limitele matricelor alocate dinamic. Rețineți că acest lucru necesită cunoașterea dimensiunii inițiale a matricelor alocate dinamic.

Șabloane poate fi folosit ca înlocuitor pentru pointerii C void și definițiile macro pentru programarea generică, permițând astfel verificări ale sistemului de tip.

Funcțiile sigure (adnotate cu `@safe`) sunt verificate static împotriva cazurilor de comportament nedefinit. Variabilele neinițializate, luarea adresei unei variabile locale, turnările explicite, aritmetica pointerului, apelurile la funcții nesigure și multe altele, sunt interzise și vor emite o eroare de timp de compilare.

Parametrii funcției **Scope**, **return ref** și **return scope** sunt utilizați pentru a se asigura că parametrii nu își ies din domeniul de aplicare, nu își depășesc durata de viață a parametrilor de potrivire și sunt urmăriți corect chiar și prin indicații de indicator.

2.1.3 Evaluare

Pentru a valida abordarea noastră, arătăm că: 1) codul D are exact același comportament ca și codul C pe care îl înlocuiește, 2) mecanismele de siguranță introduse cu succes previn apariția erorilor de corupție a memoriei și 3) performanța software-ului de înlocuire. nu se degradează față de predecesorul său. Am creat o configurație în care oferim ambele implementări ale driverului *virtio_net* (C și D) și am rulat scenarii similare pentru a compara funcționalitatea, siguranța și performanța.

Corectitatea funcțională. Am rulat instrumente de rețea în fiecare mașină virtuală pentru a verifica paritatea funcționalității. De exemplu, folosind `ping` pentru a valida funcționalitatea, folosind `wget` pentru a descărca informații de pe Internet. În plus, verificăm dacă fișierul transferat este cel corect comparând hash-ul MD5 cu cel așteptat.

Siguranță. Pentru a spori siguranța codului de driver portat, am modificat codul pentru a folosi mai multe caracteristici ale limbajului D: verificarea limitelor matricei, funcții și șabloane `@safe`.

Din numărul total de accesări ale matricei din driverul *virtio_net*, am putut activa verificarea limitelor matricei în 88,4% din cazuri. Restul de 11,6% reprezintă accesări la matrice dinamice care au fost alocate în afara driverului portat. Pentru a testa efectul adăugării verificării limitelor matricei asupra driverului, am adăugat acces artificial în afara limitelor la cod. În 60% din cazuri, versiunea C a driverului a terminat execuția cu grație, în timp ce versiunea D a încetat cu o panică a nucleului în 100% din cazuri.

Funcții `@safe`. Pentru a permite compilatorului D să verifice siguranța codului, am urmărit să adnotăm toate funcțiile prezente în driver cu cuvântul cheie *@safe*. 19% dintre funcții s-au compilat cu succes fără nicio modificare, în timp ce 81,2% au eșuat compilarea din cauza efectuării unor operațiuni nesigure. Cele mai multe dintre aceste funcții se bazează pe operațiuni cu pointer și transformări care sunt interzise în codul *@safe*. Sunt necesare modificări suplimentare pentru a aduce codul într-o stare *@safe*, totuși, acest lucru se poate face treptat după portul inițial al driverului.

Tabela 2.1: Performanță comparativă

		C	D	Slowdown
TCP (Gbps)	vm-to-vm	2.662	2.642	0.88%
	vm-to-host	2.447	2.502	-2.24%
	vm-to-remote	0.934	0.935	-0.1%
UDP (pkts)	vm-to-vm	87677	85285	2.72%
	vm-to-host	151873	152253	-0.25 %
	vm-to-remote	135606	135698	-0.06%

Șabloane (templates). Codul D poate folosi funcții șablon care sunt instanțiate în timpul compilării cu tipul potrivit. În cazul unei nepotriviri de tip, aceasta va duce la o eroare de compilare, făcând astfel imposibilă existența erorilor de corupție a memoriei de rulare. Folosind funcții șablon, am înlocuit 56% din numărul total de utilizări de pointer *void*. Restul de 44% nu a putut fi înlocuit, deoarece nu a existat un model de conversie pe care l-am putea detecta și folosi pentru transformarea noastră.

Performanță

Pentru a evalua performanța, am folosit instrumentul *iperf3*, care trimite pachete înainte și înapoi într-o comunicare client-server. Am folosit o instanță de mașină virtuală care rulează versiunea C originală a driverului *virtio_net* și o mașină virtuală care rulează versiunea D. Fiecare VM a primit 1 GB de RAM și 1 CPU. *iperf3* a fost implementat pe ambele VM.

Am conceput 3 setări: 1) **vm-to-vm** Un VM rulează serverul, o VM rulează clientul. Ambele mașini sunt de același tip: fie C, fie D; 2) **vm-to-host** Gazda rulează serverul, VM rulează clientul; și 3) **vm-to-remote** Un alt sistem din rețeaua gazdă rulează serverul, VM rulează clientul.

Rezultatele sunt rezumate în Tabel **??** și arată o suprasarcină neglijabilă pentru implementarea modulului D în comparație cu implementarea C. Având în vedere că anumite părți ale măsurătorilor arată o încetinire negativă, considerăm performanța similară și supusă variațiilor de rețea și de măsurare.

2.1.4 Observații notabile

Am investigat posibilitatea utilizării D, un limbaj de programare a sistemelor cu siguranță pentru memorie, pentru a dezvolta module de kernel Linux pentru a crește robustețea și securitatea generală a sistemului.

Am dezvoltat o metodologie și am aplicat-o pentru a porta driverul de rețea `virtio_net` la

D, ca Proof-of-Concept. Am demonstrat corectitudinea funcțională și paritatea de performanță a driverului nostru portat la implementarea C originală și am evidențiat beneficiile suplimentare de securitate.

Este important de reținut că nesiguranța în interiorul nucleului este o realitate. Deși se poate folosi un limbaj de programare care folosește mecanici diferite care măresc siguranța codului pe care îl scrie un dezvoltator, la un moment dat dezvoltatorul va fi obligat să efectueze acțiuni nesigure. Acestea pot proveni din nevoia de a interacționa cu pini specifici de pe hardware-ul de bază sau din nevoia de a interacționa cu API-ul kernelului. Majoritatea nucleului API-ului kernel funcționează cu pointeri bruti, ca atare, chiar dacă codul sigur ar putea implementa un algoritm de durată de viață a obiectului sunet, fiind forțat să treacă pointerul brut către nucleu va anula toate pariurile și presupunerile de siguranță. În ciuda acestui fapt, credem că există două argumente puternice care permit utilizarea în practică a limbajelor sigure: 1) nucleul nucleului este extrem de stabil și robust, deoarece beneficiază de 30 de ani de dezvoltare și remedieri de erori și 2) nucleul API definește clar a cui responsabilitate, a nucleului sau a driverului, este de a elibera resursele alocate.

2.2 Integrarea automată a codului D cu kernelul Linux

Limbajul de programare D oferă caracteristici încorporate pentru siguranța memoriei verificate de compilator și concepte de programare la nivel înalt, cum ar fi concepte de programare orientată pe obiecte, metaprogramare și programare funcțională. De asemenea, are suport nativ pentru interoperabilitatea cu C. Cu toate acestea, așa cum se vede în Secțiunea 2.1, pentru a utiliza D în nucleu, trebuie să traduceți manual fișierele de antet C necesare în fișiere de antet D, ceea ce poate fi un proces obositor și consumator de timp. . Deși DPP este un instrument care poate automatiza această sarcină pentru fișierele antet C din spațiul utilizatorului, eșuează atunci când lucrează cu codul complex și uneori complicat al nucleului.

În acest capitol, ne extindem munca anterioară și DPP îmbunătățit pentru a permite traducerea antetelor nucleului în D. Acest lucru ușurează utilizarea codului D în nucleul Linux, oferind un mijloc de îmbunătățire a siguranței memoriei nucleului.

2.2.1 Introducere

După cum sa discutat anterior, credem că șoferii își pot îmbunătăți considerabil securitatea utilizând un limbaj de programare modern care a fost conceput pentru siguranța memoriei, cum ar fi limbajul de programare *D* [26]. *D* poate efectua automat verificări de siguranță (de exemplu, verificarea automată a limitelor matricei, evitând utilizarea după liber, dublu gratuit, evitați aritmetica pointerului etc.), compilați la cod și interfață nativ rapid și eficient cu relativă uşurinţă cu codul *C*. În acest scop, este necesar să se declare antetul funcției și aspectul (de exemplu, declarația `struct`) a tipurilor care vor fi utilizate, la fel cum se face în interiorul fișierelor de antet *C*. Având în vedere natura repetitivă și sincer plictisitoare a sarcinii, *DPP* [27] a fost dezvoltat în comunitatea *D* pentru a ajuta dezvoltatorii să refo-

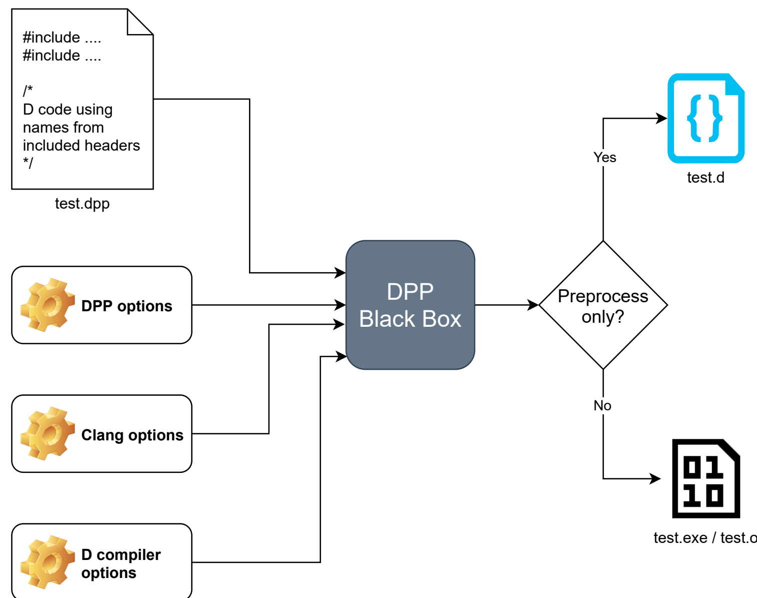


Figura 2.2: Flux de lucru DPP: (1) Scrieți un modul D normal, dar și *#include* anteturile C dorite, (2) Configurare (adică, opțiunile clang sunt utilizate pentru anteturile C; opțiunile compilatorului D sunt eliminate dacă DPP este utilizat numai pentru preprocesare, astfel încât pentru generarea unui modul D valid), (3) Rulați instrumentul, (4) Ieșirea depinde pe opțiunile DPP a fost rulat cu: modul D gata de compilat, fișier obiect sau fișier executabil.

losească codul *C* din modulele lor *D*. In orice caz, DPP nu funcționează bine atunci când este utilizat cu fișierele antet kernel-ului Linux din cauza constructelor magice din interior miezul. În această lucrare, îmbunătățim DPP pentru a traduce cu succes astfel de fișiere de antet și pentru a facilita integrarea D, un limbaj de programare sigur pentru memorie în kernel-ul Linux.

2.2.2 DPP

DPP împachetează compilatorul D pentru a acceptă sintaxa *#include* în interiorul unui modul D. Cu alte cuvinte, DPP este un preprocesor pentru limbajul D, generând legături în interior modulul de implementare D în sine. DPP are multe povești de succes, inclusiv traduceri de *curl.h*, *Python.h*, *stdio.h*, *stdlib.h* și *pthread.h*. Cu toate acestea, DPP se luptă cu anteturile Linux, în special:

- Traduceri incorecte ale mai multor macrocomenzi, funcții și structuri.
- Numele coliziunilor și corelațiile nu sunt tratate corect și consecvent.
- Se blochează la testarea cu antete specifice (adică indexarea în afara limitelor).
- Consum anormal de memorie (sunt necesar mai mult de 8 GB de RAM).
- Timpi lungi de execuție (anteturi multiple ar putea dura chiar și mai mult de 30 de minute).

Un flux de lucru tipic atunci când utilizați instrumentul *DPP* este descris în Figura 2.2. Cutia neagră DPP este compusă din patru componente diferite, așa cum se arată în Figura 2.3. Fiecare componentă are propriul său rol definit, unele descompunându-se în module granulare suplimentare (de exemplu, modulele *Runtime* și *Translation*). Rețineți că, în acest proiect, există trei concepte diferite care rotesc compilatorul Clang: (1) biblioteca *libclang*, care oferă o interfață *C* pentru analiza codului sursă, (2) *libclang* Biblioteca de legături D, care include, de asemenea, biblioteca *libclang* și (3) componenta *Clang*, care include o mică parte din biblioteca de legături *libclang* D. Arhitectura DPP este evidențiată în Figura 2.3.

Figura 2.4 arată modul în care componentele DPP se interconectează, schimbul de informații dintre componente și fluxul de lucru intern general al aplicației. Mai jos, descriem fiecare componentă.

2.2.3 Implementare

App Component. Modulul *App* este punctul de intrare al *DPP*. Acesta trebuie să fie furnizat cu un obiect de tip *Opțiuni*, care poartă opțiunile liniei de comandă utilizate pentru *DPP*. Pe baza opțiunilor furnizate, este instanțiat un obiect *Context* care este utilizat pentru a urmări cursoarele *AST* deja procesate. Apoi, fișierul *.dpp* este preprocesat de componenta *Expansion*. La rândul său, acesta din urmă extinde toate directivele *#include* în linie, traduce toate definițiile și redefinește orice macrocomandă definită în acestea. În cele din urmă, componenta *App* gestionează operațiunile de compilare și fișier *I/O*: creează fișierul de ieșire *.d*, scrie liniile de cod traduse în acesta, rulează *C* preprocesor, rulează compilatorul *D*.

Context Component. Modulul *Context* încapsulează informațiile necesare pentru traducerea curentă, evitând declararea variabile globale. Rezolvă ciocnirile și coliziunile de nume, declară structuri necunoscute (adică atunci când un antet folosește doar un pointer către un tip, fără a oferi o definiție pentru acel tip). În plus, ține evidența: *cursoare AST* vizitate, liniile de ieșire traduse până acum, informații privind ciocnirea numelor, coliziunile și mapările de redenumire, toate macrocomenzile definite, agregatele și câmpurile acestora și alte informații specifice pentru *C++* cod. Un *Context* este instanțiat o singură dată de către modulul *App* al *Runtime*, înainte de a începe procesul de traducere.

Options Component. Modulul *Opțiuni* conține toate opțiunile din linia de comandă transmise de utilizator: opțiunile *DPP*, opțiunile *Clang* (transmise intern la *libclang*) și opțiunile compilatorului *D* . Poate opri execuția *DPP* după generarea legărilor, dar înainte de a compila fișierul sursă rezultat (adică folosind opțiunea de preprocesare a *DPP*). Această componentă este utilizată de modulul *App* direct și de componenta *Expansion*, indirect prin *Context*.

