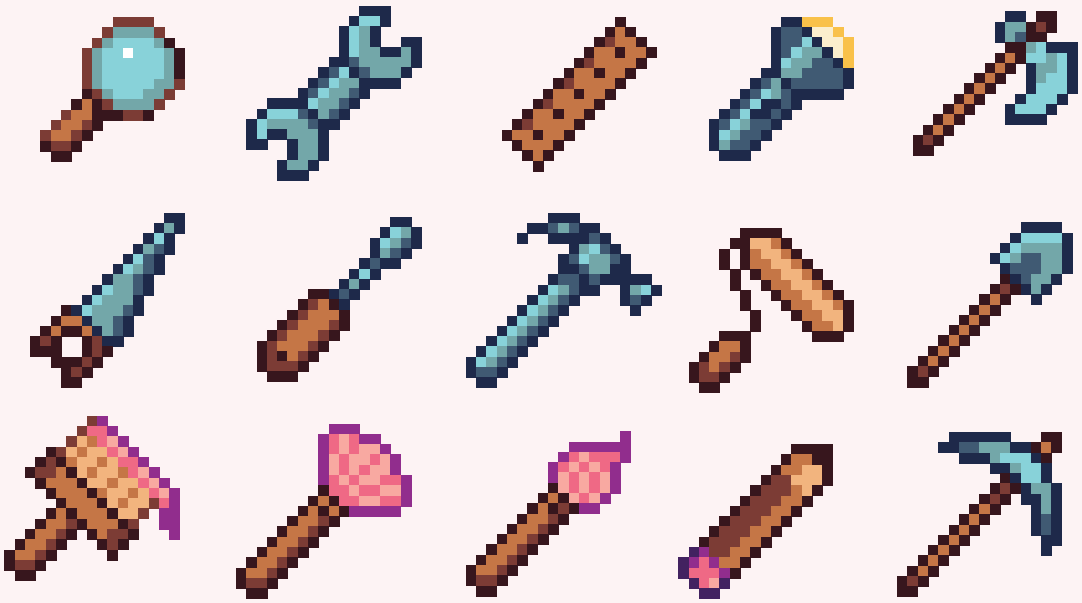


En utökad studie av verktyg som identifierar mjukvarusårbarheter

Verktyg som utvecklats ur forskning publicerad 2014–2024

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

FOI-R--5790--SE

December 2025

FOI

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

En utökad studie av verktyg som identifierar mjukvarusårbarheter

Verktyg som utvecklats ur forskning publicerad 2014–2024

Titel	En utökad studie av verktyg som identifierar mjukvarusårbarheter – Verktyg som utvecklats ur forskning publicerad 2014–2024
Title	An Extended Study on Tools for Identifying Software Vulnerabilities – Tools Developed from Academic Research 2014–2024
Rapportnr/Report no	FOI-R--5790--SE
Månad/Month	December
Utgivningsår/Year	2025
Antal sidor/Pages	78
ISSN	1650-1942
Uppdragsgivare/Client	Försvarsmakten
Forskningsområde	Cyberförsvar och cybersäkerhet
FoT-område	Operationer i cyberdomänen
Projektnr/Project no	E38562
Godkänd av/Approved by	Linda Sjödin
Ansvarig avdelning	Cyberförsvar och ledningsteknik
Exportkontroll	Innehållet är granskat och omfattar ingen information som är underställd exportkontrollagstiftningen.

Bild/Cover Shutterstock

Sammanfattning

Rapporten presenterar en utökad analys av verktyg som identifierar mjukvarusårbarheter och som publicerats vetenskapligt under perioden 2014–2024. Av totalt 273 verktyg beskrivna i litteraturen bedöms 27 vara särskilt intressanta: antingen har de haft en hög teknologisk mognadsgrad redan vid sin ursprungliga publicering, eller så har de fortsatt utvecklas på ett sätt som kan öka deras praktiska användbarhet.

Resultaten visar att de vidareutvecklade verktygen i huvudsak använder fuzzning, dataflödesanalys och symbolisk exekvering, ofta i hybridform. Flera verktyg har hittat nya sårbarheter i produktionsmjukvara, varav många tilldelats CVE-nummer. Studien identifierar även tydliga beroenden inom forskningsområdet, där ett fåtal forskargrupper står bakom en stor del av verktygen. Verktygens målsystem domineras av C/C++ och inbyggda system, men omfattar även Java, Android, Rust och komplexa indataformat.

Nyckelord: mjukvarusårbarheter, cybersäkerhet, fuzzning, dataflödesanalys

Summary

The report presents an extended analysis of tools that identify software vulnerabilities and that have been scientifically published during the period 2014–2024. Of the 273 tools described in the literature, 27 are assessed as particularly relevant: either they exhibited a high level of technological maturity at the time of their original publication, or they have continued to evolve in ways that increase their practical applicability.

The results show that the tools which have undergone further development primarily rely on fuzzing, data-flow analysis, and symbolic execution, often in hybrid form. Several tools have discovered previously unknown vulnerabilities in production software, many of which have been assigned CVE identifiers. The study also highlights clear dependencies within the research field, where a small number of research groups account for a substantial share of the tools. While the dominant target systems are C/C++ and embedded systems, the tools also cover Java, Android, Rust, and systems requiring complex input formats.

Keywords: software vulnerabilites, cybersecurity, fuzzing, data flow analysis

Innehåll

1 Inledning

Den snabba samhälleliga och tekniska utvecklingen har medfört nya sårbarheter och angreppspunkter [1]. Mjukvara är en nyckelkomponent i system som blir allt mer komplexa och sammanvävda. Sårbarheter i mjukvaran kan utnyttjas av en angripare för att få obehörig åtkomst, störa systemets funktion eller stjäla känslig information. Att identifiera och åtgärda sårbarheter är avgörande för att stärka systemets säkerhet.

Genom att studera vetenskapliga publikationer från perioden 2014–2024 sammanställer rapporten kunskap om tekniker och verktyg som identifierar mjukvarusårbarheter. Arbetet tillför Försvarsmakten som kan användas för att testa och utvärdera mjukvara, exempelvis vid ackreditering av system.

Denna studie ingår i ett projekt som syftar till att bygga kunskap om tekniker och verktyg som kan användas för att identifiera mjukvarusårbarheter. För att underlätta användning av projektets samlade resultat inkluderar den här rapporten både verktyg som identifierats i tidigare arbeten och de verktyg som identifierats i det här arbetet.

Sammantaget beskrivs 27 verktyg som identifierats som relevanta utifrån deras tekniska mognad eller utifrån hur aktivt deras community är i verktygets fortsatta utveckling.

1.1 Syfte och mål

Under 2024 publicerades FOI-rapporten *Tekniker och verktyg som identifierar mjukvarusårbarheter: En skanning av forskning publicerad mellan 2014 – 2024* [2]. Rapporten, som framöver benämns som den *tidigare studien*, redogjorde för en genomgång av akademiska publikationer under perioden 2014–2024.

