FOI MEMO

Projekt/Project
Tekniker för att identifiera mjukvarusårbarheter

Sidnr/Page no
1 (20)

Projektnummer/Project no
E38549

Uppdragsgivare/Client
Försvarsmakten

FoT-område
Operationer i cyberdomänen

Författare/Author
Casper Jensen, Christian Vestlund, Christian Gustavsson, Viktor Andersson, Lovisa Nyholm, Daniel Eidenskog

Datum/Date
2024-12-04

Memo nummer/Number
FOI Memo 8673

Inventering av testfall för verktyg som identifierar mjukvarusårbarheter

1 Inledning

Att analysera kod och testa mjukvara för att identifiera sårbarheter är en resurskrävande aktivitet. Inte sällan är det också ett manuellt arbete som baseras på kunskap och erfarenhet hos den som gör analysen. Med en mjukvarusårbarhet avses en brist som kan exploateras av en hotaktör, antingen avsiktligt eller oavsiktligt, och som resulterar i en oönskad händelse som påverkar säkerheten negativt i det system där mjukvaran används.

Forskning om tekniker och verktyg som kan identifiera mjukvarusårbarheter är ett aktivt forskningsområde med en lång historik. Ett tidigt verktyg var Lint som introducerades 1978 [1]. Varianter av linters, exempelvis PC-lint[1], används än idag av mjukvaruutvecklare.

Utvecklingen av analysverktyg pågår kontinuerligt med vidareutveckling av klassiska verktyg samtidigt som nya verktyg introduceras. Hybridverktyg, som kombinerar flera olika tekniker, och verktyg baserade på maskininlärning har varit ett stort fokus för forskningen under de senaste fem åren [2]. Parallellt med utveckling av verktyg uppstår också frågan om hur deras förmågor och gränser kan utvärderas. För det behövs testfall, datamängder som samlar sårbar kod, som kan användas för att utvärdera tekniker och verktyg som identifierar mjukvarusårbarheter.

Det finns ett stort behov av realistiskt implementerade sårbarheter för att utvärderingar ska kunna genomföras i en så verklighetstrogen testmiljö som möjligt. För att möta behovet har en rad aktörer tagit initiativ till att producera datamängder med sårbarheter och verktyg som kan generera sårbar kod eller skriva om given kod så att sårbarheter introduceras i den. Två sådana aktörer är amerikanska NIST, National Institute of Standards and Technology, och OWASP, Open Web Application Security Project.

Befintlig forskning är inte konsekvent i hur datamängder med sårbarheter och verktyg för att producera sårbar kod används för att utvärdera kodanalysverktyg [2]. Olika datamängder, testsviter och produktionsmjukvaror används på ett sätt som gör det svårt att jämföra resultat från olika tester. Initiala sökningar visar också att få vetenskapliga publikationer diskuterar eller utvärderar användningen av datamängder.

Detta memo presenterar en inventering av datamängder med sårbar kod samt verktyg som kan producera sårbar kod. En ytlig analys av dem genomförs utifrån den vetenskapliga litteratur som finns att tillgå kring deras användning. Arbetet ingår i ett projekt som syftar till att bygga kunskap om olika tekniker och verktyg som kan användas för att identifiera mjukvarusårbarheter.

[1] https://pclintplus.com/

1.1 Syfte och mål

Memots syfte är att bygga kunskap kring offentligt tillgängliga samlingar av testfall som används vid utvärdering av tekniker och verktyg som identifierar mjukvarusårbarheter. Dessa testfall kan bestå av färdiga datamängder eller syntetiskt skapade av verktyg.

Målet med detta memo är följande:

- Göra en inventering av datamängder med sårbar kod som används för att testa verktyg som identifierar mjukvarusårbarheter.
- Göra en inventering av verktyg som kan producera sårbar kod för testning av verktyg som identifierar mjukvaråsårbarheter.

1.2 Avgränsningar

Inventeringen som redovisas i detta memorandum gör inte anspråk på att vara heltäckande utan ger en översiktlig bild av offentligt tillgängliga datamängder.

2 Metod

Detta memo är resultatet av en explorativ litteraturstudie som främst utgår från akademisk litteratur, men som även tar med källor utanför den akademiska sfären, såsom Github.

Utgångspunkten för arbetet är FOI-rapporten *Tekniker och verktyg som identifierar mjukvarusårbarheter* av Gustavsson m.fl. [2]. Den är en systematisk litteraturstudie som har skannat av forskning publicerad mellan 2014 och 2024. I rapporten presenteras 237 verktyg, varav 7 bedöms som verktyg med hög mognadsgrad. Många av de verktyg som publicerats i forskningen under de senaste tio åren har utvärderas på datamängder med sårbarheter eller andra typer av kodbaser. Material från arbetet med rapporten har använts också som underlag för detta memo.

Då FOI-rapporten fokuserar på tekniker och verktyg, snarare än på datamängder för utvärdering, har kompletterande explorativa sökningar gjorts för detta memo. Sökningarna har gjorts i sökmotorn Google samt i Google Scholar för att hitta ytterligare datamängder med testfall samt verktyg som kan generera sårbar kod.

3 Bakgrund

Detta kapitel ger en översiktlig beskrivning av terminologi som vanligen används för att beskriva mjukvarusårbarheter. Kapitlet ger också en översikt av kodanalysverktyg och hur de utvärderas i akademisk litteratur.