Expansion Component. Componenta *Expansion* este responsabilă pentru extinderea (de exemplu, analizarea) directivelor *#include* în linie, populând instanța *Context* cu definițiile *D* rezultate. Folosește legăturile personalizate *libclang D* pentru a începe analiza antetelor *C*, oferind opțiunile *clang* din instanța *Opțiuni* din *Context*. Rezultatul analizei este un *LLVM AST*. Componenta *Expansion* rezolvă declarațiile multiple și problemele de definiție

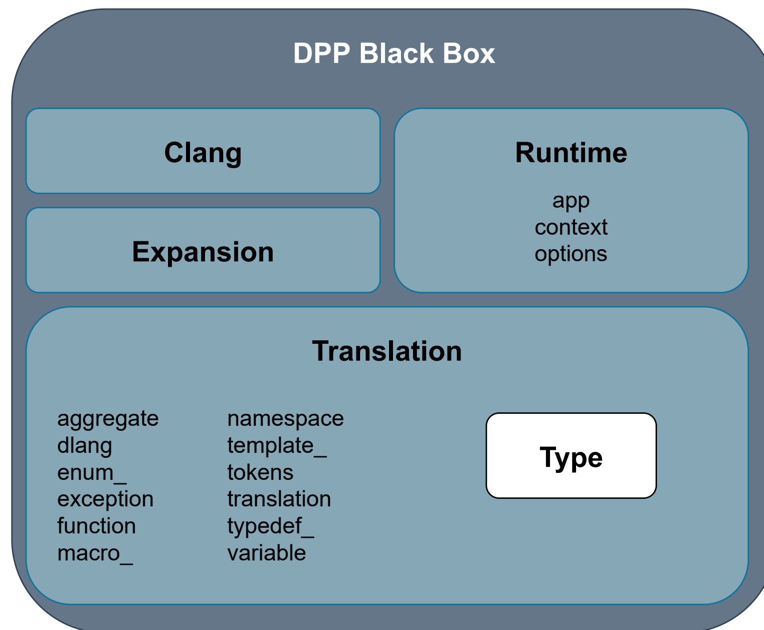


Figura 2.3: Organizarea DPP

tentativă ale *C*, prin modificarea cursoarelor *AST*. *D* nu acceptă mai multe declarații și o definiție a unui tip (adică acceptă fie o declarație, fie o definiție, dar nu ambele în același timp).

O altă funcție a acestei componente este să itereze peste toate cursoarele din *AST* și să folosească componenta *Translation* pentru a le traduce în cod valid *D*.

Translation Component. Scopul acestei componente este de a traduce fiecare construct *C* întâlnit în timp ce ex- deplasarea antetelor în fișierul de intrare original *.dpp*. Această componentă are doisprezece componente module:

- *Aggregate* este folosit pentru a traduce *struct*-uri, *union*-uri, *class*-uri, înregistrări anonime și *enum*-uri.
- *Dlang* se ocupă de traduceri specifice *D* (de exemplu, evitarea folosirii cuvintelor cheie *D* ca nume de agregate).
- *Enum* traduce numai *enum* care stochează constante cu o singură valoare.
- *Excepție* este un modul intern care definește o excepție aruncată atunci când se utilizează concepte *C++* fără echivalent cu *D*.
- *Funcția* este folosit pentru a traduce declarații și definiții de funcții, precum și metode de clasă, constructori, destructori, constructori de mutare etc.
- *Macro* este folosit pentru a gestiona macrocomenzi și definiția lor respectivă (adică fragmente de cod, reprezentări șiruri în hex, octal).
- *Namspace* și *Template* gestionează conceptele specifice *C++* cu același nume.

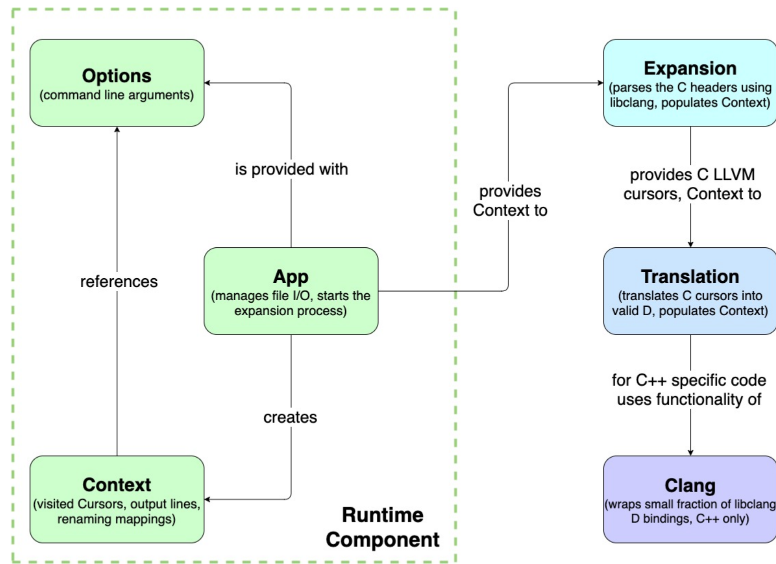


Figura 2.4: Fluxul de lucru al componentelor interne DPP

- *Tokens* este un modul intern care oferă funcții utilitare pentru a ajuta la gestionarea jetoanelor din interiorul cursoarelor.
- *Translation* este folosit pentru traducerea cursoarelor C AST, furnizate de componenta Expansion.
- *Typedef* și *Variable* traduc constructele typedef și, respectiv, declarațiile de variabile.

Componenta *Translation* conține, de asemenea, o subcomponentă numită *Type*, care este folosită pentru traduce/mapează tipurile C/C++ primitive și încorporate în tipurile D corespunzătoare.

2.2.4 Evaluare

Performanță

În starea sa anterioară, DPP nu a putut să traducă parțial niciunul dintre anteturile nucleului Linux din cauza multiplelor erori și a constructelor de sintaxă netratate. În plus, DPP este relativ ridicat consumul de memorie a împiedicat traducerea mai multor anteturi în același timp.

În prezent, DPP poate traduce cu succes un modul care conține toate anteturile Linux C necesar pentru portarea *virtio_net.c*, listat în Listing 2.1. Traducerea rezultată fișierul de ieșire are 150.000 de linii de cod și nicio eroare de sintaxă, nu necesită intervenții manuale și este gata de compilare.

```
1 #include <linux/device.h>
2 #include <linux/mod_devicetable.h>
3 #include <linux/workqueue.h>
4 #include <linux/netdevice.h>
5 #include <linux/filter.h>
```

```
6 #include <linux/socket.h>
7 #include <linux/bpf.h>
8 #include <linux/virtio.h>
9 #include <linux/virtio_config.h>
10 #include <linux/virtio_net.h>
11 #include <linux/cpuhotplug.h>
12 #include <linux/netdev_features.h>
13 #include <linux/cpu.h>
```

Listing 2.1: C Linux headers necessary for porting virtionet.c

După cum se arată în Listing 2.2, traducerea antetelor din Listing 2.1 a durat 17 minute și a folosit 4,5 GB de memorie. Acest lucru este foarte încurajator pentru viitorul DPP, având în vedere acest lucru anterior, s-a luptat să ofere cifre rezonabile de rulare: pentru câteva antete, DPP obișnuia să dureze mai mult de 30-40 de minute și, în cele din urmă, rămânea fără memorie. Înainte de modificările noastre, DPP folosea 6 până la 7 GB de memorie numai pentru antetul *netdevice.h*.

```
1 User time (seconds):: 992.60
2 System time (seconds): 30.63
3 Percent of CPU this job got: 100%
4 Elapsed (wall clock) time (h:mm:ss or m:ss): 17:03.07
5 [...]
6 Maximum resident set size (kbytes): 4780252
7 [...]
```

Listing 2.2: /usr/bin/time output

Productivitate

Cea mai importantă măsurătoare pe care DPP își propune să o îmbunătățească este productivitatea programatorilor D care trebuie să se interfațeze cu bibliotecile C. Identificăm două aspecte ale productivității unui inginer software: timpul de dezvoltare și timpul de întreținere.

Știm din munca noastră anterioară că ne-a luat câteva luni pentru a traduce manual **doar** structurile necesare și declarațiile de funcții ale nucleului. În cele din urmă, baza noastră de cod a avut 8284 de linii de cod (LoC) și se întinde pe 27 de fișiere. Prin contrast, folosind DPP, timpul de traducere a scăzut la 17 minute, iar baza de cod a fost redusă la 3821 LoC și doar 3 fișiere.

Folosind DPP, timpul de traducere este cu ordine de mărime mai mic, măsurabil în minute, mai degrabă decât în luni. Acest lucru este foarte valoros, deoarece le permite dezvoltatorilor să-și investească timpul în logica reală de dezvoltare, mai degrabă decât pe sarcina repetitivă (și predispusă la erori) de a defini manual interfețele. Traducerile manuale sunt în mod implicit mai predispuse la erori și dificil de rezolvat decât traducerile furnizate de un generator automat. O eroare într-o traducere manuală ar putea însemna realizarea modificări în mai multe părți ale codului, în timp ce rezolvarea unei erori într-un generator se rezolvă automat problemele din toate părțile codului.

2.2.5 Observații notabile

Scopul proiectului a fost de a face DPP robust și capabil să traducă chiar și cele mai ciudate constructe și sintaxe C, ducând în cele din urmă la traducerea cu succes a Antetele nucleului Linux sunt necesare pentru a construi driverul *virtio_net*. Strategia a fost testarea progresivă a antetelor nucleului, identificarea și investigarea problemelor, identificarea tiparelor problemelor și furnizarea unei soluții generice care le rezolvă.

Am rezolvat probleme, de la lipsa agregatelor anonime C11 gestionate, ciocniri de nume, incoerențe de redenumire și a adăugat funcționalitate nouă instrumentului în sine. Unul dintre blocanții finali care au făcut DPP inutilizabil în nucleul Linux a fost consumul excesiv de memorie. Aceasta nu mai este o problemă și am demonstrat că DPP lucrează cu resurse decente disponibile.

Așa cum este, DPP poate genera legături pentru toate funcțiile, structurile, enumerarile etc. utilizate de *virtio_net*. Acest rezultat este extrem de important și relevant în domeniul programării sistemelor, în special pentru construirea de drivere sau module de sisteme de operare într-un mod inerent sigur.

Capitolul 3

Îmbunătățirea securității microserviciilor și aplicațiilor

3.1 Un nou framework de colecții pentru biblioteca standard a limbajului de programare D

D este o programare de uz general, de nivel înalt și de înaltă performanță limbaj capabil să interfațeze cu *API-ul sistemului de operare* și *hardware-ul sistemului*. Una dintre caracteristicile de bază ale lui D este reprezentată de intervale, o abordare puternică și nouă de iterare printr-un set de elemente. Cu toate acestea, fiind o caracteristică de ultimă generație, intervalele nu sunt utilizate în ecosistemul D la potenţialul lor maxim.

În această lucrare oferim un *noi cadru de colecții pentru biblioteca D Standard* care este compatibilă cu algoritmii existenți și care oferă un API similar cu cel oferit de intervale. Implementarea noastră permite colecțiilor să deducă siguranța operațiunilor din tipul conținut, furnizat de utilizator.

Arătăm că implementarea noastră poate oferi beneficii de performanță de până la 2x în comparație cu implementările standard existente ale bibliotecii, atunci când sunt utilizate împreună cu un alocator personalizat.

3.1.1 Introducere

Limbajul de programare D [28] își propune să ofere un mod rapid și eficient de a scrie corect, programe rapide și ușor de întreținut. D implementează funcții moderne și puternice, cum ar fi siguranța memoriei, funcția puritate și lizibilitate îmbunătățită a codului, pentru a satisface nevoile industriei de inginerie software în continuă creștere.

Intervalele, așa cum vom detalia în capitolul următor, sunt modul D de iterare printr-un set de elemente. O gamă este capabilă să efectueze următoarele operații: accesarea elementelor, testarea golului și modificarea elementelor. Gamele oferă un adaptor excelent de acces pentru algoritmii din biblioteca standard. Acest lucru se datorează faptului că a avea o interfață comună simplifică utilizarea structurilor de date și permite utilizatorului să aibă așteptări rezonabile cu privire la colecții.

În prezent, biblioteca standard D oferă utilizatorilor un set de colecții de utilizat, dar implementarea acestora precede unele dintre caracteristicile existente ale limbajului. Colecțiile

existente folosesc Garbage Collector [29] încorporat, presupunem că tipul de utilizator subiacent este nesigur și nu funcționează bine cu calificatorii de tip *const* și *immutable* ai limbajului.

În această lucrare, propunem un set de colecții de uz general (vector, listă, hartă, hash-tables, heaps etc.) care sunt rapide, fiabile și compatibile cu algoritmii existenți, valorificând în același timp funcțiile puternice ale limba. Colecțiile permit utilizatorului să aleagă schema de alocare dorită, prin furnizarea unui alocator personalizat pentru a fi utilizat de structura de date. Ei nu fac presupuneri despre tipul de utilizator subiacent, ci ei deduceți siguranța și puritatea acestuia, totul în timp ce puteți lucra într-un mediu imuabil. Rezultatul final al muncii noastre este o bibliotecă de colecții care a fost integrată în biblioteca standard D care este mai eficientă, mai flexibilă și mai sigură în ceea ce privește utilizarea memoriei decât soluțiile alternative existente.

3.1.2 Iteratori

Un iterator reprezintă o modalitate de a oferi acces la elementele de un container. Pentru că pointerii reprezintă abstractizarea fundamentală model folosit în STL [30], nu există multe cazuri de utilizare pentru un singur iterator; aveți nevoie atât de **begin**început cât și de **end**iteratorii containerului pentru a putea fi în siguranță traversează-l. Această abordare suferă de două neajunsuri:

- Când lucrează cu iteratoare, utilizatorul trebuie să fie atent cu împerecherea dintre **begin** și **end** iteratoare. Asocierea greșită a două iteratoare este atât o greșeală frecventă, cât și o eroare greu de descoperit.
- Atunci când implementează algoritmi, un utilizator, în esență, are nevoie pentru a furniza ambii iteratori pentru a defini **intervalul** a recipientului său. Acest lucru duce la un cod mai greoi, predispus la erori și mai puțin întreținut.

Intervalele reprezintă o alternativă la iteratoarele care oferă toate beneficii, dar nu suferi de niciunul dintre neajunsuri. Urmatorul secțiunea va prezenta în detaliu cum sunt definite și cum sunt acestea îmbunătățirea iteratoarelor existente.

3.1.3 Range-uri în D

Range-urile sunt o abstractizare a accesului la elemente și o parte de bază de D. Ele oferă o nouă abordare a problemei accesării la elementele unui container. Gamele au fost introduse de Andrei Alexandrescu în articolul **On Iteration** [31], subliniind punctele slabe ale *iterator design*, folosit de biblioteca de șabloane standard C++ (STL) și demonstrând avantajele *proiectarea gamei*.

Designul gamei care a fost prezentat de Alexandrescu a fost implementat în limbajul de programare D. Pentru simplitate, vom face prezintă conceptul de intervale în urma implementării D. Notă că alte implementări pot fi ușor diferite, totuși conceptul de bază rămâne același.

În D, orice structură care oferă acces la elementele unui container, pentru a fi considerate o gamă, trebuie să implementeze trei metode:

- **empty()** - confirmă dacă intervalul este gol sau nu.
- **front()** - oferă acces la primul element al gamei.
- **popFront()** - elimină primul element al gamă, micșorându-și dimensiunea cu una.

Aceste trei operațiuni definesc tipul de interval de bază, the **InputRange**. Intervalul de intrare abstrage secvenţialul iterația unui container și se adaptează bine la fluxurile de date, cum ar fi citirea de la intrarea standard.