I den tidigare studien identifierades sammanlagt 237 verktyg ur 9 516 publikationer. Av dessa analyserades särskilt 7 verktyg som bedömdes hålla en hög teknologisk mognadsgrad, TRL 8–9 (eng. Technology Readiness Level) [3].

Den här rapporten utökar den tidigare studiens resultat och bidrar till en mer heltäckande beskrivning av forskningsområdet. Rapporten besvarar nedanstående forskningsfråga:

1. *Vilka vetenskapligt publicerade verktyg som identifierar mjukvarusårbarheter är intressanta att utvärdera praktiskt?*

För att besvara frågan genomförs en granskning av de verktyg som i den tidigare studien enbart bedömdes hålla en medelhög mognadsgrad (TRL 5–7). Målet är att identifiera verktyg där det utförts relevant utvecklingsarbete och där verktygens mognadsgrad potentiellt stigit sedan de introducerades genom vetenskaplig publicering.

1.2 Avgränsningar

Arbetet omfattar verktyg beskrivna i akademisk litteratur och som identifierades i den tidigare studien. I dess litteratursökning användes bland annat söktermerna *vuln**, *weakness*, samt *bugs*. Studien inkluderar verktyg som inte bara identifierar mjukvarusårbarheter, utan även identifierar bredare kategorier av buggar. Den här studien ärver även övriga avgränsningar, det vill säga exkludering av verktyg inriktade mot webbsårbarheter liksom verktyg som är tänkta som kodstöd i utvecklingsmiljöer.

2 Metod

I studien av Gustavsson m.fl. (2024) [2] bedömdes den teknologiska mognadsnivån för 237 verktyg utifrån hur de beskrevs i vetenskapliga publikationer. Av dem bedömdes 173 verktyg hålla en medelhög mognadsnivå, vilket definieras som TRL 5–7. Den här studien granskar dessa medelmogna verktyg på nytt, för att identifiera de som har utvecklats sedan ursprunglig publicering. Identifieringen sker genom en process baserad på verktygets tillgänglighet, när uppdateringarna genomförts i tid samt uppdateringarnas relevans.

2.1 Urvalsprocess

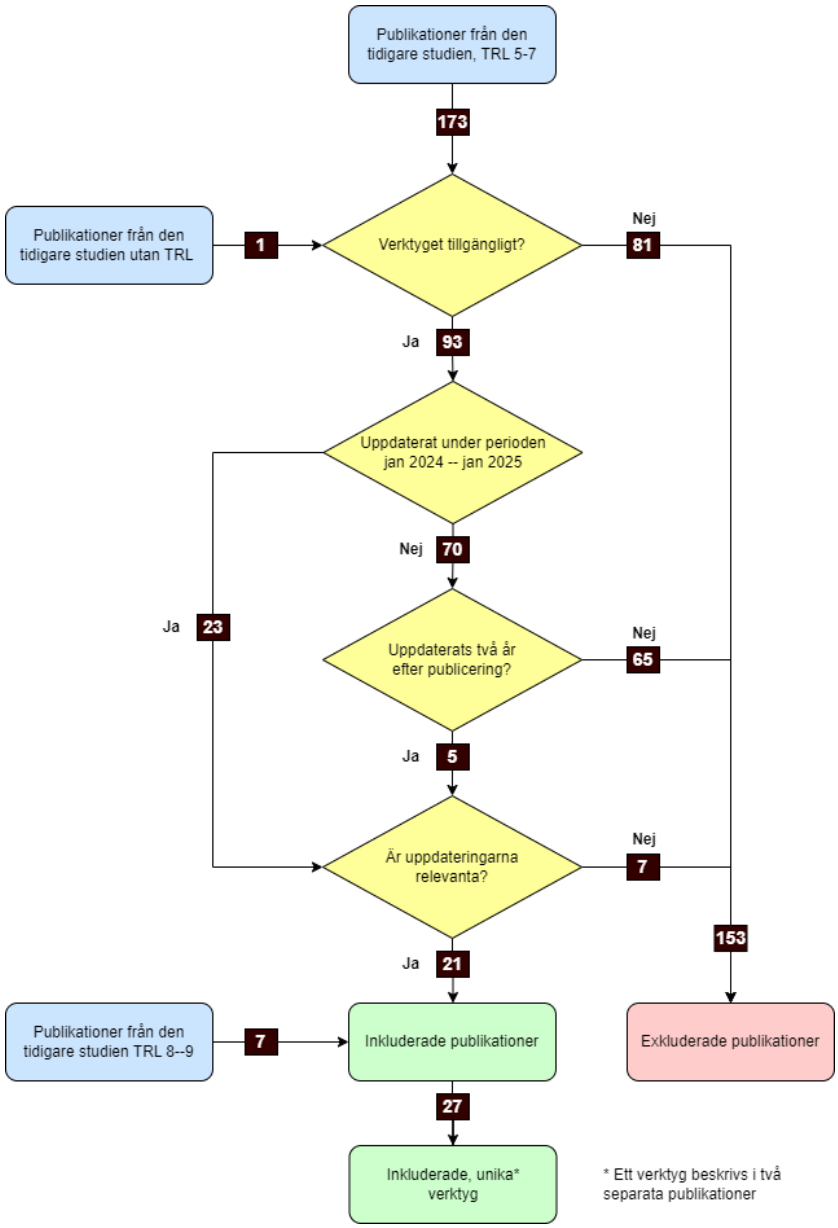
En urvalsprocess genomfördes för att identifiera verktyg där det bedrivits ett relevant utvecklingsarbete. Med relevant utvecklingsarbete avses uppdateringar som potentiellt fått verktyget att stiga i mognadsgrad jämfört med den granskade vetenskapliga publikationen.

Processen är översiktligt beskriven i figur 2.1. Kriterierna för varje steg i processen utvecklas nedan:

Verktyg tillgängligt?
För att kunna undersöka verktygen djupare är det viktigt att de finns publikt tillgängliga för nedladdning på Github.[1]

Uppdaterat under perioden jan 2024–jan 2025?
Kriteriet användes för att fånga upp äldre verktyg som fortfarande utvecklades aktivt samt verktyg vars vetenskapliga publicering skedde nyligen.

[1] Github är en plattform för versionshantering och samarbete där utvecklare kan lagra, dela och arbeta tillsammans på kodprojekt. Mjukvaruprojekt publiceras ofta på Github i syfte att göra det tillgängligt för andra att använda och bygga vidare på.



Figur 2.1. Urvalsprocess för att bestämma vilka publiceringar som ska analyseras vidare.

Uppdaterats två år efter publicering?

Kriteriet användes för att fånga upp äldre verktyg som inte längre uppdateras men som har haft en aktiv utveckling som sträckt sig bortom två år efter ursprunglig publicering.

Är uppdateringarna relevanta?