3.1 Kategorisering av mjukvarusårbarheter

Sårbarheter brukar ofta kategoriseras enligt taxonomin Common Weakness Enumeration (CWE), som hålls uppdaterad av MITRE.[2] CWE-kategoriseringen innehåller generella beskrivningar av sårbarheter. Exempelvis har ”CWE-787: Out-of-bounds Write” följande beskrivning:[3]

> *Produkten skriver data efter slutet, eller före starten, av den avsedda bufferten.*

I praktiken förekommer mjukvarusårbarheter i all mjukvara och i många olika former. Medan CWE listar sårbarheter på en konceptuell nivå finns andra databaser som listar konkreta sårbarheter som upptäckts i mjukvaruprodukter [3]. Det är framförallt två databaser som är relevanta för det här arbetet. Den ena är NVD, National Vulnerability Database, som underhålls av amerikanska NIST.[4] Den andra är CVE List[5] som underhålls av MITRE. Gemensamt för de båda databaserna är att de konkreta sårbarheterna benämns med ett CVE-ID, ett unikt identitetsnummer i listan av Common Vulnerability and Exposures. CVE är ett väletablerat sätt att hänvisa till specifika sårbarheter. Exempel på välkända sårbarheter är CVE-2014-0160 (Heartbleed)[6] och CVE-2021-44228 (Log4Shell)[7].

3.2 Kodanalysverktyg

För att identifiera sårbarheter vid utveckling av mjukvara används ett antal olika tillvägagångssätt, som exempelvis testning och kodanalys. Branschstandarder såsom IEC 61508 [4] och ISO/SAE 21434 [5] förespråkar ett antal tillvägagångssätt för mjukvaruverifiering där kodanalys spelar en viktig roll. Vanligt förekommande kodanalysmetoder i standarder är:

- **Statisk analys.** Med statisk analys undersöks källkod, binärkod eller intermediära representationer (eng. intermediate representation) utan att behöva exekvera mjukvara. Statiska kodanalysverktyg varierar i komplexitet i vilka typer av sårbarheter de kan identifiera. En vanlig slutsats i tidigare litteratur är att det sällan räcker med ett verktyg för att få täckning av alla sorters sårbarheter [6].

[2] https://cwe.mitre.org
[3] https://cwe.mitre.org/data/definitions/787.html (författarnas översättning).
[4] https://nvd.nist.gov/
[5] https://cve.org
[6] https://www.cve.org/CVERecord?id=CVE-2014-0160
[7] https://www.cve.org/CVERecord?id=CVE-2021-44228

- **Dynamisk analys.** Med dynamisk analys undersöks vad som händer i mjukvara under exekvering. Mjukvara exekveras i en kontrollerad miljö och data från exekveringen samlas in för att sedan analyseras. En svårighet med dynamisk analys är att uppnå en hög nivå av kodtäckning, dvs. hur stor andel av koden som testas [7].
- **Formell verifiering.** Formell verifiering används för att bevisa egenskaper hos en mjukvara. Generellt anses formell verifiering vara svåranvänt. Framförallt gäller det om källkoden inte utvecklats för formell verifering [8]. Även mer komplex mjukvara såsom virtualiseringslösningar [9] går att analysera med formell verifiering. Det är vanligt att det görs antaganden och förenklingar för att underlätta för formella verifieringar [8].
- **Fuzzning.** Fuzzning är ett specialfall av dynamisk analys där ogiltig, oväntad eller slumpmässig indata genereras (med varierande systematik) för att få mjukvara att krascha eller generera oväntat beteende på grund av potentiella sårbarheter [10]. Fuzzning är en välanvänd kodanalysmetod där det finns ett stort driv för att utveckla verktyg, såsom American Fuzzy Loop.[8]

Vilken metod och vilka verktyg som är lämpliga att använda är inte en enkel fråga. Olika verktyg har olika för- och nackdelar, vilket bland annat påpekats i en tidigare studie av Karlzén m.fl. [3]. För att utvärdera kodanalysverktyg behövs testfall med sårbar kod för verktygen att analysera i en kontrollerad miljö.

Som nämndes i avsnitt 3.1 finns många olika samlingar, databaser och kategoriseringar av kända sårbarheter. Att använda sig av befintlig mjukvara eller specifikt konstruerade testfall för att påvisa effektiviteten hos analysverktyg är därför vanligt förekommande [11–14]. Linux och OpenSSL är exempel på kända mjukvaror som använts i akademisk litteratur [13]. Ett uppenbart problem med att använda sig av befintliga mjukvaror är att de riskerar att bli uppdaterade och ändras, vilket kan försvåra jämförelser mellan verktyg.

Det finns exempel på publikationer som undersökt hur verktyg har utvärderats. Exempelvis studerade Klees m.fl. hur olika fuzzers utvärderades [15]. Författarna pekar tydligt ut ett behov av systematisk metodologi för att utvärdera fuzzers, men samma behov finns för alla typer av kodanalysverktyg. Vetenskapliga publikationer nyttjar olika datamängder, exempelvis Juliet som är en del av SARD (se tabell 4.1),[9] för att påvisa kodanalysverktygs effektivitet att hitta olika typer av sårbarheter [16, 17]. Många publikationer nämner användandet av datamängder med sårbarheter och kommenterar på användandet av testfall. Men det finns få publikationer som fokuserar på att utvärdera egenskaper hos datamängderna.