Unele containere trebuie repetate de mai multe ori. Un astfel de scenariu face ca InputRange să nu fie adecvat. ForwardRange adaugă la interfața InputRange Operația **save()**, care returnează o copie a intervalului.

Pentru situațiile în care este necesară o traversare inversă, **BidirectionalRange** adaugă încă două operații la Interfață ForwardRange:

- **back()** - oferă acces la ultimul element al intervalului
- **popBack()** - elimină ultimul element al intervalului

În cele din urmă, cea mai puternică gamă care există, cea **RandomAccessRange** furnizează **opIndex(size_t i)** operator, care oferă acces la al-lea element **i** al containerului.

Range-uri vs Iteratori

Intervalele din limbajul de programare D sunt în general considerate a fi o modalitate mai convenabilă și mai expresivă de a itera colecții în comparație cu iteratoarele C++. Unele dintre motivele pentru aceasta includ:

- Intervalele sunt mai flexibile: Intervalele pot fi compuse și transformate cu ușurință folosind o varietate de primitive de interval, cum ar fi hartă, filtru și pliere, ceea ce facilitează efectuarea de operațiuni complexe asupra colecțiilor.
- Intervalele sunt mai expresive: intervalele furnizează o sintaxă mai intuitivă și mai lizibilă pentru iterarea colecțiilor, ceea ce poate facilita înțelegerea intenției codului.
- Intervalele sunt mai eficiente: intervalele sunt implementate într-un mod care le permite să fie folosite leneș și să evite alocările inutile de memorie, ceea ce le poate face mai eficiente decât iteratoarele C++ în unele cazuri.
- Intervalele sunt mai sigure: Intervalele sunt concepute pentru a fi mai sigur de utilizat decât iteratoarele C++, deoarece oferă verificarea limitelor și gestionează automat iterarea peste sub-intervaluri.

În general, intervalele oferă o modalitate mai convenabilă, expresivă, eficientă și mai sigură de a itera colecțiile în limbajul de programare D, motiv pentru care sunt în general preferate

față de iteratoarele C++. Intervalele au fost, de asemenea, adăugate la biblioteca standard C++, începând cu C++20.

3.1.4 Implementare

Viteză. Ne dorim ca cadrul colecțiilor noastre să fie cât mai rapid posibil, având în vedere constrângerile utilizatorilor. Pentru a atinge acest obiectiv urmărim durata de viață a unei colecții folosind numărarea referințelor și permite utilizatorului să utilizeze alocatoare personalizate [32].

Siguranța memoriei. Sistemele de tip garantează acel cod adnotat cu **@safe** nu provoacă coruperea memoriei. Cu toate acestea, o colecție va interacționa cu tipuri definite de utilizator care pot apela funcții nesigure. În consecință, indiferent cât de sigur este codul de colectare, siguranța întregului testament fi determinat de siguranța tipului conținut.

Pentru a obține siguranța, trebuie luate în considerare următoarele două observații: (1) Alocarea memoriei este o operațiune sigură. Nu ar trebui să fie nimic nesigur aici. Cerem doar alocatorului o bucată de memorie, pe care o va oferi dacă mai are ceva. (2) Dealocarea este o operațiune nesigură din punctul de vedere al alocătorului, deoarece acesta nu poate ști că au mai rămas referințe la memoria tampon pe care urmează să-l elibereze. Dealocările făcute de colecții, pe de altă parte, sunt sigure, deoarece numărarea referințelor oferă garanția că nu mai există referințe la buffer în momentul eliberării acestuia.

Puritatea funcțională. Conceptul de funcții *pure* provine din programarea funcțională [33]. O funcție pură are două principalele caracteristici care sunt detaliate mai jos. În primul rând, o funcție pură va produce același rezultat pentru același set de parametri. Ca rezultat, rezultatul unei funcții pure poate fi stocat în cache și utilizat pentru a elimina apelurile ulterioare ale aceleiași funcție cu aceiași parametri. În al doilea rând, o funcție pură nu are voie să acceseze date globale mutabile. În consecință, funcțiile pure nu au efecte secundare. Astfel de funcții pot fi dovedite în mod oficial a fi corecte sau nu. Puritatea ajută utilizatorul să raționeze cu privire la logica codului și reprezintă un instrument puternic de optimizare pentru compilator.

Stil funcțional. D furnizează calificatorii de tip *const* și *imutable*, imuabilitatea fiind o parte esențială a paradigmei de programare funcțională. Înainte de munca noastră, modulul container nu a acceptat utilizarea unor astfel de calificative.

Noile colecții, pe de altă parte, susțin astfel de calificative. Acest lucru permite implementarea algoritmilor multithreading și utilizarea expresiilor de stil funcționale.

Proprietățile colecției

Colecțiile definite în biblioteca noastră au următoarele proprietăți:

- Furnizați cel puțin următoarele metode: *empty()*, *front()*, *popFront()*, *save()*.
- Sunt utilizabile în contexte *safe*, *pure*, *nogc*, *nothrow*.

- Sunt utilizabile împreună cu calificatorii de tip tranzitiv ai lui D: *const*, *immutable* și *shared*
- Sunt utilizabile cu algoritmii existenți care funcționează pe intervale
- Sunt optime în ceea ce privește performanța, cu condiția ca constrângerile utilizatorului să fie îndeplinite.

Noul cadru de colecții permite alegerea bazată pe constrângeri: utilizatorul poate solicita o colecție care satisface anumite constrângeri [34]. De exemplu: dacă utilizatorul dorește o colecție care are o mapare cheie - valoare cu timp constant de inserare și recuperare, i se va furniza un hashtable. Pentru a realiza acest lucru, implementarea internă a colecțiilor folosește puternica introspecție în timp de compilare a lui D.

3.1.5 Evaluare

Pentru a ne evalua munca, am creat un punct de referință sintetic, adică concepute pentru a evalua implicațiile de performanță ale implementării noastre.

```
1  auto getSList(size_t steps, RCIAllocator alloc)
2  {
3      SList!size_t a = SList!size_t(alloc);
4      for (size_t i = 0; i < steps; ++i) {
5          a.insert(i);
6      }
7      return a;
8  }
9
10 void benchmark(size_t steps, RCIAllocator alloc)
11 {
12     for (size_t i = 0; i < 10; ++i) {
13         auto a = getSList(steps, alloc);
14         auto b = getSList(steps, alloc);
15         auto c = a ~ b;
16     }
17 }
```

Listing 3.1: Exemplu benchmark

Listarea 3.1 evidențiază o prezentare generală la nivel înalt a codului de evaluare comparativă. Creăm două liste conectate individual folosind un alocător personalizat. Inserăm o serie de elemente (de la 10K la 40M) pentru a analiza implicațiile utilizării diferiților alocatori.

Figura 3.1 prezintă rezultatele rulării benchmark-ului nostru în 3 scenarii separate: (1) folosind implementarea bibliotecii standard a unei liste cu legături unice care utilizează colectorul de gunoi, implementarea listei noastre cu legături unice (etichetată ca exp) folosind atât (2) colectorul de gunoi, cât și (3) alocatorul bazat pe malloc. Observațiile noastre arată că, pe măsură ce numărul de elemente alocate crește, implementarea noastră care

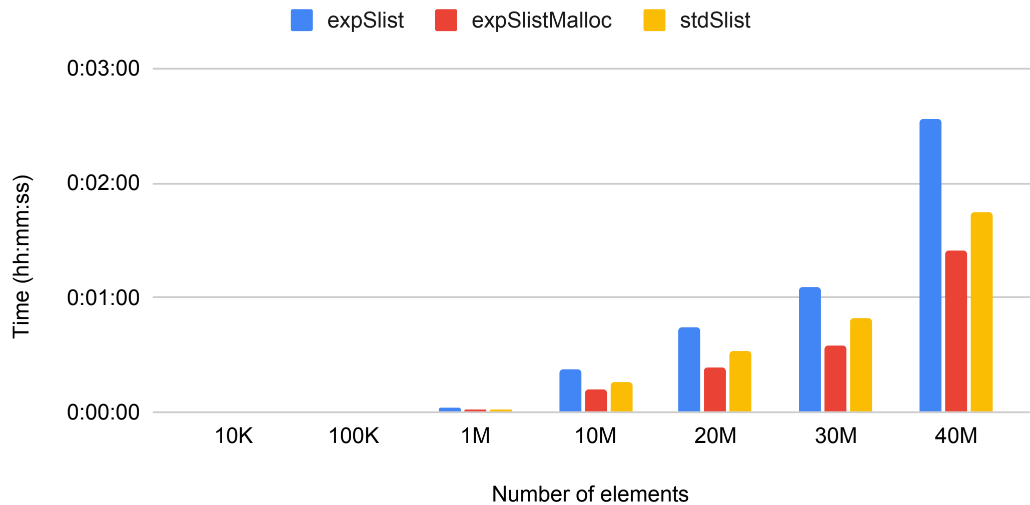


Figura 3.1: **Evaluarea performanței listei cu legături unice folosind diferiți alocători**: *stdSlist* este implementarea Bibliotecii standard care utilizează GC; *expSlist* este implementarea cadrului experimental care folosește GC; *expSlistMalloc* este implementarea cadrului experimental care folosește MAllocator (alocator bazat pe malloc). *expSlistMalloc* arată beneficiile utilizării unui alocator personalizat

utilizează malloc este cea mai rapidă (cu un factor de accelerare maxim de 2, având în vedere configurația noastră experimentală). Diferența se explică prin faptul că malloc este o alternativă mai ușoară la colectorul de gunoi. Ceea ce este surprinzător, totuși, este că implementarea bibliotecii standard este mai rapidă decât implementarea experimentală a listei cu legături unice care utilizează colectorul de gunoi. Explicația constă în faptul că, în primul caz, colectorul de gunoi are libertatea de a decide când memoria este alocată și eliberată și, prin urmare, își poate aplica în mod optim strategia de alocare a memoriei, în timp ce, în al doilea scenariu, sunt inserate apeluri către colectorul de gunoi. de cadrul de colectare. Acest lucru forțează colectorul de gunoi să acționeze la comandă, fără a ține cont de metricile sale formale.

Rezultatele noastre demonstrează că implementarea listei noastre cu legături unice poate fi utilizată împreună cu un alocător personalizat pentru a obține performanțe mai bune decât implementarea standard colectată de gunoi.

3.1.6 Observații notabile

D este un limbaj în creștere care trebuie să ofere o suită puternică de colecții. Folosind beneficiile oferite de aderarea la interfața intervalelor, colecțiile se vor potrivi perfect cu algoritmii din biblioteca standard fără niciun efort suplimentar.

Am dezvoltat un nou cadru de colecții care reunește toate caracteristicile importante ale limbajului D, fiind în același timp ușor de utilizat și intuitiv pentru utilizator.

Această abordare nouă de proiectare a colecțiilor ca game cu primitive opționale s-a dovedit că oferă performanțe mai bune decât alternativa standard de colectare a gunoiului. În plus, oferă flexibilitate utilizatorului pentru a alege cea mai potrivită strategie de alocare.

În plus, am actualizat cu succes modulul alocătorilor la un API mai sigur, fără a afecta performanța și am îmbunătățit atributele de funcție ale API-ului alocătorilor. Acest lucru a îmbunătățit inferența tipului de timp de compilare a sistemului. Am lansat pachete dub pentru cadrul de colecții[1] și modulul de alocare[2]; la momentul scrierii acestui articol, bibliotecile au fost descărcate de 6916 ori, respectiv de 1473901 ori.

3.2 Extinderea suportului API client-server pentru limbaje de programare sigure pentru memorie

Aplicațiile web Google au devenit o componentă integrală a viața de zi cu zi atât a organizațiilor, cât și a persoanelor deopotrivă. Aceste poate fi accesat prin interfața grafică cu utilizatorul (GUI) sau prin interfața de programare a aplicațiilor (API). Acesta din urmă este folosit în principal de programatori pentru a integra astfel de servicii în aplicatiile lor.

Majoritatea limbajelor folosite pentru implementarea unor astfel de aplicații sunt proiectate având în vedere performanța, neglijând adesea securitatea. Cu toate acestea, securitatea a devenit o preocupare majoră pentru astfel de sisteme, crescând astfel cererea pentru limbi sigure pentru memorie. Din păcate, se cunosc limbi precum D și Rust pentru a fi în siguranță pentru memorie, nu au suport pentru serviciile Google.

În acest scop, dezvoltăm o metodologie de integrare a Google servicii cu limbaje de programare sigure. Arătăm că Limbajul de programare D se poate integra cu ușurință și cu succes astfel de servicii aducând un impuls în securitate și productivitate.

3.2.1 Introducere

API-urile Google sunt un set de interfețe care oferă dezvoltatorilor programe programatice acces la serviciile Google, inclusiv Google Drive, Google Calendar, etc., oferind astfel posibilitatea de a le integra în altele aplicatii. Accesarea API-urilor Google făcând solicitări *HTTP* către serverul poate părea o metodă simplă, dar este se recomandă utilizarea bibliotecilor client care facilitează accesul serviciile din cod. Astfel, cineva nu este obligat să se ocupe de toate solicitări și erori, acestea făcând deja parte din bibliotecă implementare. Prin utilizarea bibliotecilor, cantitatea de cod standard trebuie să scrieți este redus și cu siguranță funcționează așa cum este intenționat deoarece bibliotecile sunt periodic testate și actualizate. De asemenea, bibliotecile client gestionează toate detaliile și pot fi ușor in-

[1] https://code.dlang.org/packages/collections
[2] https://code.dlang.org/packages/stdx-allocator

stalat folosind un manager de pachete, cum ar fi *pip*, *npm* sau *dub* (Manager de pachete Dlang).

Limbajul de programare D [35] este un limbaj modern, de nivel înalt și sigur limbaj de programare capabil să interfațeze direct cu API-ul și hardware-ul sistemului de operare. După cum sugerează și numele, D a fost creat ca o formă de reinginerie C++, păstrându-și eficiență și acces la nivel scăzut, câștigând în același timp siguranța memoriei și o sintaxă mai simplă [28]. De asemenea, Dlang este compatibil cu C++, adică acel cod C++ poate fi folosit în codul sursă D. Ca la toate cele moderne limbaje de programare, D are un manager de pachete numit *dub*, care facilitează adăugarea pachetelor de care aveți nevoie pentru a dezvolta o aplicarea.

Limbajul de programare D nu are biblioteci client pentru Google API-uri, făcând astfel dificilă interacțiunea dezvoltatorilor Dlang cu servicii de la Google. De asemenea, lipsa bibliotecilor poate fi ducând la o scădere a popularității, chiar dacă D are vârf Caracteristici. Pentru a interacționa cu API-urile Google, este necesar să folosiți mai multe decât o bibliotecă, dintre care unele sunt biblioteci destul de grele, sau scrie toate solicitările *HTTP* [36] necesare pentru autentificarea și accesarea resurselor, deoarece API-urile expun o interfață tradițională simplă *REST* [37]. Deși oferă mai mult control asupra codului, această abordare poate duce la multe erori sau rezultate nedorite. De exemplu, s-ar putea folosi parametri greșiți pentru un anumit apel API sau încercați să modificați o resursă numai în citire. Într-un mediu open source, este mai bine implementează biblioteci care pot fi actualizate ulterior de către comunitate, asigurându-se astfel că fiecare utilizator are un sistem complet funcțional și actualizat.