Kriteriet användes för att exkludera verktyg som uppdaterats men inte på ett relevant sätt sedan publicering. Det finns ett antal aspekter som gjorde att en uppdatering inte sågs som relevant. Exempel på irrelevanta förändringar är uppdateringar i projektets så kallade ”readme”-fil eller att en licens lagts till. Även små uppdateringar av kod som inte påverkar verktygets funktionalitet räknas som irrelevanta.

Av figur 2.1 framgår att ett verktyg, som inte tidigare TRL-skattats, tillförts beslutsprocessen. Den tidigare studien exkluderade verktyget PhASAR [4] då dess publikation brast i information om hur verktyget utvärderats. Vid en ny granskning av tidigare exkluderade publikationer bedöms PhASAR ändå hålla en mycket hög TRL-nivå, trots formella brister. Det motiverar att verktyget inkluderas för den här studien.

Resultatet från varje steg i urvalsprocessen redovisas i figur 2.1. Totalt uppfylls de fastställda urvalskriterierna av 21 publiceringar. Verktyget MOVEC förekommer dock i materialet två gånger, beskrivet i två separata publikationer [5], [6]. Det innebär att 21 publiceringar innebär 20 unika verktyg. Mognadsgraden för dessa verktyg, som sattes i den tidigare studien, presenteras i tabell 2.1.

Tabell 2.1. TRL-fördelning över de verktyg som tog sig igenom till kvalitetsbedömning. Streck indikerar antalet publikationer som inte hade en tilldelad TRL-nivå.

TRL	**–**	**5**	**6**	**7**
Antal publikationer	1	4	6	9

För att underlätta användning av projektets resultat samlar den här rapporten alla verktyg som identifierats i arbetet. Det innebär att de sju verktyg som identifierades i den tidigare studien inkluderas i urvalsprocessen slutsteg.

2.2 Analysarbete

En analys genomfördes av de inkluderade verktygens publikationer i syfte att sammanfatta syfte, använda tekniker och avsedda målsystem. Rapporten använder begreppet målsystem för bland annat olika programspråk, binärer, inbyggda system, komplexa indata. Verktygsanalysen, som genomfördes av en forskare, testar inte verktygen praktiskt, utan begränsar sig till vad som beskrivs i respektive publikation.

Ett arbete genomfördes för att identifiera kopplingar mellan publikationernas författare. Med forskargrupp avses i rapporten en sammanslutning av forskare som samarbetar antingen tillfälligt som författare till enstaka publikationer eller i en permanent universitets- eller forskningsmiljö.

För varje beskrivet verktyg inhämtades också metadata från Github, publicerad i anslutning till respektive verktyg. Den ger läsaren möjlighet att bilda sig en egen uppfattning om mjukvaruprojektets aktivitetsnivå. En sammanställning av inhämtad metadata finns i bilaga A.

3 Resultat

Genom att betrakta verktygen ur både ett vidareutvecklings- och mognadsperspektiv har studien kunnat identifiera ytterligare intressanta verktyg. Dessa nya verktyg bygger på fler tekniker och är inriktade mot fler typer av målsystem.

Kapitlet består av två distinkt olika delar. Den första beskriver identifierade verktyg på en övergripande nivå, samt analyserar verktygens ursprung och beroenden. Den andra delen är tänkt som ett uppslagsverk över identifierade verktyg, med mer detaljerade beskrivningar för den som vill läsa verktygsspecifik information.

För att underlätta användning av resultatet inkluderar kapitlet beskrivningar som fanns med redan i den tidigare studien [2]. De är markerade med en asterisk (*). Beskrivningarna återges i lätt omskriven form för att öka läsbarhet samt som anpassning till den teknikkategorisering som används i rapporten.

3.1 Identifierade verktyg

Genom rapportens urvalsprocess identifierades 27 intressanta verktyg. Verktygen listas i tabell 3.1, ordnade efter målsystem samt huvudsaklig teknikkategori.

Listan över identifierade verktyg domineras av ett begränsat antal målsystem. C/C++ är vanligaste målsystemet för identifierade verktyg, vilket bekräftar tidigare resultat [2]. Gränsdragningen mellan målsystemkategorierna C/C++ och binärer är inte helt entydig. I praktiken analyserar vissa dynamiska verktyg den kompilerade binären, även om de riktar sig mot C-kod. Det finns också verktyg som verkar på mellanliggande representationsnivåer, såsom LLVM IR.[2] I denna studie har uppdelningen därför gjorts mellan verktyg som inte kräver tillgång till källkoden och verktyg som behöver kompilera eller tolka koden för att kunna utföra analysen.

[2] Projektet LLVM har en kollektion av modulära och återanvändbara moduler att konstruera kompilatorer [33]

Tabell 3.1. Identifierade verktyg ordnade efter målsystem samt huvudsaklig teknikkategori. Verktyg markerade med * inkluderas från den tidigare studien [2].

Målsystem	**Teknik**	**Verktyg**
Binärer	Symbolisk exekvering	Angr* [7]
	Fuzzer	VUzzer [8]
C/C++	Dataflödesanalys	PhASAR [4] Goshawk* [9]
	Fuzzning	AFLGo [10] Futag [11] JIGSAW [12]
	Maskininlärning	SySeVR [13]
	Symbolisk exekvering	OSS-Sydr-Fuzz* [14], [15]
	Övriga dynamiska metoder	MOVEC [5], [6] C11Tester [16]
Rust	Dataflödesanalys	SafeDrop [17]
	Övriga statiska metoder	MIRCHECKER [18]
Flera språk	Fuzzning	PolyFuzz* [19]
	Maskininlärning	FUNDED [20]
Android	Dataflödesanalys	AUSERA [21]
Inbyggda system	Dataflödesanalys	SaTC [22]
	Fuzzning	Fuzzware* [23] SFuzz [24] Hoedur [25]
	Symbolisk exekvering	Inception* [26]
Komplex indata	Dataflödesanalys	Crystallizer [27]
	Fuzzning	Zest [28] AFLNET [29] QuickFuzz [30] AFLSmart* [31]
	Övriga dynamiska metoder	Fuzztruction [32]

Inbyggda system är ett annat vanligt målsystem. Samtidigt kan inbyggda system inte sägas vara en sak, utan skiljer sig mycket åt i val av programspråk, hårdvara och gränssnitt. Därmed skiljer sig också testverktygen mycket åt inom kategorin.

Verktyg för komplexa indata är också en stor kategori, men dessa verktyg har en delvis annan karaktär än övriga verktyg. Deras funktion är att generera syntaktiskt och semantiskt korrekt indata som i nästa steg kan användas för att testa ett specifikt målsystem.

Symbolisk exekvering, statisk och dynamisk, samt fuzzning var de mest använda teknikerna hos verktygen som identifierades i den tidigare studien [2]. Bland verktygen som analyserats i denna uppföljande studie dominerar fuzzers, men ofta kombineras också flera tekniker i en hybridapproach. Verktyg sopm använder maskininlärningstekniker förekommer också i den här studien.