[8] https://github.com/google/AFL
[9] https://samate.nist.gov/SARD/test-suites/112

4 Resultat

I detta kapitel presenteras inventeringen av testfall med mjukvarusårbarheter. Från inventeringen kunde två typer av datamängder identifieras: konstruerade datamängder med kod som innehåller sårbarheter samt verktyg som kan skapa sårbar kod. Inventeringen i detta kapitel är därför uppdelad i två delar. Den första delen består av färdiga datamängder med sårbar kod, med antingen verkliga eller konstruerade exempel. Den andra delen består av verktyg som kan generera sårbar kod, antingen från grunden eller genom att modifiera befintlig källkod.

4.1 Datamängder med sårbarheter

I tabell 4.1 redovisas en översikt av datamängder som innehåller sårbarheter. Sårbarheter som är kopplade till Androidmjukvara visas istället i tabell 4.2. I tabell 4.3 listas datamängder med sårbarheter för att prestandamätning av specifikt fuzzers. Alla identifierade datamängder är utgivna av forskare på universitet, stora företag eller statliga organisationer. I de fall datamängderna beskrivs i vetenskapliga publikationer refereras både publikationen och själva datamängden, exempelvis med länk till GitHub.

Några datamängder uppdateras kontinuerligt med fler sårbarheter, medan andra uppdaterats väldigt sällan eller inte alls sedan deras introduktion. Av de olika datamängderna i tabell 4.1 till 4.3 är det endast SARD, MegaVul, CVEfixes, PrimeVul och Fuzzbench som har fått uppdateringar från juli 2024 och framåt.

Innehållet som anges för datamängderna är de som uppges av respektive utgivare. Det finns dock betydande skillnader i hur detta anges för respektive datamängd. Vissa skapare anger sårbarheterna som mängd testfall medan andra anger i antal funktioner, vissa anger antal sårbarhetstyper medan andra inte säger något om det alls. Som skapare av verktygen anges universitet eller organisation.

Tabell 4.1: Datamängder med sårbarheter.

Namn	**Program-språk**	**Senast uppda-terad**	**Innehåll**	**Skapare**
CVEfixes	Flera olika språk[10]	2024	ca 11 800 CVE:er i ca 138 000 funktioner; 272 olika CWE:er	Simula Research Laboratory [18, 19]
IoTVulCode	C, C++	2024	130 000+ sårbara kodexempel och funktioner; 10+ olika CWE:er	HK [20, 21]
MegaVul	C, C++	2024	17 000+ sårbarheter insamlade mellan 2003-2023	ZJU, CUFE [22, 23]
OWASP Benchmark	Java	2024	< 3000 testfall; 11 olika CWE:er	OWASP [24]
PrimeVul	C, C++	2024	ca 7000 sårbara funktioner; 140+ olika CWE:er	CU, UW, KACST, Google Deepmind, UC Berkley, UMD [25, 26]
ReposVul	C, C++, Java, Python	2024	6100+ testfall; 236 olika CWE:er	HIT [27, 28]
SARD	C, C++, Java, PHP, C#	2024	450 000+ testfall; 150 olika CWE:er	NIST [29]
BigVul	C, C++	2021	3 700+ sårbarheter; 91 olika CWE:er	NJIT, UTD [30, 31]
D2A	C, C++	2021	ca 11 000 testfall med både sårbar kod och åtgärdade sårbarheter	IBM [32, 33]
BugHunter	Java	2020	ca 160 000 metoder; oklar mängd sårbara metoder och CWE:er	USZ [34, 35]
Devign	C, C++	2019	ca 23 000 testfall[11]	NTU [36, 37]
Unified Bug Dataset	Java	2019	ca 47 000 klasser och filer; oklar mängd CWE:er	USZ [38, 39]
Draper	C, C++	2018	ca 1,2 miljoner icke sårbara- och ca 87 000 sårbara funktioner; oklar mängd CWE:er	Draper, BU [40, 41]

[10] CVEfixes stödjer flera programmeringsspråk men exakt vilka framgår inte av publikationen [18].
[11] Sårbarheter som är extraherade från commits som åtgärdar sårbarheter.

Datamängderna i tabell 4.1 består till majoriteten av testfall som baseras på C och C++. Några datamängder innehåller Java, medan enbart ett fåtal innehåller testfall för Python och C#. En potentiell anledning kan vara att C och C++ är minnesosäkra programspråk och innehåller sårbarheter såsom buffertöverskridningar. Minnesrelaterade buggar förekommer inte i lika stor utsträckning i minnessäkra språk, såsom Java och C#, vilket kan ha en påverkan på utvecklingen av datamängder.

Tabell 4.2: Datamängder med sårbarheter relaterade till Android-applikationer.

Namn	**Programspråk**	**Senast uppdaterad**	**Innehåll**	**Skapare**
LVDAndro	Android-applikationer	2023	ca 26 miljoner sårbara kodexempel; 23 olika CWE:er	RGU, UoK, BCU, MDX [42, 43]
Ghera	Java (Android)	2019	60 olika testfall för prestandamätning	KSU [44, 45]

I tabell 4.3 presenteras en inventering av testsamlingar (eng. benchmarks) för prestandamätning som specifikt används för att testa fuzzers. Till skillnad från datamängderna i tabell 4.1 och 4.2 är dessa testsamlingar specifikt till för att jämföra fuzzers förmåga att identifiera sårbarheter. Vissa av dessa samlingar, exempelvis Fuzzbench, är utformade för att man ska kunna testa förbestämda fuzzers som redan är integrerade med testplattformen. Dock finns det möjlighet att lägga till stöd för andra fuzzers. En annan skillnad mot datamängderna är att dessa testsamlingar är mer stängda än datamängderna när det kommer till innehåll, alltså storlek och utformning av testfall, samt sårbarhetstyper som testas.