În această lucrare implementăm o bibliotecă pe partea client scrisă în limbajul de programare D, conceput pentru a fi folosit pentru a scrie programe sigure pentru memorie, de înaltă performanță, cu care interacționează servicii Google. Am deschis biblioteca noastră și am făcut-o accesibilă prin Github. Folosind biblioteca noastră, dezvoltatorii pot scrie cu ușurință și în siguranță programe fără reimplementarea aceleiași funcționalități și obținerea avantajelor utilizării unei biblioteci robuste.

3.2.2 Google APIs

O interfață de programare a aplicațiilor (API) [38] este o interfață între sisteme sau programe de computer. Totul, de la cel mai popular sistem de operare la aplicațiile bancare pe care le folosim și serviciul nostru de streaming preferat se bazează pe API-uri. În ultimii cincisprezece ani, numărul de API-uri publice a crescut exponențial. Conform ProgrammableWeb [39], din 2021, există peste 24.000 de API-uri. Evoluția telefoanelor mobile spre smartphone-uri și expansiunea web sunt câțiva dintre factorii majori care au influențat creșterea API-urilor.

API-urile web au nevoie de un format standard definit pentru datele pe care le schimbă. Javascript Object Notation[40] este un format care poate fi citit de om pentru reprezentarea datelor, bazat pe sintaxa obiectului Javascript. Poate împacheta tipuri de date primitive,

cum ar fi șiruri de caractere, valori booleene și numere, precum și matrice și obiecte, permițând API-ului să construiască o ierarhie de date [41].

Un API RESTful[42], cel mai frecvent cunoscut sub numele de API REST, este o interfață de programare a aplicațiilor (API), caracterizată prin:

- arhitectura client-server folosind cereri *HTTP*
- focus pe resurse[**?**]
- solicitări neconectate

API-urile Google reprezintă un set de API-uri publice, în principal RESTful, care le permite dezvoltatorilor să interacționeze cu produsele și serviciile Google și să le integrați în alte produse și servicii [43]. Cazurile de utilizare obișnuite pentru aceste API-uri includ:

- înregistrarea sau autentificarea utilizatorului care permite utilizatorilor să se conecteze la servicii sau aplicații terță parte folosind conturile lor Google
- stocarea documentelor din diverse instrumente, cum ar fi draw.io în Google Drive Căutare personalizată
- care permite site-urilor web să încorporeze casete de căutare, oferind astfel o metodă de căutare a site-ului respectiv cu ajutorul API-ului Custom Search.

Majoritatea API-urilor Google necesită autorizare folosind protocolul OAuth 2.0 [44][45]. OAuth este un standard deschis pentru delegarea accesului, folosit de utilizatori pentru a acorda alte aplicații accesul la informațiile lor în siguranță și fără a partaja parola. Acest standard nu se referă la autentificare, ci la autorizare. În timp ce autentificarea este funcția de specificare a drepturilor/privilegiilor de acces la resurse, autorizarea reprezintă funcția de a dovedi că cineva este cine pretinde că este.

Figura 3.2 evidențiază pașii necesari unui client accesați o resursă: autorizare, autentificare și achiziție de resurse.

Pentru ca un dezvoltator să integreze API-urile Google în alte servicii sau produse, detaliile despre API-ul trebuie să fie cunoscute pentru a interacționa cu acesta, cum ar fi cum să autorizeze solicitările, cum să structureze solicitările și răspunsurile, ce metode pot fi utilizate pentru o resursă. Google a oferit o soluție pentru această problemă sub formă de documente care descriu fiecare versiune a fiecărui API, numite documente de descoperire. Documentele de descoperire sunt fișiere în format JSON care oferă detalii despre cum să accesați API-ul. Documentele includ o descriere a resurselor și metodelor asociate, parametrii pentru metode, solicitări și structura de răspuns, precum și domeniile OAuth 2.0.

Discovery documents

Google API Discovery Service[46] este un API care oferă documente de descoperire pentru dezvoltatorii care intenționează să implementeze biblioteci client și alte instrumente

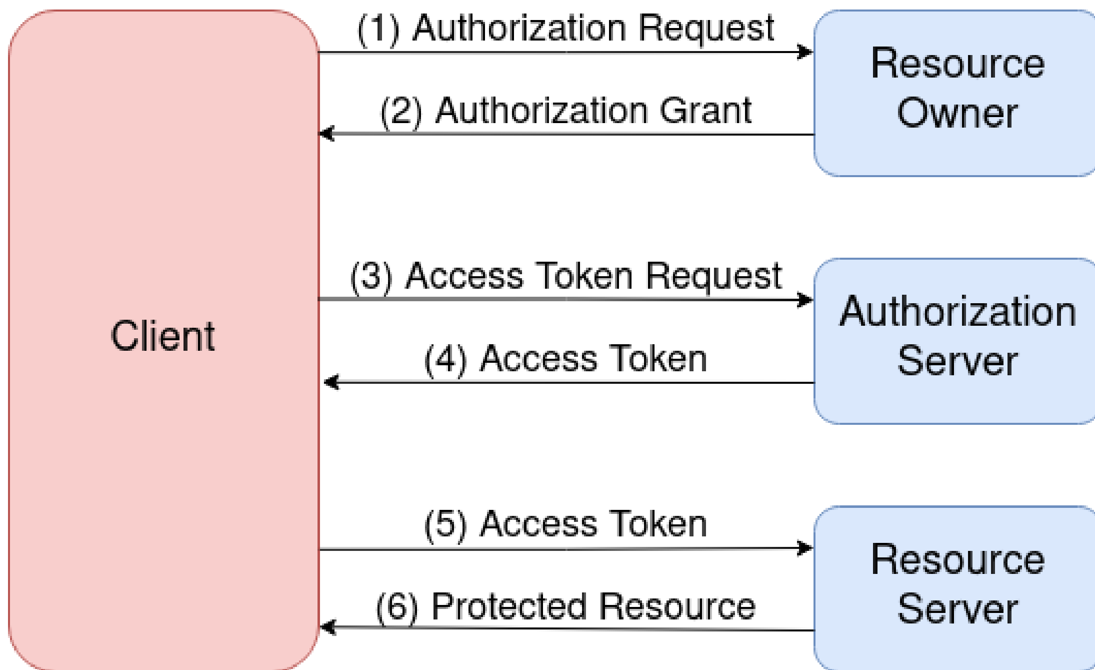


Figura 3.2: Flux OAuth 2.0: (1) clientul solicită autorizare, (2) clientul primește autorizație, (3) clientul solicită acces jeton, (4) clientul primește jeton de acces, (5) client solicită resursă, (6) clientul primește resurse

utilizate pentru a interacționa cu API-urile Google. Documentele de descoperire furnizate de aceste servicii au un format specific care include informații care sunt împărțite în șase secțiuni principale:

- Informații despre API
- Informații de autentificare - protocol și domenii
- Resurse și modele (sau scheme pentru cereri și răspunsuri)
- Metode pentru fiecare resursă
- Alte caracteristici acceptate de API
- Documentație

Listarea 3.2 afișează un eșantion de document de descoperire potențial. Conține descrierea pentru o anumită versiune, inclusiv numele API-ului, URL-ul rădăcină, calea serviciului și protocolul care este fixat la „rest”.

```
1 {
2   "rootUrl": "https://www.googleapis.com/",
3   "servicePath": "drive/v3/",
4   "version": "v3",
5   "protocol": "rest",
6   "discoveryVersion": "v1",
7   "name": "drive",
```

```
8   "ownerDomain": "google.com",
9   "ownerName": "Google"
10 }
```

Listing 3.2: Descrierea Google Drive API

3.2.3 Evaluare

Pentru a valida codul sursă generat, fiecare limbă avea un fișier test unitar[?], prin urmare am decis să scriem unul pentru D pentru a ne asigura că rezultatul este structurat așa cum ne așteptăm. Ca exemplu, Listarea 3.3 reprezintă un caz de testare simplu folosit pentru a valida calea modulului generată de acest instrument.

```
1  def testPackagePath(self):
2      discovery = {
3          'name': 'foo',
4          'version': 'v1',
5          'packagePath': 'data',
6          'schemas': {},
7          'resources': {},
8          }
9      gen = d_generator.DGenerator(discovery)
10     gen.AnnotateApiForLanguage(gen.api)
11     api = gen.api
12     self.assertEquals('Google.Apis.Data.Foo.v1', api.module.name)
13     self.assertEquals('Google.Apis.Data.Foo.v1.Data', api.model_module.
           name)
```

Listing 3.3: D support unit test

În plus, pe lângă testarea făcută în timpul procesului de dezvoltare, am scris un set de teste funcționale care fac acest proces mai ușor și mai rapid. Pentru fiecare metodă a unui API, am implementat un test care o execută și se asigură că rezultatele sunt cele așteptate. Dacă metoda care este testată modifică configurația, revine modificările la sfârșitul testului, astfel că data viitoare când testele vor rula, eșecul va fi evitat.

Procesul de testare funcțională este împărțit în două părți, și anume configurarea și testarea în sine. Configurarea constă în construirea serviciului pentru API-ul testat și obținerea id-urilor resurselor care vor fi folosite în partea următoare. De exemplu, Lista 3.4 face parte din `initUtils()` folosit în testele pentru API-ul Google Drive și preia id-ul fișierului folosit pentru a testa răspunsurile la comentarii.

```
1 /* Get Replies Test File Id */
2 auto response = Utils._drive.files()
3                 .list_()
4                 .setQ("name = 'repliesTestFile.txt'")
5                 .execute();
6
7 assert(!response.getFiles().empty);
8
```

```
9  Utils._replies = response
10                   .getFiles()[0]
11                   .getId();
```

Listing 3.4: D init tests

Testele execută cererile și apoi compară răspunsul cu răspunsul așteptat folosind expresia assert. În prezent, singurele biblioteci client testate complet sunt cele folosite pentru a interacționa cu Google Drive și Gmail, urmând să fie testate mai multe în lucrările viitoare. Lista 3.5 evidențiază modul în care un fișier din Google Drive poate fi copiat folosind implementarea noastră.

```
1  import Google.Apis.Drive.v3.Drive: Drive;
2  import Google.Apis.Drive.v3.DriveScopes: Scopes, DriveScopes;
3
4  void main()
5  {
6      Drive _drive = new Drive("credentials_file", Scopes.Drive);
7      auto result = _drive.files().copy(DUMMY_FILE_ID).execute();
8  }
```

Listing 3.5: Preluarea unei copii a unui fișier din Google Drive

3.2.4 Observații notabile

În această lucrare am adăugat suport pentru serviciile API Google în limbajul de programare D prin implementarea unei biblioteci. Am atins acest obiectiv prin utilizarea unui generator pentru astfel de biblioteci menținut de Google ca un efort voluntar. O parte substanțială a bibliotecilor generate împărtășește o structură comună cu bibliotecile client Java, ceea ce face mai ușor pentru dezvoltatorii care le-au folosit în alte limbi să le folosească în Dlang.

Comunitatea limbajului de programare D și-a exprimat interesul și sprijinul în implementarea acestor biblioteci client, dovedind astfel relevanța acestui proiect. Bibliotecile client API Google Drive[47] și Gmail[48], precum și fork-ul generatorului de biblioteci client Google cu suportul D adăugat sunt disponibile pe Github. Astfel, dezvoltatorii Dlang pot clona sau furca cu ușurință depozitele pentru a utiliza biblioteca. În plus, pentru a facilita procesul de instalare, intenționăm să oferim pachete dub pentru fiecare bibliotecă client.

Serviciile Google expun API-uri complexe, ceea ce duce la un proces de testare dificil și consumator de timp. Acesta este motivul pentru care doar două biblioteci client sunt disponibile pentru moment. Am reușit să testăm complet bibliotecile client Google Drive și Gmail. De îndată ce alte biblioteci vor fi testate, acestea vor fi disponibile pe Github și curând după, de asemenea, pe dub.

Capitolul 4

Auditarea securității arhitecturilor și aplicațiilor IoT

4.1 IoT Fuzzing folosind AGAPIA și Framework-ul River

Pe măsură ce utilizarea sistemelor Internet of Things (IoT) se extinde, crește și riscurile de securitate asociate cu conectarea dispozitivelor de la diverși furnizori. Poate fi o provocare să testați și să validați în mod eficient securitatea sistemelor IoT, ca urmare a naturii proprietare (sursă închisă) a multor software-uri și a dificultăților în colectarea datelor, cum ar fi corupțiile memoriei, din cauza naturii încorporate a sistemele. Propunem extinderea limbajului AGAPIA astfel încât să permită echipelor IoT să creeze aplicații mai sigure care pot fi auditate prin instrumente fuzzing precum RiverIoT. Acest lucru ar permite, de asemenea, utilizatorilor să-și testeze sistemele, îmbunătățind încrederea generală. Evaluăm modul în care extensiile mici la limbile existente pot ajuta investigațiile de securitate ale sistemelor IoT.

4.1.1 Introducere

Internetul lucrurilor (IoT) este un sistem de dispozitive interconectate care poate colecta date din mediul înconjurător și poate acționa asupra acestuia. IoT a crescut semnificativ în ultimii ani, ajutând oamenii să lucreze mai inteligent și îmbunătățindu-și calitatea vieții. Acum putem găsi dispozitive inteligente peste tot, de la casele noastre (becuri inteligente, camere IP), până la mașini (senzori inteligente, camere) și orașe (senzori de vreme, senzori de poluare). Odată cu dezvoltarea continuă a tehnologiilor de rețea, cum ar fi 5G, numărul de dispozitive IoT va continua să crească [5], aducând și mai multă tehnologie în viața noastră, în încercarea de a face totul mai rapid, mai inteligent și mai confortabil.

Numărul tot mai mare de dispozitive IoT (și furnizorii acestora) ridică preocupările legate de confidențialitate și securitate. Într-o lume cu ritm rapid, în care vânzătorii concurează între ei pentru a obține cel mai bun timp de comercializare pentru dezvoltarea dispozitivelor [4], există mult loc pentru erori de programare care pot duce la vulnerabilități și exploatări [6] [7]. Arhitectura tipică pentru un sistem IoT este compusă din mai multe dispozitive eterogene, nu neapărat de la același furnizor, conectate prin diferite protocoale prin Internet. Majoritatea dispozitivelor rulează firmware proprietar, cu implementări interne ale standardelor de protocol, dezvoltate într-un limbaj nesigur pentru memorie (cum ar fi limbajul de programare C), făcându-le o țintă profitabilă pentru atacatori.

Având în vedere natura sursă închisă a codului dispozitivelor IoT, este dificil pentru o terță parte (utilizator, alt furnizor etc.) să auditeze și să valideze corectitudinea implementării. Deoarece nu poate accesa codul, a treia parte trebuie să se bazeze pe testarea cutie neagră, combinând tehnicile de fuzzing cu testarea funcțională. RiverIoT [49] este un cadru open-source care permite combinarea aplicațiilor IoT end-to-end. Pentru a oferi fuzzing eficient, RiverIoT se bazează pe o specificație JSON a Intrării/Ieșirii (I/O) a dispozitivelor fuzz. Vom discuta acest lucru mai detaliat în Secțiunea 4.1.3.