De verktyg som uppdaterats under 2025 är Angr, Sydr, SafeDrop, AFLNet, Futag, Fuzzware, SFuzz, Zest och Phasar. Av dessa uppdateras Angr, Sydr, SafeDrop och Phasar vanligtvis flera gånger i månaden medans övriga har ett fåtal uppdateringar under ett år.

Det är värt att notera att urvalsprocessen medfört ett stort bortfall. Det är bara 20 verktyg ur 174 publikationer som uppfyller urvalskriterierna för att vara ett aktivt utvecklingsprojekt. Det största bortfallet var till följd av att verktyg inte längre är publikt tillgängliga. Av de som är tillgängliga så är det bara en liten andel som vidareutvecklats efter sin ursprungliga publicering. Knappt 12 % av de verktyg som bedömts ha en medelhög mognadsnivå (TRL 5–7) har vidareutvecklats på något relevant sätt efter den initiala publiceringen.

3.2 Verktygens ursprung och beroenden

Analysen visade att ett fåtal forskargrupper låg bakom en stor del av de identifierade verktygen, vilket framgår av tabell 3.2. Sex forskargrupper dominerade studien och stod tillsammans för 12 av de 27 verktygen. Två av dessa grupper hade dessutom utvecklat ytterligare tre verktyg – Driller, Karonte och Soot – som inte fångades upp i litteratursökningen, men som ofta användes som komponenter i de verktyg som presenterades i rapporten.

Tabell 3.2. Identifierade författargrupper ansvariga för mer än ett verktyg som identifierats i projektet.

Forskargrupp	Verktyg	Ursprungsland
1	Angr Driller Karonte	USA
2	SFuzz SaTC	Kina Kina
3	Fuzzware [3] Hoedur Fuzztruction	Tyskland/USA Tyskland Tyskland
4	PhASAR Soot	Tyskland
5	AFLGo AFLNET AFLSmart	Singapore Singapore/Australien Singapore/Australien
6[4]	Sydr Futag	Ryssland Ryssland

Forskargrupper som står bakom enbart ett av studiens identifierade verktyg har sitt ursprung vid institutioner främst i USA och Kina, men även Australien, Frankrike, Nederländerna, Argentina, Schweiz och Hongkong. Flera av dessa verktyg har utvecklats genom internationella samarbeten, till exempel mellan Kina och Singapore eller USA och Australien.

En annan observation är de många beroenden som finns inom forskningsområdet. Nya verktyg tenderar att bygga vidare på äldre verktyg eller använder äldre verktyg som delsystem. I tabell 3.3 listas identifierade beroenden. Ett vanligt beroende är verktyg som kommer ur LLVM-projektet, som LLVM IR, KLEE, libFuzzer och Clang. Det finns också många beroenden mellan fuzzerverktyg, där AFL och AFL++ används i fem andra fuzzerverktyg.

[3] Delvis gemensamma författare med Angr.

[4] Inget överlapp bland forskare mellan artiklarna men det finns överlapp i andra artiklar rörande till exempel Sydr. Verktygen har även samma Github skapare.

[5] Det finns fler verktyg som använder LLVM IR, men då oftast för kompilering och inte analys.

Tabell 3.3. Verktygens identifierade beroenden av andra verktyg och stödjande tekniker. Under kolumnen ”använt system” finns bara verktyg som är relevanta för verktygens sårbarhetsanalys och som används av mer än ett verktyg eller skapats av en författargrupp från tabell 3.2.

Använt system	Verktyg
AFL	Fuzzware, AFLGo, AFLSmart, AFLNet
AFL++	Fuzzware, PolyFuzz
Angr	SFuzz
LibFuzzer	Hoedur, Futag
Driller	SFuzz
Ghidra	SFuzz, SaTC
Joern	FUNDED, SySeVR
KLEE	Angr, Inception
Karonte	SaTC
LLVM IR	PhASAR, AFLGo, Futag, JIGSAW, Inception, Fuzztruction, Polyfuzz [5]
QuickCheck	QuickFuzz, Zest
SaTC	SFuzz
Soot	FUNDED, Crystallizer, AUSERA, Polyfuzz

3.3 Verktygsbeskrivningar

Bland de sammanlagt 27 verktygen som identifierats i projektet använder sig tolv huvudsakligen av fuzzning, sex av dataflödesanalys, tre av symbolisk exekvering, tre av övriga dynamiska metoder, två av maskininlärning och en av övriga statiska metoder. Flera verktyg kompilerar koden till LLVM intermediärrepresentation (IR) som ett steg i sin analys. En modul är LLVM:s intermediärrepresentation, till vilken kod skriven i andra programspråk kan omtolkas.

Avsnittet fungerar som ett uppslagsverk över identifierade verktyg, med en detaljerad beskrivning för den som vill läsa mer. De har ordnats utifrån hur projektet bedömer deras huvudsakliga teknikkategori. Varje teknikkategori inleds med en allmän bakgrund till tekniken och viktiga begrepp. För respektive verktyg beskrivs också metadata gällande Github-projektet.[6]

[6] Data inhämtad från Github 2025-09-22.

Två begrepp är centrala vid katalogisering och klassificering av mjukvarusårbarheter:

- Common Weakness Enumeration (CWE) är en katalog med sårbarhetstyper.[7] Sårbarhetstyperna tilldelas ett CWE-nummer och är beskrivna i katalogen på en övergripande och konceptuell nivå.
- Common Vulnerabilities and Exposures (CVE) är en katalog med inrapporterade, konkreta, sårbarheter.[8] När en sårbarhet är bekräftad, av en partnerorganisation till databasen, tilldelas den ett CVE-nummer och publiceras i katalogen. I många fall publiceras sårbarheten med en översiktlig beskrivning. [9]

[7] https://cwe.mitre.org
[8] https://cve.org
[9] https://www.cve.org/About/Process#CVERecordLifecycle

3.3.1 Dataflödesanalys

Dataflödesanalys (eng. data flow analysis) undersöker hur data propagerar genom en mjukvara, med målet att ta reda på hur variabler och mjukvarans tillstånd påverkas. Dataflödesanalys genomförs vanligtvis genom att först bygga upp en kontrollflödesgraf för att få information om vilka vägar som finns i systemet och därefter analysera hur värden sprids mellan olika variabler och uttryck. Genom dataflödesanalys kan olika typer av fel upptäckas, till exempel om en variabel används innan den tilldelats ett värde [34]. Dataflödesanalys kan vara både statisk och dynamisk.