Tabell 4.3: Testsamling för prestandamätning (eng. benchmarking) av fuzzers.

Namn	**Senast uppdaterad**	**Skapare**
Fuzzbench	2024	Google [46]
DataRaceBench	2023	Lawrence Livermore National Laboratory [47, 48]
mua-fuzzer-benchmark	2023	CISPA, RUB, USYD [49, 50]
Magma	2022	EPFL, ANU [51, 52]

4.2 Verktyg för att generera sårbar kod

I tabell 4.4 presenteras verktyg framtagna för att generera sårbar kod. Genereringen sker antingen från grunden eller utifrån befintlig källkod som verktyget injicerar sårbar kod i. När generering av sårbar kod sker från grunden betyder det att verktyget skriver helt nya färdiga testfall som innehåller sårbar kod.

Tabell 4.4: Verktyg som genererar sårbar kod.

Namn	**Programspråk**	**Senast uppda-terad**	**Skapare**
LAVA	C	2024	NYU, MIT Lincon Laboratory, NEU [53, 54]
Vulnerability Test Suite Generator (VTSG)	PHP, C#, Python, stöd för fler	2024	NIST [55, 56]
IntJect	C, C++	2022	UNamur, uni.lu [57, 58]
Learning Realistic Bugs	Java, JavaScript, Python	2022	Uni Oldenburg [59, 60]
Apocalypse	(okänt)[12]	2018	IIT Kanpur, NYU Tandon [61]
EvilCoder	C	2016	RUB [62, 63]

[12] Skaparna av Apocalypse konstaterar aldrig exakt vilket språk som verktyget är anpassat mot. Verktyget är dock byggt på bland annat Clang, vilket är ett kompilatorgränssnitt för C och C++ [61].

5 Diskussion

I detta kapitel diskuteras egenskaper och problem som identifierats hos datamängder med sårbarheter och verktyg för att generera sårbar kod.

5.1 Kritik mot datamängder av sårbarheter

Detta avsnitt går översiktligt igenom kritiken som riktats mot datamängder med sårbarheter. Flera av datamängderna som presenteras i avsnitt 4.1 har kritiserats på ett eller annat sätt. Flera studier påpekar problem med bland annat markering av sårbarheter, obalans mellan olika typer av sårbarheter, kvalitet och komplexitet i kod.

5.1.1 Dålig eller felaktig märkning av sårbarheter

I ett antal publikationer påpekas att datamängder med sårbarheter har dålig eller rent av felaktig märkning av sårbara funktioner och icke sårbara funktioner [25, 64–66]. Utöver kritik mot bristfälliga markeringar finns också kritik mot att många datamängder saknar tydliga märkning av exempelvis CWE-ID, CVE-ID och sårbarhetsbeskrivningar [67].

5.1.2 Obalans i sårbarhetstyper

En faktor som påverkar kvaliteten hos datamängder är att det finns en obalans mellan olika sårbarhetstyper (CWE:er) [64, 68]. Vissa datamängder innehåller endast några få sårbarhetstyper medan andra datamängder innehåller många olika typer. Det finns dessutom en stor skillnad i andelen sårbarhetstyper i datamängderna jämfört med hur det ser ut i verkligheten [69]. Många datamängder speglar inte proportionerna för de sårbarheter som har rapporterats och finns representerade i sårbarhetsdatabaserna.

I en publikation som jämför statiska kodanalysverktyg diskuterar Esposito m.fl. potentiella problem i datamängden Juliet [70]. Bland annat omnämns risker med att vissa sårbarheter kan vara under- eller överrepresenterade, att vissa specifika typer av sårbarheter saknas och att realismen är undermålig.

5.1.3 Variation i kodkvalitet

En kritik som riktas mot datamängder med sårbarheter är att de håller en låg kvalitet [25]. Med låg kvalitet avses att kod saknas för att göra funktioner och även sårbarheter kompletta [65]. Det förekommer också datamängder där många filer är tomma och helt saknar kod [64] samt datamängder med duplicerad kod, där samma kod och sårbarheter förekommer flera gånger [25, 65].

5.1.4 Olika komplexitet i sårbarheter

Något som kan observeras i de identifierade datamängderna är olikheter i hur sårbarheterna är uppbyggda samt hur de skiljer sig i komplexitet. Skillnader i komplexitet av sårbarheter inom och mellan datamängder har tidigare påpekats av forskare [68].

Datamängder som skapas i syfte att inkludera specifika sårbarheter tar inte nödvändigtvis hänsyn till programkonstruktioner som hindrar eller försvårar för kodanalysverktyg att identifiera sårbarheter. I praktiken kan det finnas sårbarheter som endast behöver en rad kod för att introduceras, medan andra kan behöva sträcka sig mellan många rader kod och eventuellt mellan olika filer för att tillsammans åstadkomma en sårbarhet.