După cum sa menționat anterior, limbajul de programare utilizat pentru dezvoltarea firmware-ului dispozitivelor joacă un rol semnificativ în securitatea și robustețea sistemului. Sistemele IoT și aplicațiile lor sunt, de fapt, un sistem distribuit foarte dinamic și modular. Programarea distribuită și sincronizarea este în general privită ca o sarcină netrivială. Există mai multe cadre și limbaje de nivel înalt care abordează problema programării distribuite. Limbajul AGAPIA [50] încearcă să crească productivitatea dezvoltatorului și expresivitatea limbajului folosind un model de comunicare transparent și declarații simple de nivel înalt. AGAPIA exprimă aplicațiile distribuite ca *module* care expun o interfață I/O clară, simplă și structurată.

AGAPIA este un limbaj specific domeniului (DSL) construit pe deasupra limbajului de programare C. Din acest motiv, (credem că) poate fi utilizat cu ușurință cu codul firmware existent pentru a modela interacțiunea sistemului IoT și a expune interfața I/O a dispozitivului.

Explorăm opțiunea de a exprima dispozitivele IoT ca module AGAPIA și de a reprezenta interacțiunile sistemului IoT ca relații între modulele AGAPIA. Procedând astfel, avem o specificație clară, structurată a I/O-ului unui dispozitiv. Putem extinde cu ușurință AGAPIA pentru a converti specificațiile I/O structurate în (și din) JSON, astfel încât să putem folosi cu ușurință RiverIoT pentru a testa atât dispozitivul, cât și sistemul IoT.

4.1.2 AGAPIA

Există o nevoie reală de a dezvolta un limbaj de programare unificator, care să permită dezvoltatorilor să-și prototipeze rapid și să testeze software-ul înainte de a-l implementa pe un dispozitiv IoT. Tendința actuală este de a scrie codul într-un limbaj de programare de nivel scăzut, cum ar fi C, care va fi compilat cu compilatoare specializate (cum ar fi Arduino IDE) pentru fiecare dispozitiv IoT, deoarece codul poate fi rulat pe arhitecturi diferite (cum ar fi PIC, AVR). , ARM).

Problema cu scrierea într-un limbaj de programare de nivel scăzut este că este un proces obositor și predispus la erori, deoarece trebuie să implementăm cu atenție structurile de bază de date și canalele de comunicare. Deși acest lucru oferă multă flexibilitate și control în ceea ce privește granularitatea instrucțiunilor, împiedică dezvoltarea. Chiar și atunci când cineva are acces la un dispozitiv IoT puternic (cum ar fi Raspberry PI) și poate scrie cod într-un limbaj de programare de nivel superior, cum ar fi Python, o problemă persistă: este dificil și consuma mult timp să scrieți programe corecte, distribuite.

Există lucrări care investighează o modalitate de a programa folosind obiecte vizuale precum [51] și [52], dar nu se dezvoltă pentru dispozitivele multiprocesor.

AGAPIA [53] este un limbaj specific domeniului (DSL) care a fost conceput pentru a simplifica dezvoltarea software-ului paralel folosind o sintaxă mai simplă pentru generarea de noi procese prin intermediul fork-urilor și integrarea cu Open-MPI [54] pentru a oferi o distribuție mai prietenoasă. și experiență de programare paralelă.

Componenta principală a unui program AGAPIA este modulul care poate fi gândit la un bloc bidimensional (un pătrat) care poate primi informații atât din partea de nord cât și din partea de vest și poate trece informații atât prin părțile de est cât și de sud (Figura ??). Intrările și ieșirile sunt opționale, astfel încât se pot defini module care generează date (fără intrări, doar ieșiri) sau care produc efecte secundare atunci când sunt date anumite intrări (doar intrări, fără ieșiri).

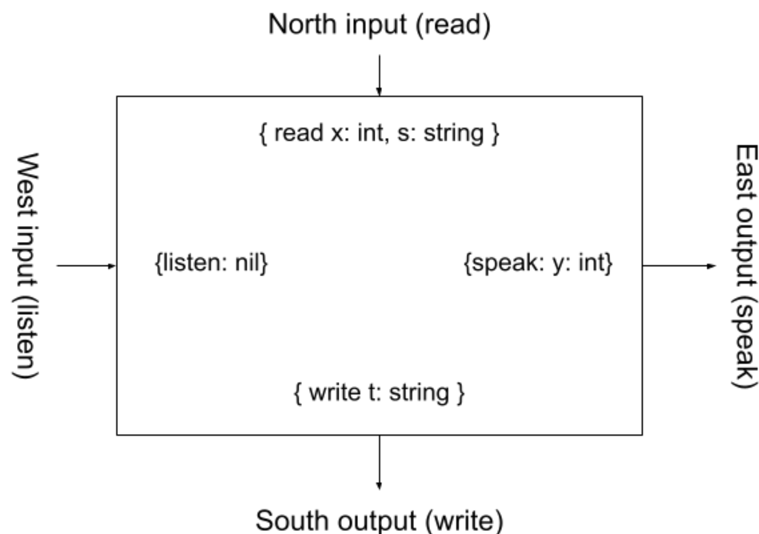


Figura 4.1: Un modul AGAPIA

Sarcina principală a unui modul este de a transforma intrările date în ieșiri. Modul de bază de definire a unui modul este de a defini interfața acestuia (intrarile și ieșirile) și pe baza acestora se pot compune module împreună.

Prin definirea unui modul în acest fel, se pot compune module în trei moduri: vertical, orizontal și diagonal (Figurile ??, 4.3 și 4.4). Un program AGAPIA se poate gândi la o structură bidimensională [55], compusă din module care comunică între ele pentru a rezolva o problemă mai eficient. Modelul de compoziție folosit de AGAPIA este multiplexat atât în spațiu (Figura ??), cât și în timp (Figura 4.3) [56].

După definirea interfeței pentru fiecare modul, se poate scrie codul pentru acel modul în limbajul C sau C++. Codul utilizatorului pentru fiecare modul poate conține fie cod pur C/C++ (numit aici atomic), fie poate conține instrucțiuni sau compoziții AGAPIA [57]. Diferența dintre cele două moduri este că codul atomic trebuie să aibă acces la toate varia-

bilele înainte de a fi programat, în timp ce atunci când se utilizează instrucțiuni AGAPIA se poate rula codul în paralel.

Echipa AGAPIA lucrează în prezent la producerea unei interfețe grafice cu utilizatorul în care se poate defini cu ușurință topologia unui program AGAPIA și se poate defini interfața dintre module și apoi se poate scrie codul modulului.

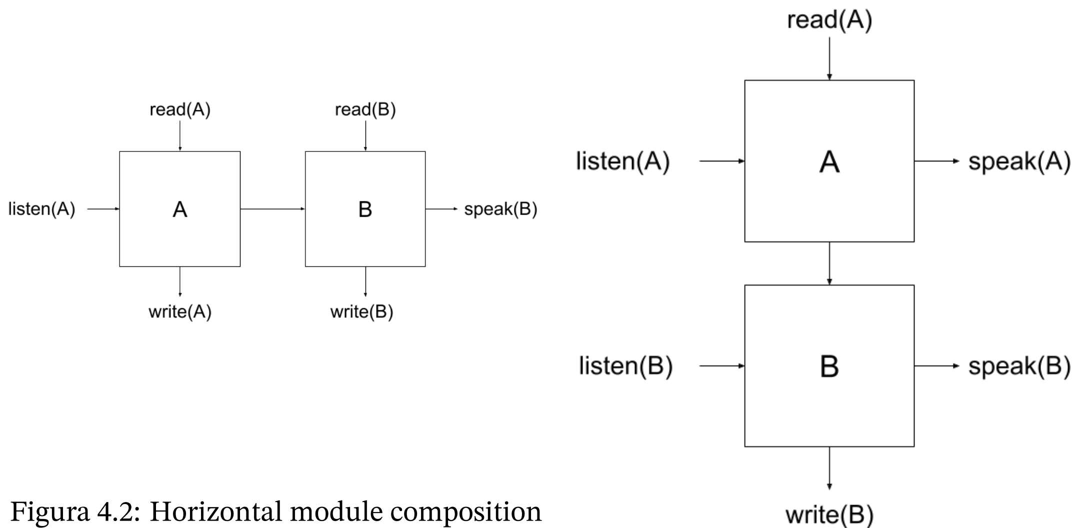


Figura 4.2: Horizontal module composition

Figura 4.3: Vertical module composition

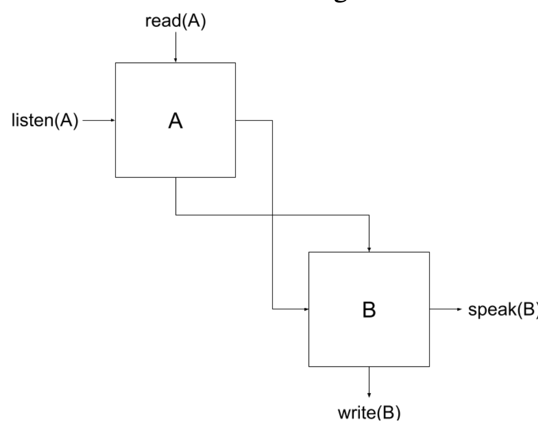


Figura 4.4: Diagonal module composition

4.1.3 RiverIoT

RiverIoT este un cadru de testare integrat care permite fuzzing end-to-end pentru sistemele IoT. Cadrul folosește fuzzing ghidat prin metode de ultimă generație, cum ar fi execuția concolică [58] și tehnici AI [59], [60], [60] , [61].

Există două provocări majore la fuzzing-ul dispozitivelor încorporate: 1) dependența hardware a firmware-ului și 2) natura sursă închisă a firmware-ului. Inspirat de [62] și [63],

RiverIoT abordează ambele probleme prin emulare. Utilizarea unui emulator permite RiverIoT să ruleze firmware-ul pe un sistem de uz general, fără a necesita 1) hardware-ul fizic al dispozitivului IoT. Când un binar este emulat, emulatorul traduce fiecare instrucțiune binară într-o reprezentare intermediară înainte de a o traduce din nou în instrucțiuni pentru arhitectura mașinii gazdă. Procedând astfel, RiverIoT poate instrumenta în mod dinamic instrucțiunile binare fără a avea nevoie de 2) codul sursă.

Deoarece un sistem IoT poate fi reprezentat logic ca un grafic al componentelor conectate, RiverIoT poate efectua testarea fuzz la trei niveluri diferite:

- graph level - schimbarea nodurilor și marginilor din interiorul sistemului IoT implementat
- node level - fuzzing fiecare dispozitiv individual
- interconectarea nodurilor de intrare/ieșire între diferite noduri și comunicarea acestora (adică modul în care intrarea unui nod poate fi influențată de alte noduri care sunt conectate la acesta sau modul în care canalul de comunicare între noduri le poate afecta intrarea).

Pentru a utiliza RiverIoT, trebuie să furnizați o descriere a graficului (noduri și margini) și a structurii I/O pe care aplicațiile o așteaptă, în format JSON. Cu această specificație JSON, cadrul poate înțelege comunicarea dintre diferitele componente și poate începe să modifice graficul și să genereze intrări pentru aplicații.

RiverIoT se așteaptă ca graficul să poată gestiona configurații dinamice în timpul rulării, cum ar fi schimbarea nodurilor și a marginilor. Motivul pentru aceasta este că cadrul încearcă să simuleze probleme precum defecțiunile dispozitivului, repornirile, repoziționarea testând astfel rezistența sistemului testat.

După cum se poate vedea în specificația graficului prezentată în Lista 4.1, modelul este destul de simplu și sigur-explicativ. Trebuie definite aplicațiile IoT (”iot-nodes”) și canalele lor de comunicare (”io-edges”). Fiecărei aplicații i se atribuie un ID și o descriere a bufferelor sale de intrare și ieșire, așa cum este exemplificat în Listarea 4.2. Fiecare margine, de asemenea, identificată printr-un ID, descrie nodul de intrare și de ieșire (pe baza ID-ului nodului) și specificația bufferului utilizată pentru acel nod (pe baza ID-ului buffer-ului nodului).

```
1  {
2    "configuration-name": "Generic Network",
3    "iot-nodes": {
4      "1": { // Buffer description },
5      "2": { ... }
6    },
7    "io-edges": {
8      "1": {
9        "vout": 3, // Output node
10       "vin": 1, // Input node
11       "vout-buffer": 1, // Buffer id sending the data
12       "vin-buffer": 2, // Buffer id receiving the data
```

```
13      },
14      "2": { ... }
15    }
16  }
```

Listing 4.1: Exemplu de specificare a graficului de compatibilitate

Exemplul din Listarea 4.2 conține probabil mai multe câmpuri decât s-ar putea aștepta, dar cele mai multe dintre ele sunt de fapt acolo pentru a oferi mai mult context cititorului. RiverIoT este interesat în principal de câmpurile „tip token” și „dimensiune octet”. Cu cele două, poate începe să genereze intrări pentru aplicația testată.

```
1  {
2    "device-name": "An IP Camera",
3    "optional": "false",
4    "class": "camera",
5    "buffers": {
6      "1": {
7        "token-delimitators": " ",
8        "protocol": "HTTP",
9        "protocol-setting": "http://192.168.0.112:8080/",
10       "buffer-tokens": [{
11         "name": "Camera command",
12         "description": "Input that selects command",
13         "token-type": "string",
14         "byte-size": 256,
15         "regex-rule": "[a-zA-Z]+=[a-zA-Z0-9]+", // Optional parameter to
               guide fuzzer generator
16         "optional": false
17       },
18       {
19         "name": "Camera ISO Value",
20         "description": "Sensitivity to light",
21         "token-type": "int",
22         "byte-size": 4,
23         "optional": true
24       }]
25     },
26   }
27 }
```

Listing 4.2: Specificații pentru un singur dispozitiv buffer

Cu specificația JSON gata, cadrul poate începe procesul de fuzzing. Pentru fiecare nod, cadrul va efectua fuzzing ghidat izolat, testând fiecare program binar care poate fi implementat pe un dispozitiv fizic. RiverIoT folosește o combinație de execuție simbolică [58] și algoritmi genetici [64] pentru a fuziona țintele în încercarea de a obține o acoperire bună a codului, păstrând în același timp performanța de rulare și overhead acceptabile.

4.1.4 Observații notabile

Propunem o extensie a limbajului de programare AGAPIA cu macrocomenzi de implementare conștiente de IoT, care permite o integrare rapidă și simplă cu un cadru de testare, cum ar fi RiverIoT. Extensiile adăugate generează o specificație JSON a întregului sistem testat (SUT), care descrie atât graficul dispozitivelor IoT conectate, cât și interfețele lor de comunicare I/O. Procedând astfel, oferă cadrului de testare o vedere la nivel înalt a întregului sistem, precum și intrarea și ieșirea așteptate pentru fiecare nod. Specificația generată este citită de componenta de orchestrare a RiverIoT, care decide dacă va efectua generarea de intrare sau mutații grafice pentru SUT. Pe scurt, această cercetare investighează modul în care modificările minore aduse limbajelor de programare existente pot îmbunătăți procesele de testare și validare pentru sistemele IoT.