Dataflödesanalys av potentiellt osäkra data (eng. taint analysis) är en specialisering av dataflödesanalys som enbart följer data från källor som betraktas som osäkra, till exempel data från ett användargränssnitt. Denna analys kan vara såväl statisk som dynamisk [35]. Statisk dataflödesanalys av potentiellt osäkra data sker genom spårning i en dataflödesgraf. I sin dynamiska form spåras istället indatas väg genom mjukvaran vid körning. Dynamisk analys använder information som samlas in under körning (exempelvis genom instrumentering som beskrivs i avsnitt 3.3.5) för att få mer tillförlitliga resultat. Dynamisk flödesanalys kan vara mer effektiv än statisk flödesanalys [36].

AUSERA

Nyckelord: Android, informationsläckage

Commits	21	**Första commit**	2022-04-18	**Uppdaterat senast (år)**	2024
Contributors	2	**Forks**	3	**Stars**	32
Releases	0	**Branches**	1		
Listade CWE/CVE: Ej listade. Författarna menar att AUSERA kan hitta sårbarheter från en egenutvecklad taxonomi, innehållandes kategorier som datalagring, datakryptering, säkra konfigurationer, behörighetskontroller och om datan kan kommas åt genom osäkra funktioner.					
Projekt tillgängligt: https://github.com/tjusenchen/AUSERA **Verktygsdemonstration:** https://www.youtube.com/watch?v=UCiGwVaFPpY **DOI:** https://doi.org/10.1145/3551349.3559524, https://doi.org/10.1145/3377811.3380417					

AUtomated SEcurity Risk Assessment system (AUSERA) identifierar sårbarheter i androidapplikationer [21]. Verktyget letar specifikt efter sårbarheter relaterade till dataläckage.

AUSERA använder statisk analys och nyckelordsidentifiering för att identifiera sårbarheter [37]. För att använda verktyget behöver användaren viss domänkunskap för att skapa en lista med känsliga nyckelord, exempelvis personnummer eller kontonummer. Listan används sedan för att hitta känsliga variabler och delar av koden genom statisk analys. Koden analyseras för att hitta de funktioner som leder till den data som bedöms som känslig. Dataflödesanalys används därefter för att spåra vilka känsliga delar som går att nå. Dataflödesanalysen är baserad på verktyget Soot.[10]

AUSERA:s författare utgår från en egenutvecklad taxonomi med sårbarheter relaterade till dataläckage. De menar att AUSERA kan hitta alla sårbarhetstyper som finns i taxonomin. Taxonomin är delvis baserad på en studie gjord av AUSERA:s författare där över 2 000 svagheter samlades in från nästan 700 bankapplikationer. AUSERA har hittat flertalet sårbarheter, varav 52 sårbarheter i 21 mjukvaror har bekräftats och åtgärdats.

[10] https://soot-oss.github.io/soot/

Crystallizer

Nyckelord: Java, serialiseringssårbarheter

Commits	22	**Första commit**	2023-11-07	**Uppdaterat senast (år)**	2024
Contributors	0	**Forks**	1	**Stars**	14
Releases	0	**Branches**	1		
Listade CWE/CVE: Inga nya nolldagarssårbarheter är listade i publikationen. Verktyget letar sårbarheter kopplade till serialisering och deserialisering.					
Projekt tillgängligt: https://github.com/HexHive/Crystallizer **DOI:** https://doi.org/10.1145/3611643.3616313					

Crystallizer söker efter sårbarheter som uppstår i samband med serialisering och deserialisering av data [27]. Serialisering används för att omvandla data till ett format som kan överföras mellan system, medan deserialisering återskapar den ursprungliga datastrukturen. Sårbarheter kan uppstå när denna process hanterar skadlig eller oväntad indata, till exempel om objekt återskapas på ett sätt som möjliggör exekvering av oönskad kod. Bristande validering av indata är en vanlig orsak till att sådana sårbarheter kan utnyttjas. Crystallizer kombinerar statisk och dynamisk analys för att identifiera dessa problem.

Statisk analys används för att identifiera funktioner och koddelar som tillsammans kan skapa sårbarheter. Dessa kopplas samman i en övergripande funktionsgraf som beskriver möjliga exekveringsvägar. Den dynamiska analysen testar sedan vilka av dessa vägar som faktiskt kan utnyttjas under körning. Crystallizer analyserar Java-kod och använder Soot både för den statiska analysen och för att instrumentera övervakningskod i den dynamiska fasen.

Verktyget har utvärderats på produktionsmjukvaror och har hittat 41 nya sårbarheter. Författarna menar att de meddelat systemägarna om sårbarheterna men inga nya CVE:er listas i publikationen.

Goshawk*

Nyckelord: C/C++, minneskorruptionsbuggar

Commits	59	**Första commit**	2021-09-27	**Uppdaterat senast (år)**	2023
Contributors	4	**Forks**	15	**Stars**	99
Releases	0	**Branches**	4		
Listade CWE/CVE: Ej listade. Verktyget letar minnessårbarheter, exempelvis use-after-free och double-free sårbarheter.					
Projekt tillgängligt: https://github.com/Yunlongs/Goshawk **DOI:** https://doi.org/10.1109/SP46214.2022.9833613					

Goshawk inriktar sig på att hitta minneskorruptionsbuggar [9]. Författarna menar att andra verktyg inte kan hitta buggar som existerar i specialskrivna minneshanteringsfunktioner, utan att de kan endast hitta buggar där koden använder standardfunktioner såsom malloc och free.

Verktyget använder sig av språkteknik[11] (eng. natural language processing) och dataflödesanalys för att identifiera specialskrivna minneshanteringsfunktioner. När funktionerna har identifierats kan de analyseras genom att leta efter minneskorruptionsbuggar med en anpassad version av analysverktyget Clang Static Analyzer (CSA).[12] Goshawk använder verktyget CodeChecker vilket i sin tur använder LLVM/CSA.[13]

I publikationen anges att Goshawk hittat 92 nya buggar.

[11] Bygger på samma neurala nätverksarkitekturer som stora språkmodeller
[12] https://clang-analyzer.llvm.org/
[13] https://codechecker.readthedocs.io/en/latest/

PhASAR

Nyckelord: C/C++, större ramverk

Commits	3,226	**Första commit**	2017-02-09	**Uppdaterat senast (år)**	2025
Contributors	48	**Forks**	149	**Stars**	1007
Releases	4	**Branches**	52		
Listade CWE/CVE: Ej listade. Phasar är ett brett ramverk och kan enligt författarna lösa arbiträra dataflödesproblem.					
Projekt tillgängligt: https://github.com/secure-software-engineering/phasar **Projekthemsida:** https://phasar.org/ **DOI:** https://doi.org/10.1007/978-3-030-17465-1_22					

PhASAR är ett LLVM-baserat, statiskt, analysverktyg som är skapat för att analysera C/C++–kod. Verktyget beskrivs även som användbart för andra programspråk ifall koden kompileras till LLVM IR [4]. Verktyget använder sig av dataflödesanalys för att hitta sårbarheter. Dataflödesanalys kan utföras med flera olika metoder och PhASAR är modulärt uppbyggt för att en användare ska kunna välja lämplig metod för sitt problem. Användare kan även skriva sina egna metoder för att utföra ytterligare analyser. PhASAR kan även användas som en modul i andra verktyg.