I produktionsmjukvara kan olika komplexitet och storlekar på sårbarheter observeras då karaktäristiken för sårbarheter varierar stort [68]. Ett sätt att potentiellt mäta komplexitet i testfall är att titta på antalet kodrader per sårbara kodblock. Här skiljer sig många datamängder mellan varandra, där vissa datamängder har ett fåtal rader i snitt, medan andra har hundratals [64].

En studie har påpekat att konstruktionen hos datamängderna är en viktig faktor att ta hänsyn till [67]. Specifikt pekar studien på mängden effektiv information i varje sårbarhetsinstans, exempelvis hur mycket kod, filer och funktioner som inkluderas för varje sårbarhet. Om det är för mycket normal kod utan sårbarheter gentemot sårbar kod för varje sårbarhet kan det försvåra lärandet hos maskininlärningsbaserade verktyg [67]. Det kan skada inlärningen för verktyget eftersom det kan bli svårt att faktiskt veta vilken del av koden som är kritisk för sårbarheten och att verktyget riskerar därmed att lära sig fel.

En risk med datamängder som enbart innehåller sårbarheter i simpla programkonstruktioner är att utvärderingar av verktyg med sådana datamängder kan resultera i bra resultat. Det kan leda till att utvecklare använder verktyg i en falsk trygghet att de kan hitta specifika typer av sårbarheter även i komplex mjukvara.

5.2 Verktyg för att generera sårbar kod

Inom forskning har verktyg som kan generera sårbar kod agerat komplement eller ersättning till datamängder med sårbarheter. Problemet med att generera sårbar kod är att få verktyg finns att tillgå. Stödet för andra programspråk än C och C++ är begränsat, vilket också är de programspråk som är vanligast förekommande i de identifierade färdiga datamängderna med sårbarheter. Bara VTSG och Learning Realistic Bugs har stöd för andra programspråk, som Python, Javascript och Java.

Verktygens mognad och framtida vidareutveckling är faktorer som påverkar hur relevanta de kommer att vara som komplement eller ersättning till datamängder med sårbarheter. Det finns begränsat med forskning om hur bra sårbarhetsgenererande verktyg är i jämförelse med färdiga datamängder som innehåller sårbarheter. Fyra av verktygen har uppdaterats inom de senaste två åren och två av dem under 2024.

NIST har genomfört utvärderingar av statiska kodanalysverktyg i SATE VI-projektet, där även verktyg som kan generera sårbar kod har utvärderats [71]. Verktyg som injicerar buggar i befintlig källkod har också studerats i andra arbeten, där det har konstaterats att de har svårigheter i vilka typer av sårbarheter de kan injecera [72]. Det finns konstaterade problem med att injicerade buggarna inte är lika svåra att hitta som verkliga sårbarheter [73], men också att mer forskning behövs hur bra sårbarhetsgenererande verktyg är [74]. Verktyg som kan generera sårbar kod är något som behöver undersökas ytterligare, då tidigare arbeten som diskuteras i avsnitt 5.1 påvisar på att det finns brister hos existerande datamängder.

Användningen av verktyg som kan generera sårbar kod är inte lika omfattande som användningen av färdiga datamängder. På grund av svårigheter med att hitta riktiga buggar att använda i teständamål har verktyg som injicerar buggar i befintlig källkod dock börjat få större intresse hos forskare [72–74].

6 Slutsatser

I detta memo har 19 datamängder och testsamlingar identifierats samt sex olika verktyg som kan generera sårbar kod. Majoriteten av de identifierade datamängderna och verktygen är att betrakta som övergivna av sina utvecklare.

De olika datamängderna som innehåller sårbarheter är under omfattande kritik då flera studier identifierat problem såsom felaktig eller dålig markering av sårbarheter, obalans mellan sårbarhetstyper i datamängderna och verkligheten samt undermålig kodkvalitet. Utformningen av datamängderna har även kritiserats för att den gör det svårt för AI-baserade kodanalysverktyg att tränas tillräckligt bra då sårbarheterna inte är implementerade på realistiska sätt.

Verktyg som genererar sårbar kod, eller injicerar sårbarheter i redan existerande kodbaser, kan vara en lösning till problemet med orealistiska sårbarheter i datamängder och ett sätt att testa kodanalysverktyg i mer verkliga förhållanden. Ytterligare forskning behövs dock kring mognaden av dessa typer av verktyg och hur bra de står sig mot färdiga datamängder med sårbarheter.

Sammanfattat finns det ett behov av att undersöka vad som gör samlingar med sårbarheter bra. Även vilka krav som bör uppfyllas i konstruktionen av datamängder med sårbarheter för att effektivt testa eller träna kodanalysverktyg. Ytterliggare forskning behövs också på sårbarhetsgenererande verktyg och deras faktiska förmåga att producera testfall och träningsdata för kodanalysverktyg.

7 Referenser

[1] S. C. Johnson. *Lint, a C Program Checker*. Bell Laboratories, 1978.

[2] C. Gustavsson, C. Vestlund, V. Andersson, L. Nyholm, C. Jensen och D. Eidenskog. *Tekniker och verktyg som identifierar mjukvarusårbarheter*. FOI-R–5692–SE. Totalförsvarets forskningsinstitut, 2024.

[3] H. Karlzén, D. Eidenskog, J. Falkcrona och C. Valassi. *Varför har mjukvaror sårbarheter?* FOI-R–5550–SE. Totalförsvarets forskningsinstitut, 2023.