4.2 Execuție concolică scalabilă a binarelor COTS cu cadrul River

Sistemele software cresc în complexitate pe măsură ce trece timpul și apar noi cerințe de afaceri. Împreună cu ritmul rapid și cererea în creștere a dezvoltării de software, acest lucru face loc pentru erori și duce la o suprafață de atac mai mare. Testarea fuzz și execuția simbolică s-au dovedit a fi metode de succes pentru a descoperi căi și vulnerabilități netestate și au fost adoptate de echipe din întreaga lume. Produsele software folosesc în mod regulat biblioteci și componente precompilate de la terți. Acele binare custom of-the-shelf (COTS) fac execuția simbolică dificilă, deoarece fuzzer-ul nu poate introduce codul de instrumentare în timpul compilării, fiind astfel nevoit să recurgă la instrumentare binară dinamică. Vă prezentăm River, un fuzzer de execuție concolică pentru binare COTS. Emulăm sistemul sub testare (SUT) cu Triton, o bibliotecă de analiză binară dinamică și delegăm apelurile de sistem și bibliotecă către GDB. Procedând astfel, evităm explozia căii și nevoia de a implementa manual apelurile de sistem și bibliotecă în cadrul fuzzing, două probleme pe care le au KLEE și angr. Comparăm soluția noastră cu KLEE testând bibliotecile de la Google FuzzBench. După ce a rulat libarchive sub cele două timp de 30 de minute, River are o acoperire a ramurilor de 4,56%, în timp ce KLEE are o acoperire de 1,78%; după 1h30 de minute, River a ajuns la 6,06% față de 2,07% atins de KLEE.

4.2.1 Introducere

Securitatea software-ului este esențială pentru sistemele din zilele noastre. Lumea a fost martoră la creșterea atacurilor de securitate cibernetică în ultimii ani, așa cum este prezentat în Figura 1.1. Exploatarea vulnerabilităților în sălbăticie a costat companiile de miliarde [1], cercetătorii de la IBM System Science Institute estimează că repararea unei vulnerabilități costă de 100 de ori mai mult decât costurile de dezvoltare[2].

În domenii precum IoT și auto, auditarea securității software-ului este o provocare, deoarece utilizatorul primește o bucată de hardware cu un binar personalizat disponibil (COTS)

care rulează pe acesta. Utilizatorul este obligat să-și pună încrederea în terțul, deoarece nu este capabil să valideze software-ul, deoarece nu are acces la codul sursă.

Testarea Fuzz poate ajuta echipele să automatizeze procedura de testare și să descopere căi și vulnerabilități netestate în produsele lor și componentele terțe. Dacă codul sursă este disponibil, cadrul de fuzzing va adăuga cod de instrumentare la binarul generat, ceea ce permite cadrului să monitorizeze sistemul testat (SUT) în timpul rulării și să îmbunătățească faza de generare a intrării fuzzer-ului; aceasta este cunoscută sub numele de *white-box fuzzing*. Când nu există cod sursă disponibil, fuzzer-ul generează intrare (fie de la zero, fie dintr-un corpus inițial) și verifică ieșirea (starea finală) a SUT-ului pentru a determina dacă a avut loc un accident; aceasta este cunoscută sub numele de *black-box fuzzing*. Între cele două metode se află *grey-box fuzzing*: chiar dacă codul sursă nu este disponibil, fuzzer-ul va instrumenta instrucțiunile binare pentru a monitoriza sistemul în timpul rulării și a ghida procesul de fuzzing.

Rescrierea binară statică înseamnă posibilitatea de a modifica un binar compilat astfel încât să rămână executabil după modificări. Instrumentația binară statică ar oferi cea mai bună performanță pentru un fuzzer, dar rescrierea binară este o problemă de cercetare grea, activă[65][66] [67][68][**?**] pentru a avea o soluție solidă, inter-arhitectură. Instrumentația binară dinamică (DBI) modifică execuția binară în timpul rulării. Acest lucru este mult mai ușor de realizat, deoarece nu depinde de o analiză statică complexă și permite utilizatorului să utilizeze informațiile de rulare[69][70][71][72][73]. Dezavantajul utilizării DBI este supraîncărcarea de rulare pe care o implică, care poate varia între 1,2x și 5x. La fuzzing, se dorește cea mai mică instrumentație deasupra capului, deoarece aceasta va permite fuzzer-ului să încerce mai multe intrări în aceeași perioadă de timp. Intenționăm să compensăm cheltuielile generale suportate de DBI prin utilizarea execuției simbolice.

Dorim să putem realiza fuzzing eficient în caseta gri pe binare COTS. Problema cu fuzzing-ul în cutie neagră este că nu știți nimic despre ținta dvs.: ca atare, doar testarea împotriva intrării aleatorii nu va ajunge prea departe. Astfel, dorim să îmbunătățim River fuzzer pentru a realiza execuția simbolică pe binare prin DBI: 1) urmărim execuția binară, 2) de fiecare dată când întâlnim o condiție de ramificare, salvăm condiția întâlnită sub formă de relații între variabile simbolice , 3) variabilele simbolice reprezintă constrângerile noastre de cale și vor construi o ecuație algebrică a căii de execuție curente, 4) astfel încât să putem furniza ecuația unui solutor de satisfaciabilitate modulo teorii (SMT) [74][75], cum ar fi Z3[76], pentru a descoperi dacă există valori de intrare care satisfac constrângerile unei anumite căi de program. Prin utilizarea execuției simbolice, se pot descoperi rapid valorile de intrare „interesante" care îmbunătățesc acoperirea codului unui SUT și declanșează vulnerabilități; acest lucru se datorează faptului că ecuația constrângerilor reprezintă matematic diferitele stări pe care un fuzzer obișnuit le-ar putea descoperi prin rularea mai multor intrări.

Apelurile de funcții de bibliotecă și sistem reprezintă cele mai mari provocări ale execuției simbolice pe binare. Figura 4.5 prezintă un flux regulat de aplicație, cu toate tranzițiile pornind de la spațiul utilizatorului și terminând în spațiul kernel. Începând cu instrucțiu-

nea punct de intrare în program, vom itera prin instrucțiunile binare până ajungem la un apel de funcție, eventual tradus într-un apel de funcție de sistem sau bibliotecă.

Apelurile de funcție de bibliotecă duc la o ecuație uriașă a constrângerilor complicate, cunoscută în literatură sub numele de *explozie de cale*[77]. Ecuația ar putea deveni, fără îndoială, exponențială ca urmare a numărului de ramuri, ceea ce mărește mult timpul necesar rezolvatorului SMT pentru a găsi soluții de ecuație, până la punctul în care durează prea mult pentru a fi utilizabilă sau nu mai poate găsi o soluție. Chiar dacă am avea un solutor SMT ideal care nu este afectat de problema exploziei căii, apelurile de sistem reprezintă o nouă provocare: implementarea apelului se află în spațiul nucleului, astfel încât fuzzer-ul nu poate instrumenta acele instrucțiuni deoarece nu are acces la ele. . Soluția este fie reimplementarea, fie emularea unor astfel de apeluri.

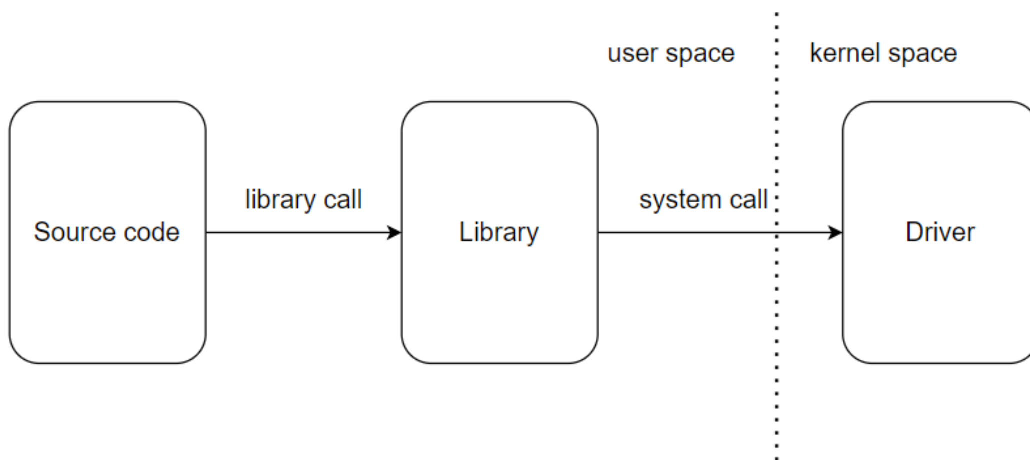


Figura 4.5: Fluxul de aplicație: de la spațiul utilizator la spațiul kernel

Alte fuzzers și cadre de execuție simbolică, cum ar fi angr[78] și KLEE[79], reimplementează funcții comune ale bibliotecii (de exemplu, `strlen`, `printf` etc.) și apeluri de sistem (de exemplu, `open`) , `citiți`, `scrieți` etc.) pentru a ocoli problemele prezentate mai sus. Există două probleme cu această abordare: 1) aceasta nu se scalează, deoarece nu se pot cunoaște toate funcțiile care vor fi utilizate de un utilizator în baza sa de cod, prin urmare este nevoie de o analiză extinsă pentru a implementa stub-urile lipsă, astfel încât să se poată fuz ținta binară și 2) implementarea personalizată s-ar putea să nu se comporte exact la fel ca biblioteca originală sau apelul de sistem, nedorind astfel să modifice comportamentul sistemului testat.

Soluția noastră propusă va folosi River pentru a instrumenta în mod dinamic doar instrucțiunile care fac parte din binarul analizat (fără biblioteci sau apeluri de sistem). Noi integrăm River cu GDB, astfel încât să îi putem delega execuția apelurilor de bibliotecă și de sistem. Astfel, River va urmări doar și va executa simbolic instrucțiunile binare de interes.

Există o serie de aspecte pe care trebuie să le luăm în considerare după ce GDB execută apelurile de sistem sau de bibliotecă:

- trebuie să mențină consistența dintre amintirile fiecărui proces. Procesul de la River trebuie să aibă aceleași informații chiar și după execuția GDB.
- trebuie să restabilească contextul funcției după execuția GDB.
- trebuie să păstreze starea simbolică consecventă înainte și după apelul unei funcții de sistem sau bibliotecă.

Avantajul semnificativ pe care îl avem prin abordarea noastră este că emulăm apelurile de funcție de sistem și bibliotecă. Astfel, dezvoltatorul nu are suprasarcina asociată cu implementarea fiecărui apel de sistem și nici nu trebuie să facă altceva pentru a efectua execuția concolică. Pretindem că implementarea noastră este scalabilă, deoarece nu necesită dezvoltatorului să depaneze instrumentul fuzzing și să implementeze orice utilități lipsă.

4.2.2 Arhitectura Fuzzerului River

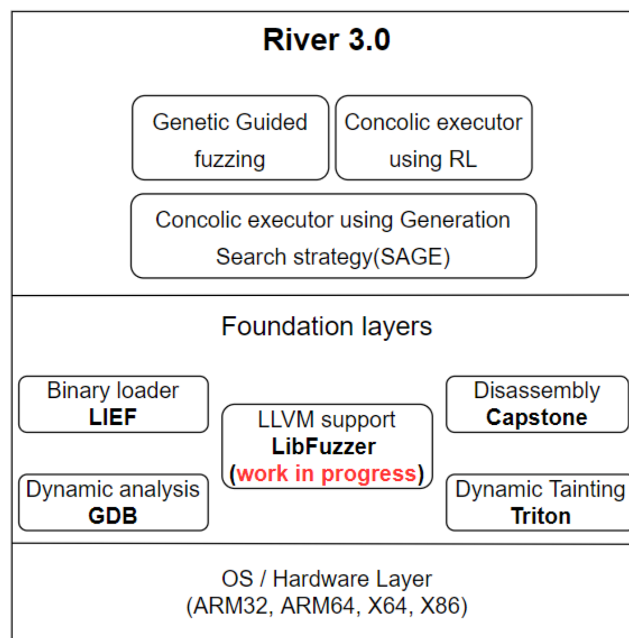


Figura 4.6: Arhitectura River

River este un fuzzer multiplatformă care poate rula pe toate sistemele de operare comune și acceptă următoarele arhitecturi: **X64, X86, ARM32** și **ARM64**. Componentele fundamentale ale arhitecturii sale sunt construite folosind următoarele biblioteci:

- **LIEF**[80] servește la încărcarea fișierelor binare. Folosind acest instrument, putem modifica fișierul executabil, putem accesa offset-urile, secțiunile și punctul de intrare, funcționalitățile utilizate în restabilirea memoriei și ajutăm la determinarea poziției instrucțiunilor necesare.
- **Triton**[81] utilizat în execuția simbolică dinamică. Mai mult decât atât, utilizarea

sa în râu este de a crea un AST și de a face o analiză dinamică a impurităților care verifică implicațiile de securitate pe baza fluxului generat de intrarea utilizatorului. Putem accesa funcții din memorie cu aceasta în River. Această componentă are două funcționalități substanțiale: funcția de analiză a impurităților și motorul de execuție simbolică. Mai mult, putem modifica regiuni ale memoriei și putem seta expresii simbolice la diferite adrese de memorie.

- **Capstone**[82] este o componentă a bibliotecii LIEF, deoarece putem dezasambla fișierele executabile prin intermediul acesteia. Acesta este un cadru de înaltă performanță, conceput pentru a oferi semantica instrucțiunilor dezasamblate. În plus, este un instrument sigur pentru fire.
- **z3-solver**[76] este un rezolvator de teoreme. Utilizarea sa este în biblioteca River și Triton pentru a rezolva diferite constrângeri adunate din instrucțiunile de ramură. Rezultatul după evaluarea unor astfel de expresii este o intrare validă care alimentează fuzzer-ul pentru a detecta defecte în sistem.
- **GDB**[83] este un instrument prin care emulăm biblioteca și apelurile de sistem și restaurăm contextul. Putem inspecta memoria în diferite momente de la execuție, folosindu-și capacitățile de analiză dinamică. Putem analiza conținutul registrelor și obține informații despre secțiunile de memorie.

Acest cadru acceptă trei metode de fuzzing, după cum puteți vedea în figura 4.6: execuție concolică bazată pe strategia de căutare generație[84, 85], fuzzing ghidat genetic și execuție concolică folosind Reinforcement Learning[**?**, 85]. Asemănarea este că toate aceste metode generează căi pe baza instrucțiunilor din fișierele executabile. Diferențele provin din modul în care fuzzerii aleg căile. Fuzzing-ul ghidat genetic calculează urmele pe baza unui algoritm genetic, în timp ce execuția concolică folosind RL se bazează pe un algoritm de învățare prin întărire.

4.2.3 Evaluare

Pentru a valida soluția noastră, am evaluat o serie de aplicații care încearcă să îndeplinească cerințele din lumea reală. Am folosit un parser HTTP[86] (Figura 4.8) și o aplicație care încearcă să analizeze un obiect JSON, numit Fuzzgoat[87] (Figura **??**). O alegem pe prima deoarece conține apeluri obișnuite de sistem și funcții de bibliotecă întâlnite în fiecare program, iar cea de-a doua pentru că este destinată a fi folosită ca un banc de testare pentru fuzzere. Un alt motiv este că fuzzer-ul acoperă o parte semnificativă a codului într-o perioadă rezonabilă de timp, în ciuda costurilor generale cauzate de emulare.