Genom att låta användarna specificera problemet, samt vilken metod som ska användas, menar författarna att PhASAR kan användas för att lösa godtyckligt dataflödesproblem. Verktyget använder sig av IR-kod istället för källkoden eftersom den generellt har en enklare struktur och medför en enklare analys [4]. Verktyget är skapat för att analysera programkod i C och C++, men eftersom många programspråk går att kompilera till LLVM IR kan det i praktiken ha ett större användningsområde.

PhASAR:s författare har tidigare utvecklat liknande verktyg för Java. Ett av verktygen heter Soot och används av flera av de andra verktygen som omnämns i den här studien, exempelvis AUSERA och Crystallizer.

SafeDrop

Nyckelord: Rust, ramverk, minnessårbarheter

Commits	780	**Första commit**	2022-11-05	**Uppdaterat senast (år)**	2025
Contributors	12	**Forks**	27	**Stars**	109
Releases	1	**Branches**	2		

Listade CWE/CVE: SafeDrop har hittat nya sårbarheter som fått CVE:nummer inom kategorierna use-after-free, double-free, hängande pekare och ogiltig minnesåtkomst. De funna sårbarheterna har tilldelats följande CVE-nummer: 2019-16140, 2018-20997, 2019-16880, 2020-35891, 2018-20991, 2019-15551, 2018-20996, 2019-16144 och 2020-25573.

Projekt tillgängligt: https://github.com/VaynNecol/SafeDrop (gammal)
https://github.com/Artisan-Lab/RAPx (nuvarande)
Projekthemsida: https://artisan-lab.github.io/RAPx/
DOI: https://doi.org/10.1145/3542948

SafeDrop identifierar minnessårbarheter relaterade till felaktig avallokering i programspråket Rust. Sårbarheterna identifieras genom statisk dataflödesanalys [17]. SafeDrop integreras i Rust-kompilatorn och analyserar koden under kompilering. Analysen sker iterativt på Rust MIR[14] (eng. mid-level intermediate representation) där det vid varje släppt referens (eng. drop statement) utförs en analys för att avgöra om åtgärden var säker att genomföra.

Verktygets publikation visar att SafeDrop kan hitta sårbarheter i Rustkod. SafeDrop utvecklas numera inom projektet RAPx.[15]

[14] https://rustc-dev-guide.rust-lang.org/mir/index.html
[15] https://github.com/artisan-lab/rapx

SaTC

Nyckelord: IoT, webbgränssnitt, nyckelordsidentifiering

Commits	96	**Första commit**	2020-12-06	**Uppdaterat senast (år)**	2024
Contributors	3	**Forks**	59	**Stars**	317
Releases	1	**Branches**	3		
Listade CWE/CVE: Hittat 36 nya sårbarheter inom bland annat kategorierna kommandoinjektion, buffertöverskridning och felaktig behörighetskontroll. Vissa av dessa har fått CVE-nummer medan andra har rapporterats till exmepelvis Kinas nationella sårbarhetsdatabas (CNVD) istället. En lista med CVE-nummer finns på verktygets Github.					
Projekt tillgängligt: https://github.com/NSSL-SJTU/SaTC **DOI:** https://doi.org/10.1109/TDSC.2023.3307430					

Shared-keyword aware Taint Checking (SaTC) använder statisk dataflödesanalys av osäker data (eng. taint analysis) för att hitta sårbarheter i IoT-enheter [22]. Verktygets författare menar att många osäkra IoT-enheter, till exempel trådlösa routers och webbkameror, har ett webbgränssnitt och att detta ofta används för att attackera enheterna. SaTC använder dataflödesanalys för att följa indata från gränssnittet genom målsystemet för att hitta sårbarheter.

Verktyget utgår från att implementationen av webbgränssnittet använder liknande nyckelord i såväl användarnära komponenter (eng. frontend) som i servernära komponenter (eng. backend), detta medför att data kan följas med statisk analys mellan de två delarna utan att manuellt behöva para ihop dem. Under verktygets författares utvärdering av verktyget hittades 36 ej tidigare rapporterade sårbarheter i 37 routrar och 2 webbkameror.

SaTC bygger vidare på verktygen Ghidra[16] och Karonte [38]. Författarna till SaTC står också bakom SFuzz som också beskrivs i rapporten.

[16] https://github.com/NationalSecurityAgency/ghidra

3.3.2 Fuzzers

Fuzzning är en dynamisk teknik som skapar stora mängder indata som i sin tur matas till en mjukvara för att trigga buggar och sårbarheter. Indatan som fuzzers skapar kan vara oväntade eller ogiltiga värden. Syftet med att använda sådana värden är att framkalla en krasch eller ett oönskat beteende. Det finns många olika typer av fuzzers där metoden som används för att generera indata varierar [39].

Två vanliga metoder för att skapa indata är genereringsbaserade och mutationsbaserade metoder [39]. Genereringsbaserade fuzzers skapar indata utifrån en given specifikation, medan mutationsbaserade fuzzers utgår från ett befintligt frö som ändras för att producera ny indata.

Fuzzers kan använda information från fuzzningen för att beräkna hur stor del av koden som testats. Informationen kan användas för att ta smartare beslut i genereringen av indata och kallas täckningsstyrd fuzzning. Både genereringsbaserade och muteringsbaserade fuzzers kan vara täckningsstyrda.

AFLGo

Nyckelord: C/C++, målsökning

Commits	345	**Första commit**	2014-12-02	**Uppdaterat senast (år)**	2023
Contributors	0	**Forks**	140	**Stars**	536
Releases	1	**Branches**	2		
Listade CWE/CVE: Verktyget kan hitta sårbarheter av bland annat följande typer: bufferöverskridning, devide-by-zero, invalid write, write access violation och stack corruption.					
Projekt tillgängligt: https://github.com/aflgo/aflgo **DOI:** https://doi.org/10.1145/3133956.3134020					

AFLGo är en greyboxfuzzer som bygger vidare på AFL [10].[17] Verktyget beskrivs som dirigerat, menat att det går att styra fuzzern mot utpekade delar av målsystemet. Verktyget analyserar mjukvaran under kompilering istället för under körning. Avsikten är att göra AFLGo mer tidseffektiv än fuzzers baserade på symbolisk exekvering, där beräkningarna sker under körning [10].

Mjukvaran analyseras i en LLVM IR av källkoden, omtolkad i ett anropsdiagram. Det används för att ta reda på kortaste vägen från en startpunkt till en annan, given punkt i mjukvaran. AFLGo genererar därefter indata som styr fuzzern till de intressanta delarna.