[4] *Funktionssäkerhet hos elektriska, elektroniska och programmerbara elektroniska säkerhetskritiska system – Del 3: Fordringar på programvara.* International standard. IEC 61508-3. International Electrotechnical Commission, 2011.

[5] *Vägfordon - Teknik för cybersäkerhet*. International standard. ISO/SAE 21434. International Organization for Standardization & SAE International, 2021.

[6] D. Stefanović, D. Nikolić, D. Dakić, I. Spasojević och S. Ristić. “Static code analysis tools: A systematic literature review”. I: *Proceedings of the 31st DAAAM International Symposium*. Vol. 31. 1. 2020, s. 565–573. DOI: https://doi.org/10.2507/31st.daaam.proceedings.078.

[7] O. Zaazaa och H. El Bakkali. “Dynamic vulnerability detection approaches and tools: State of the Art”. I: *2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS)*. 2020, s. 1–6. DOI: https://doi.org/10.1109/ICDS50568.2020.9268686.

[8] C. Bildsten, M. Cohen, D. Eidenskog och I. Rodhe. *Praktisk mjukvaruverifiering med formella metoder*. FOI-R–4642–SE. Totalförsvarets forskningsinstitut, 2018.

[9] C. Baumann, M. Näslund, C. Gehrmann, O. Schwarz och H. Thorsen. “A high assurance virtualization platform for ARMv8”. I: *2016 European Conference on Networks and Communications (EuCNC)*. 2016, s. 210–214. DOI: https://doi.org/10.1109/EuCNC.2016.7561034.

[10] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo och W. Liu. “A systematic review of fuzzing techniques”. I: *Computers & Security* 75 (2018), s. 118–137. DOI: https://doi.org/10.1016/j.cose.2018.02.002.

[11] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo och H. Cai. “PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems”. I: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, juni 2023, s. 1379–1396.

[12] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu och J. Sun. “Pata: Fuzzing with path aware taint analysis”. I: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, s. 1–17.

[13] Y. Lyu, Y. Fang, Y. Zhang, Q. Sun, S. Ma, E. Bertino, K. Lu och J. Li. “Goshawk: Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis”. I: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, s. 2096–2113. DOI: https://doi.org/10.1109/SP46214.2022.9833613.

[14] K. Kuszczyński och M. Walkowski. “Comparative Analysis of Open-Source Tools for Conducting Static Code Analysis”. I: *Sensors* 23.18 (2023), s. 7978.

[15] G. Klees, A. Ruef, B. Cooper, S. Wei och M. Hicks. “Evaluating fuzz testing”. I: *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2018, s. 2123–2138.

[16] N. Corteggiani, G. Camurati och A. Francillon. “Inception: System-Wide Security Testing of Real-World Embedded Systems Software”. I: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, aug. 2018, s. 309–326.

[17] T. Charest, N. Rodgers och Y. Wu. “Comparison of static analysis tools for Java using the Juliet test suite”. I: *11th International Conference on Cyber Warfare and Security*. 2016, s. 431–438.

[18] G. Bhandari, A. Naseer och L. Moonen. “CVEfixes: automated collection of vulnerabilities and their fixes from open-source software”. I: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2021. Athens, Greece: Association for Computing Machinery, 2021, s. 30–39. DOI: https://doi.org/10.1145/3475960.3475985.

[19] *CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software*. 2024. URL: https://github.com/secureIT-project/CVEfixes. Besökt: 2024-10-14.

[20] G. P. Bhandari, G. Assres, N. Gavric, A. Shalaginov och T.-M. Grønli. “IoTvulCode: AI-enabled vulnerability detection in software products designed for IoT applications”. I: *International Journal of Information Security* (2024). DOI: https://doi.org/10.1007/s10207-024-00848-6.

[21] *Dataset Description*. https://github.com/SmartSecLab/IoTvulCode/tree/main/data. Besökt: 2024-10-11.

[22] C. Ni, L. Shen, X. Yang, Y. Zhu och S. Wang. “MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations”. I: *Proceedings of the 21st International Conference on Mining Software Repositories*. MSR ’24. Lisbon, Portugal: Association for Computing Machinery, 2024, s. 738–742. DOI: https://doi.org/10.1145/3643991.3644886.

[23] *MegaVul*. 2023. URL: https://github.com/Icyrockton/MegaVul. Besökt: 2024-10-14.

[24] OWASP. *OWASP Benchmark*. 2024. URL: https://owasp.org/www-project-benchmark/. Besökt: 2024-10-08.

[25] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray och Y. Chen. “ Vulnerability Detection with Code Language Models: How Far Are We? ” I: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, s. 469–481.

[26] *PrimeVul: Vulnerability Detection with Code Language Models: How Far Are We?* 2024. URL: https://github.com/DLVulDet/PrimeVul/tree/main?tab=readme-ov-file. Besökt: 2024-10-14.

[27] *ReposVul*. 2024. URL: https://github.com/Eshe0922/ReposVul?tab=readme-ov-file. Besökt: 2024-10-08.

[28] X. Wang, R. Hu, C. Gao, X.-C. Wen, Y. Chen och Q. Liao. "ReposVul: A Repository-Level High-Quality Vulnerability Dataset". I: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. Lisbon, Portugal: Association for Computing Machinery, 2024, s. 472–483. DOI: https://doi.org/10.1145/3639478.3647634.