Analizorul HTTP are două componente: analizatorul pentru cerere și unul pentru adresa URL. Acoperirea liniei pentru această bibliotecă este că analizatorul URL are 21%, în timp ce cealaltă componentă are 79%. Alegem să testăm analizatorul URL, deoarece intrarea nu este foarte structurată, așa cum este pentru analizarea solicitărilor HTTP. Pentru a susține intrări foarte structurate într-o perioadă de timp sigură, River trebuie să genereze testele de date bazate pe gramatică. Abordarea actuală creează intrări bazându-se pe execuția

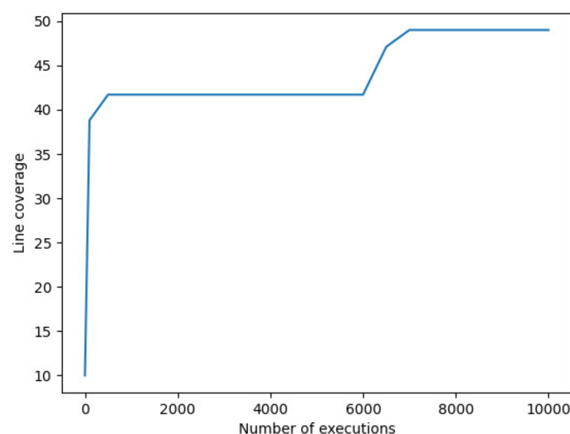


Figura 4.7: Line coverage for Fuzzgoat test

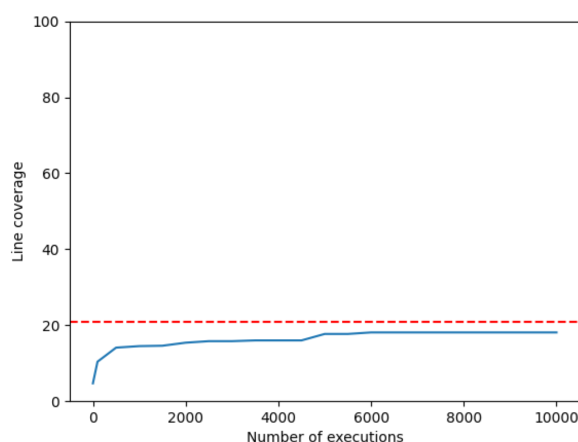


Figura 4.8: Line coverage for http parser test

simbolică, ceea ce va dura mult mai mult timp pentru a construi intrări adecvate.

Metrici și performanțe

Până acum am evaluat că cerințele noastre de funcționalitate sunt îndeplinite cu succes de soluția noastră. În continuare, comparăm abordarea noastră cu cele similare de la Klee. Am folosit bibliotecile Libre [88] și Libarchive [89] din Fuzzbench de la Google [90, 91]. Am folosit și Fuzzgoat, menționat anterior, în setul nostru de date. Rezultatele sunt prezentate în următoarele tabele:

TIME	RIVER	KLEE
30min	49%	65%
1h	49%	73%
1h30min	49%	76%

Tabela 4.1: Branch coverage for Fuzzgoat example.

TIME	RIVER	KLEE
30min	4.56%	1.78%
1h	6.02%	1.92%
1h30min	6.06%	2.07%

Tabela 4.2: Branch coverage for Libarchive example.

TIME	RIVER	KLEE
30min	28.9%	fail
1h	28.9%	fail
1h30min	28.9%	fail

Tabela 4.3: Line coverage for Libre example.

Valoarea noastră pentru a compara ambele instrumente este acoperirea ramurilor, deoarece putem obține doar acoperirea ramurilor și a instrucțiunilor, în timp ce avem posibilitatea de a analiza acoperirea liniei și ramurilor în River, deoarece cel din urmă testează acoperirea folosind instrumentul gcov, iar primul are o acoperire internă. mecanism care face măsurătorile. Deci, valoarea partajată este acoperirea sucursalei.

Putem observa din rezultatele de mai sus că Klee are o acoperire mare pentru exemplul Fuzzgoat, care conține funcții obișnuite și apeluri la bibliotecă. Celelalte două exemple sunt mai complexe în ceea ce privește apelurile de sistem și bibliotecă. Mai mult, aceste aplicații presupun un număr substanțial de astfel de apeluri. Klee fuzzer eșuează cu un semnal SEGFAULT pentru exemplu Libre din cauza lipsei de implementare pentru un apel de asamblare inline din fișierul `valgrind.cc`. Eroarea apare chiar dacă Klee a fost compilat cu suport pentru libc++. Un posibil motiv este lipsa unei implementări pentru unele apeluri de funcție de sistem sau bibliotecă. River încearcă să evite această problemă emulând astfel de apeluri prin GDB. Putem observa din tabelul 4.2 că Klee este mai lent atunci când aplicația implică un număr notabil de apeluri de funcție de sistem și bibliotecă. Mai mult, exemplul Libre implică un număr semnificativ de apeluri de sistem de la POSIX. Chiar dacă Klee are sprijin pentru astfel de apeluri, abordarea lor implică o suprasolicitare substanțială pentru ei.

4.2.4 Observații notabile

Am adăugat cu succes suport pentru execuția simbolică pentru binarele COTS în River fuzzer. Prin delegarea execuției syscall-urilor și a funcțiilor bibliotecii către GDB, reducem semnificativ riscul declanșării unei explozii de cale. Mai mult, cheltuielile generale asociate cu reimplementarea unor astfel de apeluri se diminuează drastic. Mai mult, prin această abordare, dorim să subliniem posibilitatea emulării apelurilor de funcții de sistem și bibliotecă prin păstrarea consecvenței stărilor concrete și simbolice.

Restaurarea stării simbolice este un aspect esențial al abordării curente, deoarece fuzzer-ul

poate pierde anumite valori simbolizate după apelurile de funcție de sistem sau de bibliotecă. Pentru a preveni orice erori, avem o abordare „agresivă”, restabilind memoria și contextele simbolice după fiecare apel delegat GDB. Politica de restaurare ar putea fi relaxată pentru a obține performanțe mai bune la timp de rulare.

Îmbunătățiri viitoare ale performanței

Soluția noastră de restabilire a contextului este pe deplin implementată, ceea ce este crucial pentru abordarea noastră. Cu toate acestea, implementarea actuală este foarte precaută și copiază secțiuni mari de memorie din GDB în procesul River de fiecare dată când este efectuat un apel de sistem sau de bibliotecă pentru a actualiza contextul în fuzzer. Acest lucru face ca memoria să devină un blocaj, deoarece performanța instrumentului este direct legată de dimensiunea și numărul de secțiuni de memorie copiate. În plus față de optimizarea utilizării memoriei, instrumentul de testare ar putea beneficia și de pe urma utilizării funcționalităților GDB și a utilizării operatorului de semnal GDB pentru a îmbunătăți procesul de generare a intrărilor prin filtrarea intrărilor pe baza testelor anterioare de date care au cauzat erori în sistem.

Un factor important de luat în considerare este modul în care fuzzer-ul își generează intrările, deoarece dacă folosește aceeași cale de mai multe ori, poate duce la o scădere a performanței. Pentru a îmbunătăți performanța, fuzzer-ul ar putea evita generarea de teste de date redundante modificându-și abordarea pentru generarea de intrări. Optimizarea acestei componente poate duce la o reducere a timpului necesar pentru testare. Astfel, fuzzer-ul ar putea atinge o acoperire mai largă a codului în aceeași perioadă de timp prin ajustarea procesului său de generare a intrării.

Îmbunătățiri viitoare ale emulării

În timpul emulării, fuzzer-ul urmărește să ghideze execuția concretă pentru a obține execuția corectă. Totuși, așa cum sa discutat pe parcursul acestui capitol, fuzzer-ul are și o reprezentare internă a stării simbolice, de care GDB nu este conștient, deoarece scopul său este doar de a executa aplicația. În cele mai multe cazuri, fuzzer-ul nu trebuie să ia în considerare starea simbolică atunci când gestionează apelurile de sistem și de funcții de bibliotecă, deoarece nu afectează alte adrese. Acest lucru se schimbă atunci când apelurile bibliotecii, în cazul în care ar trebui să opereze pe memorie simbolică, deoarece trebuie luată în considerare de analiza impurităților. De exemplu, după un apel `memcpy`, adresa de destinație ar trebui să aibă aceleași informații simbolice ca și adresa sursă, dar starea simbolică a adresei de destinație nu va fi actualizată de GDB deoarece nu poate interacționa cu informațiile simbolizate. În acest sens, soluția actuală se bazează pe cârlige de funcții bine cunoscute, de modificare a memoriei, dar acest lucru limitează suportul. Pentru a îmbunătăți acest lucru, trebuie să proiectăm o abordare mai generică pentru a gestiona aceste apeluri care necesită lucrul cu starea simbolică.

Capitolul 5

Concluzii

5.1 Rezumatul tezei

Capitolul 2 a prezentat contribuțiile noastre privind îmbunătățirea securității nucleului Linux. Ne-am concentrat pe utilizarea limbajului de programare D ca o modalitate de a îmbunătăți securitatea nucleului Linux. Mai exact, am evidențiat modul în care D poate fi folosit pentru a scrie drivere pentru nucleu care sunt mai sigure decât cele scrise în alte limbi, cum ar fi C. Am demonstrat că este posibil să convertim cu ușurință codul C în D fără nicio pierdere de performanță. Acest lucru este important deoarece înseamnă că dezvoltatorii pot folosi D pentru a scrie drivere sigure fără a fi nevoiți să sacrifice performanța.

În plus, am prezentat și munca noastră privind DPP, un instrument care automatizează procesul de traducere a structurilor de date kernel în cele care sunt compatibile cu D. Acest lucru face mai ușor pentru dezvoltatori să folosească D pentru a scrie drivere în ecosistemul nucleului și poate ajuta la îmbunătățirea în continuare a securității nucleului Linux. În general, contribuțiile noastre urmăresc să faciliteze crearea de drivere sigure în ecosistemul kernel-ului Linux.

Capitolul 3 a prezentat contribuțiile noastre la îmbunătățirea securității aplicațiilor. Unul dintre domeniile principale pe care ne-am concentrat a fost dezvoltarea unui nou cadru de colecții pentru biblioteca standard D. Acest cadru este conceput pentru a putea deduce siguranța operațiunilor pe care le efectuează pe baza tipului de date pe care le conține. De exemplu, dacă se știe că tipul de date este sigur, operațiunile efectuate asupra acestuia vor fi, de asemenea, considerate sigure. Colecțiile noastre acceptă, de asemenea, alocători de memorie personalizați, a căror utilizare poate oferi beneficii semnificative de performanță în comparație cu colecțiile de biblioteci standard existente.

În plus, am discutat și despre munca noastră privind construirea unui generator de biblioteci bazat pe specificațiile OpenAPI. Acest instrument este conceput pentru a ajuta dezvoltatorii să creeze rapid și ușor biblioteci pentru a fi utilizate în aplicațiile lor și are scopul de a îmbunătăți securitatea și fiabilitatea acestor aplicații.

Capitolul 4 se concentrează pe validarea securității dispozitivelor Internet of Things (IoT). În primul rând, explorează posibilitatea generării automate a unei descrieri structurate a unei rețele IoT care ar putea fi utilizată de cadrul River fuzzing pentru a evalua dispozitivele de rețea.

În continuare, prezintă munca pe care am făcut-o pentru a îmbunătăți fuzzing-ul binarelor

proprietare cu cadrul River. Am abordat provocările de a efectua execuția simbolică pe binare COTS. Execuția simbolică este o tehnică folosită pentru a analiza comportamentul unui program prin tratarea anumitor intrări ca variabile simbolice, mai degrabă decât valori concrete. Acest lucru poate fi util pentru detectarea și identificarea vulnerabilităților în software, dar poate fi dificil de realizat pe binare COTS, deoarece acestea sunt compilate și optimizate pentru arhitecturi hardware specifice. Am discutat despre limitările motoarelor actuale de execuție simbolică și despre modul în care soluția noastră a reușit să depășească aceste probleme.

5.2 Contribuții

Această teză a prezentat câteva contribuții care îmbunătățesc securitatea aplicațiilor IoT și procesul de auditare a acestor aplicații.

- Am oferit un context cuprinzător cu privire la limbajele de programare sigure pentru memorie, propunerile anterioare și actuale pentru limbaje, altele decât C, pentru a fi utilizate în nucleul Linux și lucrări de ultimă generație legate de analiza binară și detectarea automată a vulnerabilităților.
- Am prezentat o abordare care folosește D pentru a îmbunătăți securitatea nucleului Linux.
- Am selectat *virtio_net* ca driver-țintă, o componentă de dimensiune medie și întreținută activ în kernel-ul Linux. Am portat driverul în limbajul de programare D și am evidențiat paritatea funcțională și de performanță cu driverul C original și am discutat despre beneficiile securității.
- Am dezvoltat o metodologie pentru portarea driverelor de kernel la D pentru a îmbunătăți siguranța generală a unui sistem.

  datorita
- Am demonstrat că nucleul poate folosi D pentru a beneficia de îmbunătățiri de siguranță într-un modul de nucleu, verificarea limitelor matricei și polimorfismul în timp de compilare fiind cele mai importante.
- Folosind munca noastră, mai multe drivere pot fi portate în D și astfel crește mecanismele de siguranță care sunt prezente în nucleu.
- Am îmbunătățit DPP, un instrument de traducere a antetelor C/C++, pentru a permite utilizarea sa pe anteturile kernel-ului Linux. Astfel, automatizăm traducerea structurilor de date ale nucleului: o sarcină altfel manuală care este repetitivă, consumatoare de timp și predispusă la erori.
- Ne-am validat munca aplicând-o pe portul nostru `virtio_net` D Proof-of-Concept (PoC). Acest lucru ne-a permis să reducem dimensiunea implementării driverului cu 53%; o bază de cod redusă reprezintă o suprafață de atac mai mică, crescând astfel securitatea generală.

- Am îmbunătățit generarea automată de cod pentru ecosistemul D, ceea ce duce la creșterea productivității și la reducerea timpului de întreținere. Toate proiectele D, nu numai cele legate de kernel, beneficiază de munca noastră.
- Am dezvoltat un nou cadru de colecții care reunește toate caracteristicile importante ale limbajului D, fiind în același timp ușor de utilizat și intuitiv pentru utilizator. Colecțiile noastre permit utilizarea unor alocatoare de memorie personalizate, furnizate de utilizator, permițând astfel o performanță îmbunătățită.
- Am îmbunătățit siguranța alocatoarelor standard de biblioteci în ceea ce privește aplicațiile multithreading. Munca noastră este utilizată în mod activ de comunitatea D.
- Am implementat un generator automat de biblioteci D pentru serviciile care își descriu API-ul prin Standardul OpenAPI. Bibliotecile generate respectă regulile de siguranță ale memoriei D, permițând aplicațiilor să interacționeze în siguranță cu serviciile de Internet.
- Am propus o extensie a limbajului de programare AGAPIA cu macrocomenzi de implementare IoT care permit o integrare rapidă și simplă cu un cadru de testare, cum ar fi RiverIoT. Extensiile descriu relațiile și interacțiunile dintre dispozitivele IoT ale unei rețele. Descrierea poate fi folosită de fuzzer pentru a descoperi relații care pot duce la vulnerabilități în rețea.
- Am adăugat suport pentru execuția simbolică a binarelor COTS (comercial off-the-shelf) în River fuzzer. Acest lucru a fost realizat prin utilizarea GDB pentru a executa apeluri de sistem și funcții de bibliotecă, ceea ce reduce riscul de explozie a căii și reduce supraîncărcarea asociată cu reimplementarea acestor apeluri. Această abordare demonstrează, de asemenea, fezabilitatea emulării apelurilor de funcție de sistem și bibliotecă, menținând în același timp consistența între stările concrete și cele simbolice.