I publikationen visas att AFLGo upptäckt 17 nolldagarssårbarheter i välanvända mjukvaror [10]. Författarna till AFLGo jämför också verktyget med KATCH [41].[18] Resultaten visar att AFLGo är snabbare och hittar fler sårbarheter. Samtidigt noteras att kombinationen av de båda verktygen identifierar 42 procent fler buggar än när de används var för sig. Verktygen upptäcker delvis olika sårbarheter. Slutsatsen är att det kan vara rekommenderat att använda dem tillsammans.

AFLGo har samma författare som verktygen AFLNet och AFLSmart.

[17] Greybox innebär det finns viss kännedom om ett målsystems struktur vid testning vilket ger testaren större möjlighet att till exempel skräddarsy testfall [40].

[18] KATCH är baserat på plattformen KLEE — ett av de mest använda verktygen för symbolisk exekvering av C-kod [10]. KATCH utvecklades av samma utvecklare som KLEE. https://klee-se.org/

AFLNET

Nyckelord: Server, komplex indata, nätverksprotokoll

Commits	99	**Första commit**	2019-07-25	**Uppdaterat senast (år)**	2025
Contributors	38	**Forks**	203	**Stars**	956
Releases	0	**Branches**	1		
Listade CWE/CVE: Två nolldagarssårbarheter med tilldelade CVE-nummer: 2019-7314 och 2019-15232.					
Projekt tillgängligt: https://github.com/aflnet/aflnet **DOI:** http://doi.org/10.1109/ICST46399.2020.00062					

AFLNET används för att fuzztesta servrar genom att skicka skicka förfrågningar till servern via nätverk [29]. Verktygets indata utgörs av en pcap-fil med inspelade meddelanden som skickas mellan servern och en riktig klient. AFLNET tar sedan över rollen som klient men modifierar meddelandena för att testa serverns funktion.

Under analyasen bygger verktyget upp en modell i form av en tillståndsmaskin som löpande utökas för att reflektera serverns tillstånd och därigenom nå djupare delar av koden. I publikationen testas AFLNET med File Transfer Protocol (FTP) och Real Time Streaming Protocol (RTSP). Testerna visar att AFLNET kan hitta nolldagarssårbarheter. Författarna planerar även att i framtiden utföra tester med andra välanvända protokoll.

AFLSmart*

Nyckelord: Komplex indata

Commits	257	**Första commit**	2014-12-02	**Uppdaterat senast (år)**	2022
Contributors	10	**Forks**	91	**Stars**	517
Releases	0	**Branches**	2		
Listade CWE/CVE: I publikationen och på Github finns flera listade CVE:er. AFLSmart kan hitta sårbarheter i bland annat följande kategorier: dereferering av null-pekare, bufferöverläsning av heapen, bufferöverskrivning av heapen och assertion failure.					
Projekt tillgängligt: https://github.com/aflsmart/aflsmart **DOI:** http://doi.org/10.1109/TSE.2019.2941681					

AFLSmart är en täckningsstyrd greyboxfuzzer som används för att fuzztesta mjukvara som kräver komplex indata [31]. Verktyget bygger vidare på både AFL och Peach.[19]

AFLSmart använder vad författarna beskriver som smart greyboxfuzzning. Metoden bygger på att konstruera bättre indata till fuzzern, vilket ska resultera i att fuzzern tar sig djupare in i målsystemet. Peach används för att skapa semantiskt korrekt indata. Filformaten för indata behöver definieras manuellt innan AFLSmart kan användas.

Författarna presenterar resultat där AFLSmart har hittat 42 ej tidigare rapporterade sårbarheter där 22 av dem tilldelats CVE-nummer.

[19] https://peachtech.gitlab.io/peach-fuzzer-community/

Futag

Nyckelord: C/C++, mjukvarubibliotek

Commits	190	**Första commit**	2022-01-27	**Uppdaterat senast (år)**	2025
Contributors	4	**Forks**	11	**Stars**	52
Releases	17	**Branches**	2		
Listade CWE/CVE: Inga sårbarheter listas i publikationen. Däremot nämns det att verktyget bland annat kan hitta buffertöverskridningar och minnesläckor.					
Projekt tillgängligt: https://github.com/ispras/Futag **DOI:** http://doi.org/10.1109/IVMEM53963.2021.00021					

Futag automatiserar den process som krävs för att skapa fuzzningswrappers. Dessa wrappers är små mjukvaror som underlättar för fuzzers att nå specifika delar av ett mjukvarubibliotek eller ett API. Med hjälp av statisk analys bygger verktyget upp en representation av bibliotek som testas. Representationen används sedan för att generera små mjukvaror som kallar på funktioner i biblioteket. En valfri fuzzer kan sedan appliceras på dessa små mjukvaror för att identifiera potentiella sårbarheter. För analysen används LLVM IR.

Bakom Futag står samma författare som för Sydr, vilket är ett verktyg som också presenteras i rapporten.

Publikationen visar att verktyget kan identifiera sårbarheter i C och C++-kod.

Fuzzware*

Nyckelord: Inbyggda system

Commits	77	**Första commit**	2022-01-20	**Uppdaterat senast (år)**	2025
Contributors	11	**Forks**	62	**Stars**	350
Releases	2	**Branches**	2		
Listade CWE/CVE: 12 nolldagarssårbarheter hittades och är listade i publikationen. Tre av CVE-nummren är följande: 2020-12141, 2021-3321, 2021-3330					
Projekt tillgängligt: https://github.com/fuzzware-fuzzer/fuzzware					

Fuzzware testar firmware i inbyggda system [23], så kallad inbyggd mjukvara. Verktyget fokuserar på att skapa relevanta indatavärden baserat på vilka hårdvarufunktioner som används av den inbyggda mjukvaran.

Verktyget testar mjukvaran i en förenklad, emulerad hårdvarumiljö. För det används en ISA-emulator (Instruction Set Architecture Emulator).

Fuzzware bygger vidare på både AFL och AFL++.[20] Det har utvecklats av samma forskargrupp som står bakom verktygen Houdur och Fuzztruction, i samarbete med forskarna bakom Angr.

Publikationen visar att verktyget hittat 15 nya sårbarheter varav 12 har tilldelats CVE-nummer.

[20] https://github.com/AFLplusplus/AFLplusplus

Hoedur

Nyckelord: Inbyggda system

Commits	12	**Första commit**	2023-06-21	**Uppdaterat senast (år)**	2024
Contributors	2	**Forks**	10	**Stars**	66
Releases	1	**Branches**	1		
Listade CWE/CVE: Hoedur har hittat 22 nya nolldagarsårbarheter. Dessa är listade i publikationen. Många av CVE:erna är uppkomna på grund av olika minnesproblem.					
Projekt tillgängligt: https://github.com/fuzzware-fuzzer/hoedur					

Hoedur är en fuzzer som testar inbyggda system och inbyggd mjukvara [25]. Verktyget har samma författare som Fuzzware. I den här publikationen menar de dock att det kan vara svårt att emulera inbyggda system, då de ofta har en huvudsaklig loop som körs, kombinerat med avbrottsrutiner som aktiveras regelbundet.