[29] NIST. *NIST Software Assurance Reference Dataset*. 2024. URL: https://samate.nist.gov/SARD/. Besökt: 2024-10-08.

[30] *A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries*. 2021. URL: https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset. Besökt: 2024-10-08.

[31] J. Fan, Y. Li, S. Wang och T. N. Nguyen. "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries". I: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR 20. New York, NY, USA: Association for Computing Machinery, 2020, s. 508–512. DOI: https://doi.org/10.1145/3379597.3387501.

[32] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari och Z. Su. "D2A: a dataset built for AI-based vulnerability detection methods using differential analysis". I: *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '21. Virtual Event, Spain: IEEE Press, 2021, s. 111–120. DOI: https://doi.org/10.1109/ICSE-SEIP52600.2021.00020.

[33] IBM. *D2A - Differential Analysis Dataset*. 2021. URL: https://developer.ibm.com/exchanges/data/all/d2a/. Besökt: 2024-10-15.

[34] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth och T. Gyimóthy. *BugHunter Dataset*. 2020. URL: https://data.mendeley.com/datasets/8tx7kjbkg4/2. Besökt: 2024-10-14.

[35] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth och T. Gyimóthy. "An automatically created novel bug dataset and its validation in bug prediction". I: *Journal of Systems and Software* 169 (2020), s. 110691. DOI: https://doi.org/10.1016/j.jss.2020.110691.

[36] Y. Zhou, S. Liu, J. Siow, X. Du och Y. Liu. "Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks". I: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[37] *Devign*. 2023. URL: https://github.com/epicosy/devign. Besökt: 2024-10-15.

[38] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket och T. Gyimóthy. “A public unified bug dataset for java and its assessment regarding metrics and bug prediction”. I: *Software Quality Journal* 28.4 (dec. 2020), s. 1447–1506. DOI: https://doi.org/10.1007/s11219-020-09515-0.

[39] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket och T. Gyimóthy. *Unified Bug Dataset*. https://www.inf.u-szeged.hu/~ferenc/papers/UnifiedBugDataSet/. Besökt: 2024-10-15.

[40] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood och M. McConley. *Draper VDISC Dataset - Vulnerability Detection in Source Code*. https://osf.io/d45bw/. Besökt: 2024-10-24.

[41] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood och M. McConley. “Automated Vulnerability Detection in Source Code Using Deep Representation Learning”. I: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2018, s. 757–762. DOI: https://doi.org/10.1109/ICMLA.2018.00120.

[42] J. Senanayake., H. Kalutarage., M. O. Al-Kadri., L. Piras. och A. Petrovski. “Labelled Vulnerability Dataset on Android Source Code (LVDAndro) to Develop AI-Based Code Vulnerability Detection Models”. I: *Proceedings of the 20th International Conference on Security and Cryptography - SECRYPT*. INSTICC. SciTePress, 2023, s. 659–666. DOI: https://doi.org/10.5220/0012060400003555.

[43] *LVDAndro (Labelled Vulnerability Dataset on Android Source Code)*. 2023. URL: https://github.com/softwaresec-labs/LVDAndro. Besökt: 2024-10-14.

[44] J. Mitra och V.-P. Ranganath. *android-app-vulnerability-benchmarks*. https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/src/master/. Besökt: 2024-10-14.

[45] J. Mitra och V.-P. Ranganath. “Ghera: A Repository of Android App Vulnerability Benchmarks”. I: *Proceedings of the 13th international conference on predictive models and data analytics in software engineering*. Nov. 2017. DOI: https://doi.org/10.1145/3127005.3127010.

[46] Google. *FuzzBench: Fuzzer Benchmarking As a Service*. 2024. URL: https://google.github.io/fuzzbench/. Besökt: 2024-10-14.

[47] C. Liao, P.-H. Lin, J. Asplund, M. Schordan och I. Karlin. “DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools”. I: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’17)*. Denver, Colorado: Association for Computing Machinery, 2017. DOI: https://doi.org/10.1145/3126908.3126958.

[48] *DataRaceBench 1.4.0*. 2023. URL: https://github.com/LLNL/dataracebench. Besökt: 2024-10-15.

[49] P. Görz, B. Mathis, K. Hassler, E. Güler, T. Holz, A. Zeller och R. Gopinath. “Systematic Assessment of Fuzzers using Mutation Analysis”. I: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA, USA, aug. 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/gorz.

[50] *mua-fuzzer-benchmark*. 2023. URL: https://github.com/CISPA-SysSec/mua_fuzzer_bench. Besökt: 2024-10-15.

[51] A. Hazimeh, A. Herrera och M. Payer. “Magma: A Ground-Truth Fuzzing Benchmark”. I: *Proc. ACM Meas. Anal. Comput. Syst.* 4.3 (nov. 2020). DOI: https://doi.org/10.1145/3428334.

[52] A. Hazimeh, A. Herrera och M. Payer. *Magma: A Ground-Truth Fuzzing Benchmark*. 2022. URL: https://hexhive.epfl.ch/magma/. Besökt: 2024-10-14.

[53] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich och R. Whelan. “LAVA: Large-Scale Automated Vulnerability Addition”. I: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, s. 110–121. DOI: https://doi.org/10.1109/SP.2016.15.