5.3 Lista de publicații

Reviste

Eduard Stăniloiu, Răzvan Nițu, Alexandru Militaru, Răzvan Deaconescu, *"Safer Linux Kernel Modules Using the D Programming Language"*, IEEE Access, doi: 10.1109/ACCESS.2022.3229461, 2022.

Eduard Stăniloiu, Răzvan Nițu, Răzvan Deaconescu, Răzvan Rughiniș, *"A New Collection Framework For the D Programming Language Standard Library"*, U.P.B. Scientific Bulletin Series C, Vol. 84, Iss. 3, Bucharest, Romania, 2022.

Eduard Stăniloiu, Rareș Ștefan Epure, Răzvan Deaconescu, Răzvan Nițu, Răzvan Rughiniș, *"Scalable Concolic Execution of COTS Binaries with the River Framework"*, accepted for publication at U.P.B. Scientific Bulletin Series C, Bucharest, Romania, 2022.

Răzvan Nițu, **Eduard Stăniloiu**, Răzvan Deaconescu, Răzvan Rughiniș, *"Designing Copy Construction for the D Programming Language"*, accepted for publication at U.P.B. Scientific Bulletin Series C, Bucharest, Romania, 2022.

Tarbă, N., Schmidt, D., Popovici, A.E., **Stăniloiu, E.**, Avatavului, C. and Prodan, M., *"On Performing Skew Detection and Correction Using Multiple Experts' Decision"*, Journal of Information Systems & Operations Management (ISSN 1843-4711), 2020.

Dicher, I.M., Ţurcus, A.G., Cojocea, E.M., Penariu, P.S., Bucur, I., Prodan, M. and **Stăniloiu, E.**, *"Unsupervised Merge of Optical Character Recognition Results"*, Journal of Information Systems & Operations Management, pp.60-67., 2020.

Conferințe

Răzvan Nițu, **Eduard Stăniloiu**, Răzvan Deaconescu, Răzvan Rughiniș, *"Adding Support for Reference Counting in the D Programming Language"*, Proceedings of the 17th International Conference on Software Technologies (ICSOFT), Lisbon, 2022.

Stăniloiu, E., Nitu, R., Becerescu, C. and Rughinis, R., *"Automatic Integration of D Code With the Linux Kernel"*. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet) (pp. 1-6). IEEE. 2021.

Staniloiu, E., Cristea, R. and Ghimis, B., *"IoT Fuzzing using AGAPIA and the River Framework"*. In ICSOFT (pp. 324-332). 2021.

Stăniloiu, E., Nitu, R., Aron, R. and Rughiniș, R., *"Extending Client-Server API Support for Memory Safe Programming Languages"*. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet) (pp. 1-5). IEEE. 2021.

Păduraru, Ciprian, Rareş Cristea, and **Eduard Stăniloiu**. *"RiverIoT-a Framework Proposal for Fuzzing IoT Applications"*. 2021 IEEE/ACM 3rd International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT). IEEE, 2021.

Nitu, R., **Stăniloiu, E.**, Done, C. and Rughinis, R., *"Security Audit for the D Programming Language"*. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet) (pp. 1-6). IEEE, 2021.

Nitu, R., **Stăniloiu, E.**, Creteanu, C. and Rughiniș, R., *"Building an Interface for the D Compiler Library"*. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet) (pp. 1-6). IEEE. 2021.

Lăpușteanu, A., Boiangiu, C.A., Tarbă, N., Voncilă, M.L., **Stăniloiu, C.E.** and Vlăsceanu, G.V., *"Improving Upon Photographic Steganography"*. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet) (pp. 1-6). IEEE. 2021.

Brevete

Costin-Anton Boiangiu, Giorgiana Violeta Vlăsceanu and **Constantin Eduard Stăniloiu**, *"Method for identifying connected components in binary images"*. Pending approval.

Registered at OSIM A/00599, 30.09.2021.

Bibliografie

[1] A. Bannister, "Substandard software costs us economy $2tn through security flaws, legacy systems, abandoned projects," *URL: https://portswigger.net/daily-swig/substandard-software-costs-us-economy-2tn-through-security-flaws-legacy-systems-abandoned-projects*, 2021.

[2] M. Dawson, D. N. Burrell, E. Rahim, and S. Brewster, "Integrating software assurance into the software development life cycle (sdlc)," *Journal of Information Systems Technology and Planning*, vol. 3, no. 6, pp. 49–53, 2010.

[3] "Check point research: Cyber attacks increased 50% year over year - check point software," https://blog.checkpoint.com/2022/01/10/check-point-research-cyber-attacks-increased-50-year-over-year/, accessed: 2021-10-20.

[4] J. Wurm, K. Hoang, O. Arias, A.-R. Sadeghi, and Y. Jin, "Security analysis on consumer and industrial iot devices," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 519–524.

[5] S. Li, L. Da Xu, and S. Zhao, "5g internet of things: A survey," *Journal of Industrial Information Integration*, vol. 10, pp. 1–9, 2018.

[6] K. V. English, I. Obaidat, and M. Sridhar, "Exploiting memory corruption vulnerabilities in connman for iot devices," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 247–255.

[7] G. Hernandez, O. Arias, D. Buentello, and Y. Jin, "Smart nest thermostat: A smart spy in your home," *Black Hat USA*, no. 2015, 2014.

[8] "Nvd - cvss severity distribution over time," https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time, accessed: 2021-10-20.

[9] "Iot connected devices worldwide 2019-2030 | statista," https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide, accessed: 2021-10-20.

[10] A. Radovici, I. Culic, D. Rosner, and F. Oprea, "A model for the remote deployment, update, and safe recovery for commercial sensor-based iot systems," *Sensors*, vol. 20, no. 16, p. 4393, 2020.

[11] I.-M. Stan, D. Rosner, and Ș.-D. Ciocîrlan, "Enforce a global security policy for user access to clustered container systems via user namespace sharing," in *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. IEEE, 2020, pp. 1–6.

[12] G. Keizer, “Windows comes up third in os clash two years early https://www.computerworld.com/article/3050931/windows-comes-up-third-in-os-clash-two-years-early.html,” 2016.

[13] Z. Schuermann and K. Guha, “Linux device drivers in rust https://zachschuermann.com/static/6118.pdf,” 2020.

[14] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers.* ” O’Reilly Media, Inc.”, 2005.

[15] D. Goodin, “Unpatched linux bug may open devices to serious attacks over wi-fi https://arstechnica.com/information-technology/2019/10/unpatched-linux-flaw-may-let-attackers-crash-or-compromise-nearby-devices,” 2019.

[16] H. Ed-Douibi, J. L. Cánovas Izquierdo, and J. Cabot, “Example-driven web api specification discovery,” in *European Conference on Modelling Foundations and Applications.* Springer, 2017, pp. 267–284.

[17] “Operating system family / Linux | TOP50,” https://www.top500.org/statistics/details/osfam/1/, accessed: 2022-04-17.

[18] W3Techs, “Linux vs. Windows usage statistics for websites,” https://w3techs.com/technologies/comparison/os-linux,os-windows, accessed: 2022-04-17.

[19] “Mobile operating system market share worldwide,” https://gs.statcounter.com/os-market-share/mobile/worldwide, accessed: 2022-04-17.

[20] AspenCore, “Mobile operating system market share worldwide,” https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf, nov 2019, accessed: 2022-04-17.

[21] “Linux kernel CVEs,” https://www.linuxkernelcves.com/, accessed: 2022-04-17.

[22] “Kernel self-protection,” https://www.kernel.org/doc/html/latest/security/self-protection.html, accessed: 2022-04-17.

[23] A. Popov, “Linux kernel defence map,” https://github.com/a13xp0p0v/linux-kernel-defence-map, accessed: 2022-04-17.

[24] “D programming language,” https://dlang.org/, accessed: 2022-04-17.

[25] “virtio,” https://wiki.libvirt.org/page/Virtio, 2022, accessed: 2022-04-17.

[26] A. Alexandrescu, *The D programming language.* Addison-Wesley Boston, MA, 2010.

[27] “Project highlight: Dpp,” https://dlang.org/blog/2019/04/08/project-highlight-dpp/, accessed: 2022-04-17.

[28] W. Bright, A. Alexandrescu, and M. Parker, “Origins of the d programming language,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 1–38, 2020.

[29] P. R. Wilson, “Uniprocessor garbage collection techniques,” in *International Workshop on Memory Management*. Springer, 1992, pp. 1–42.

[30] D. Gregor and S. Schupp, “Making the usage of stl safe,” in *Generic Programming*. Springer, 2003, pp. 127–140.

[31] A. Alexandrescu, “On Iteration,” http://www.informit.com/articles/printerfriendly/1407357, 2009.

[32] E. D. Berger, B. G. Zorn, and K. S. McKinley, “Composing high-performance memory allocators,” *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 114–124, 2001.

[33] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun, “99.44% pure: Useful abstractions in specifications,” in *ECOOP workshop on formal techniques for Java-like programs (FTfJP)*. Citeseer, 2004.

[34] U. W. Eisenecker, “Generative programming (gp) with c++,” in *Joint Modular Languages Conference*. Springer, 1997, pp. 351–365.

[35] D. L. Foundation, “What is d?” https://dlang.org/overview.html#what_is_d.

[36] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext transfer protocol–http/1.1,” 1999.

[37] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[38] D. Ashby and C. T. Jensen, *APIs for dummies*. Hoboken, New Jersey: John Wiley & Sons Inc., 2018.

[39] ProgrammableWeb, “Search the largest api directory on the web,” https://www.programmableweb.com/category/all/apis.

[40] Mozilla, “Working with json,” https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON.

[41] I.-M. Culic and A. Radovici, “Development platform for building advanced internet of things systems,” in *2017 16th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. IEEE, 2017, pp. 1–5.

[42] RedHat, “What is a rest api?” https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[43] L. Richardson and S. Ruby, *RESTful web services*. ” O’Reilly Media, Inc.”, 2008.

[44] D. Hardt *et al.*, “The oauth 2.0 authorization framework,” 2012.

[45] Google, "Using oauth 2.0 to access google apis," https://developers.google.com/identity/protocols/oauth2.

[46] Google, "Google api discovery service," https://developers.google.com/discovery.

[47] R. Aron, "D google drive client library," https://github.com/Robert-Aron293/d-google-drive-client.

[48] R. Aron, "D google mail client library," https://github.com/Robert-Aron293/d-google-gmail-client.

[49] C. I. Paduraru, R. Cristea, and E. Staniloiu, "Riveriot - a framework proposal for fuzzing iot applications," *International Conference on Software Engineering ICSE 2021, Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*, p. to appear, 2021.

[50] C. I. Paduraru, "Dataflow programming using agapia," in *2014 IEEE 13th International Symposium on Parallel and Distributed Computing*. IEEE, 2014, pp. 87–94.

[51] P. Leonardo, "Child programming: an adequate domain specific language for programming specific robots," 2013.

[52] M. Boshernitsan and M. S. Downes, *Visual programming languages: A survey*. Citeseer, 2004.

[53] C. Paduraru, "Research on agapia language, compiler and applications," *Ph. D. dissertation*, 2015.

[54] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[55] I. T. Banu-Demergian, C. Paduraru, and G. Stefanescu, "A new representation of two-dimensional patterns and applications to interactive programming," in *International Conference on Fundamentals of Software Engineering*. Springer, 2013, pp. 183–198.

[56] C. I. Paduraru and G. Stefanescu, "Adaptive virtual organisms: A compositional model for complex hardware-software binding," *Fundamenta Informaticae*, vol. 173, no. 2-3, pp. 139–176, 2020.

[57] G. Stefanescu and C. I. Paduraru, "Self-assembling heterogeneous interactive systems," in *Proceedings of the International Colloquium on Software-intensive Systems-of-Systems at 10th European Conference on Software Architecture*, 2016, pp. 1–7.

[58] B. Ghimis, M. Paduraru, and A. Stefanescu, “River 2.0: an open-source testing framework using ai techniques,” in *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing*, 2020, pp. 13–18.

[59] C. Paduraru, M.-C. Melemciuc, and M. Paduraru, “Automatic test data generation for a given set of applications using recurrent neural networks,” in *Software Technologies*, M. van Sinderen and L. A. Maciaszek, Eds. Cham: Springer International Publishing, 2019, pp. 307–326.

[60] C. Paduraru. and M. Melemciuc., “An automatic test data generation tool using machine learning,” in *Proceedings of the 13th International Conference on Software Technologies - Volume 1: ICSOFT,*, INSTICC. SciTePress, 2018, pp. 472–481.

[61] C. Paduraru, M. Paduraru, and A. Stefanescu, “Optimizing decision making in concolic execution using reinforcement learning,” in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020, pp. 52–61.

[62] B. Feng, A. Mera, and L. Lu, “P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1237–1254.

[63] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, “Halucinator: Firmware re-hosting through abstraction layer emulation,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1201–1218.

[64] C. Paduraru, M.-C. Melemciuc, and A. Stefanescu, “A distributed implementation using apache spark of a genetic algorithm applied to test data generation,” in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, pp. 1857–1863.

[65] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, “From hack to elaborate technique—a survey on binary rewriting,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–37, 2019.

[66] S. Dinesh, N. Burow, D. Xu, and M. Payer, “Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1497–1511.

[67] P. O’sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, “Retrofitting security in cots software with binary rewriting,” in *Ifip International Information Security Conference*. Springer, 2011, pp. 154–172.

[68] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, “Ramblr: Making reassembly great again.” in *NDSS*, 2017.

[69] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[70] D. Bruening and Q. Zhao, "Building dynamic tools with dynamorio on x86 and arm," 2017.

[71] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 2003, pp. 265–275.

[72] N. Voss, "afl-unicorn: Fuzzing arbitrary binary code," *Hacker Noon*, 2017.

[73] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[74] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[75] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of model checking*. Springer, 2018, pp. 305–343.

[76] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[77] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[78] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[79] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[80] "Lief : Library to instrument executable formats," https://lief-project.github.io/doc/latest/intro.html, accessed: 2021-10-20.

[81] F. Saudel and J. Salwan, "Triton: Concolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2015.

[82] N. A. Quynh, "Capstone: Next-gen disassembly framework," *Black Hat USA*, vol. 5, no. 2, pp. 3–8, 2014.

[83] "Gdb: The gnu project debugger," https://www.sourceware.org/gdb/documentation/, accessed: 2021-10-20.

[84] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[85] C. Paduraru, B. Ghimis, and A. Stefanescu, "Riverconc: An open-source concolic execution engine for x86 binaries." in *ICSOFT*, 2020, pp. 529–536.

[86] "nodejs/http-parser: Http request/response parser for messages written in c," https://github.com/nodejs/http-parser, accessed: 2021-10-20.

[87] "fuzzstati0n/fuzzgoat: A vulnerable c program for testing fuzzers," https://github.com/fuzzstati0n/fuzzgoat, accessed: 2021-10-20.

[88] "google/re: Re2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in pcre, perl, and python. it is a c++ library." https://github.com/google/re2, accessed: 2021-10-20.

[89] "libarchive/libarchive: Multi-format archive and compression library," https://github.com/libarchive/libarchive, accessed: 2021-10-20.

[90] "Fuzzbench: Fuzzer benchmarking as a service," https://google.github.io/fuzzbench/, accessed: 2021-10-20.

[91] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1393–1403.