[54] *LAVA: Large Scale Automated Vulnerability Addition*. 2024. URL: https://github.com/panda-re/lava. Besökt: 2024-10-15.

[55] P. E. Black, W. Mentzer, E. Fong och B. Stivalet. *Vulnerability Test Suite Generator (VTSG) Version 3*. Okt. 2023. DOI: https://doi.org/10.6028/NIST.IR.8493.

[56] *Vulnerability Test Suite Generator (VTSG)*. 2024. URL: https://github.com/usnistgov/VTSG. Besökt: 2024-10-15.

[57] B. Petit, A. Khanfir, E. Soremekun, G. Perrouin och M. Papadakis. “IntJect: Vulnerability Intent Bug Seeding”. I: *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. 2022, s. 19–30. DOI: https://doi.org/10.1109/QRS57517.2022.00013.

[58] *IntJect: Vulnerability Intent Bug Seeding*. 2022. URL: https://github.com/Petit-Benjamin/VulnerabilityInjectionNLP. Besökt: 2024-10-15.

[59] C. Richter och H. Wehrheim. “Learning Realistic Mutations: Bug Creation for Neural Bug Detectors”. I: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2022, s. 162–173. DOI: https://doi.org/10.1109/ICST53961.2022.00027.

[60] *Learning Realistic Bugs: Bug Creation for Neural Bug Detectors*. 2022. URL: https://github.com/cedricrupb/ctxmutants. Besökt: 2024-10-15.

[61] S. Roy, A. Pandey, B. Dolan-Gavitt och Y. Hu. “Bug synthesis: challenging bug-finding tools with deep faults”. I: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, s. 224–234. DOI: https://doi.org/10.1145/3236024.3236084.

[62] J. Pewny och T. Holz. “EvilCoder: automated bug insertion”. I: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACSAC ’16. Los Angeles, California, USA: Association for Computing Machinery, 2016, s. 214–225. DOI: https://doi.org/10.1145/2991079.2991103.

[63] *EvilCoder*. 2016. URL: https://github.com/RUB-SysSec/EvilCoder. Besökt: 2024-10-15.

[64] R. Jain, N. Gervasoni, M. Ndhlovu och S. Rawat. “A Code Centric Evaluation of C/C++ Vulnerability Datasets for Deep Learning Based Vulnerability Detection Techniques”. I: *Proceedings of the 16th Innovations in Software Engineering Conference*. ISEC ’23. Allahabad, India: Association for Computing Machinery, 2023. DOI: https://doi.org/10.1145/3578527.3578530.

[65] R. Croft, M. A. Babar och M. M. Kholoosi. “Data Quality for Software Vulnerability Datasets”. I: *Proceedings of the 45th International Conference on Software Engineering*. ICSE ’23. Melbourne, Victoria, Australia: IEEE Press, 2023, s. 121–133. DOI: https://doi.org/10.1109/ICSE48619.2023.00022.

[66] X. Nie, N. Li, K. Wang, S. Wang, X. Luo och H. Wang. “Understanding and Tackling Label Errors in Deep Learning-Based Vulnerability Detection (Experience Paper)”. I: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, s. 52–63. DOI: https://doi.org/10.1145/3597926.3598037.

[67] Y. Lin, Y. Li, M. Gu, H. Sun, Q. Yue, J. Hu, C. Cao och Y. Zhang. “Vulnerability Dataset Construction Methods Applied To Vulnerability Detection: A Survey”. I: *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2022, s. 141–146. DOI: https://doi.org/10.1109/DSN-W54100.2022.00032.

[68] S. A. Atiiq, C. Gehrmann, K. Dahlén och K. Khalil. *From Generalist to Specialist: Exploring CWE-Specific Vulnerability Detection*. 2024. URL: https://arxiv.org/abs/2408.02329.

[69] K. N. Afanador och C. E. Irvine. “Representativeness in the benchmark for vulnerability analysis tools (B-VAT)”. I: *Proceedings of the 13th USENIX Conference on Cyber Security Experimentation and Test*. CSET’20. USA: USENIX Association, 2020.

[70] M. Esposito, V. Falaschi och D. Falessi. “An extensive comparison of static application security testing tools”. I: *arXiv preprint arXiv:2403.09219* (2024).

[71] A. Delaitre, P. E. Black, D. Cupif, G. Haben, L. Alex-Kevin, V. Okun, Y. Prono och A. Delaitre. *SATE VI Report: Bug Injection and Collection*. Juni 2023. DOI: https://doi.org/10.6028/NIST.SP.500-341.

[72] M. Cho, H. Jin, D. An och T. Kwon. “Evaluating Code Coverage for Kernel Fuzzers via Function Call Graph”. I: *IEEE Access* 9 (2021), s. 157267–157277. DOI: https://doi.org/10.1109/ACCESS.2021.3129062.

[73] J. Bundt, A. Fasano, B. Dolan-Gavitt, W. Robertson och T. Leek. “Evaluating Synthetic Bugs”. I: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ASIA CCS ’21. Virtual Event, Hong Kong: Association for Computing Machinery, 2021, s. 716–730. DOI: https://doi.org/10.1145/3433210.3453096.

[74] M. Böhme, L. Szekeres och J. Metzman. “On the reliability of coverage-based fuzzer benchmarking”. I: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, s. 1621–1633. DOI: https://doi.org/10.1145/3510003.3510230.