En emulator läser ofta en input-fil som stegvis sätter register och annan hårdvaruinput. Vid fuzzing kan mutationer i filen leda till orimliga registervärden och därmed falska fel. Hoedur skapar istället en separat inputström för varje hårdvaruåtkomst, så att varje ström kan muteras oberoende av de andra. På så vis analyserar Hoedur hur inbyggd mjukvara reagerar på olika hårdvarutillstånd.

Hoedur är baserad på fuzzern libFuzzer[21] men lånar också viss funktionalitet från AFL och AFL++. ISA-emulatorn som används består av QEMU[22] med mindre modifikationer.

I redovisade tester upptäckte Hoedur 23 ej tidigare rapporterade sårbarheter i produktionsmjukvaror.

[21] https://llvm.org/docs/LibFuzzer.html
[22] https://www.qemu.org/

JIGSAW

Nyckelord: C/C++, prestanda

Commits	27	**Första commit**	2021-09-10	**Uppdaterat senast (år)**	2024
Contributors	2	**Forks**	8	**Stars**	69
Releases	0	**Branches**	1		
Listade CWE/CVE: Publikationens fokus är att effektivisera fuzzningen snarare än att hitta sårbarheter.					
Projekt tillgängligt: https://github.com/R-Fuzz/jigsaw **DOI:** https://doi.org/10.1109/SP46214.2022.9833796					

JIGSAW (Just-in-Time Gradient descent Search for AnsWers) är en fuzzer som fokuserar på effektivitet [12]. Med det avses hur effektivt verktyget hittar sårbarheter samt hur många operationer per tidsenhet som verktyget kan genomföra. JIGSAW använder LLVMs verktyg för JIT-kompilering (Just In Time-kompilering) och att parallellisera testningen över flera processorkärnor.

Verktyget bygger ett abstrakt syntaxträd (eng. abstract syntax tree, AST) där varje nod innehåller en matematisk funktion som beskriver vilka indata som leder till den kodgrenen. Vid fuzzning testas först detta träd —- snabbare än att köra hela mjukvaran —- så att irrelevanta testfall sorteras bort och mer lovande indata prioriteras. JIGSAW-fuzzing bygger på fuzzern Angora.

Författarna utvärderar verktyget på flera produktionsmjukvaror samt Google Fuzzbench. Google FuzzBench är en öppen plattform för att automatiskt utvärdera och jämföra fuzzningsteknikers prestanda på ett standardiserat och reproducerbart sätt.[23] Författarnas utvärdering fokuserar dock inte på att hitta nya sårbarheter, utan utan fokuserar främst på metodens skalbarhet.

[23] https://google.github.io/fuzzbench/

Polyfuzz*

Nyckelord: Flerspråkiga kodbaser

Commits	773	**Första commit**	2021-11-05	**Uppdaterat senast (år)**	2022
Contributors	3	**Forks**	3	**Stars**	27
Releases	6	**Branches**	1		
Listade CWE/CVE: Polyfuzz har hittat fem nya nolldagarssårbarheter. Dessa och andra hittade sårbarheter är listade på verktygets Github. Polyfuzz har bland annat hittat följande CVEer: 2022-34070, 2022-34072, 2022-34073, 2022-34074 och 2022-34075.					
Projekt tillgängligt: https://github.com/awen-li/Polyfuzz					

Verktyget Polyfuzz är utvecklat för att hantera mjukvaror som bygger på kodbaser i flera programspråk. Författarna menar att majoriteten av fuzzerverktyg endast fokuserar på ett programspråk vilket gör dem mindre effektiva när det kommer till mjukvaror med blandade kodbaser [19].

Polyfuzz är en greybox-fuzzer som börjar med att förbereda målsystemet för fuzzning och genererar relevanta indatavariationer. Verktyget är utvecklat och utvärderat för C, Java och Python, men författarna menar att det kan utökas till fler programspråk. För att hantera skilda språkliga semantiker använder Polyfuzz en specialbyggd intermediärrepresentation: LLVM- och Soot-baserade representationer från respektive språk konverteras till Polyfuzz egen intermediärrepresentation [19]. Verktyget är baserat på AFL++.

I publikationen visas att Polyfuzz hittat 14 nya sårbarheter som fått totalt fem CVE-nummer tilldelade [19].

QuickFuzz

Nyckelord: komplex indata, blackbox

Commits	570	**Första commit**	2015-06-27	**Uppdaterat senast (år)**	2019
Contributors	16	**Forks**	45	**Stars**	198
Releases	2	**Branches**	4		
Listade CWE/CVE: Quickfuzz har hittat flera nya CVE:er vilka är listade i verktygets publikation.					
Projekt tillgängligt: https://github.com/CIFASIS/QuickFuzz **DOI:** https://doi.org/10.1145/2976002.2976017					

QuickFuzz används för att testa målsystem som tar emot komplex indata, såsom bilder, ljud, videos och PDF-filer [30]. QuickFuzz modifierar indatafilerna för att testa målsystemet med korrupt data. Författarna menar att många liknande verktyg kräver manuellt skapande av filformatspecifikationer vilket QuickFuzz undviker genom att använda bibliotek som skapar och muterar filer.

QuickFuzz är en blackboxfuzzer[24] och kan därför användas för att testa mjukvaror skrivna i alla programspråk. Quickfuzz använder tredjepartsbibliotek från plattformen Hackage[25] för att kunna hantera olika filformat. Tredjepartsverktyget QuickCheck generar slumpvisa variationer av indatafiler utifrån det valda filfomatet.[26] QuickFuzz är i sig inte en fuzzer, utan används i kombination med mutationsdrivna fuzzers av samma forskargrupp, ex. Zzuf[27], Radamsa [28], Honggfuzz[29]

Publikationen visar att QuickFuzz hittat 14 nolldagarssårbarheter i välanvända mjukvaror, exempelvis Firefox.

[24] En blackboxfuzzer har ingen kännedom om målsystemet som testas [39].
[25] https://hackage.haskell.org/package/QuickCheck
[26] QuickCheck är ett verktyg för automatiska tester av mjukvaror skrivna i språket Haskell.
[27] https://github.com/samhocevar/zzuf
[28] https://gitlab.com/akihe/radamsa
[29] https://github.com/thebabush/honggfuzz-qemu

SFuzz

Nyckelord: inbyggda system, realtidssystem

Commits	9	**Första commit**	2022-09-22	**Uppdaterat senast (år)**	2025
Contributors	4	**Forks**	11	**Stars**	90
Releases	0	**Branches**	1		
Listade CWE/CVE: Verktyget har hittat 77 nya sårbarheter och många av dessa är listade i verktygets publikation. 67 av dessa har tilldelats CVE– eller CNVD–nummer.					
Projekt tillgängligt: https://github.com/NSSL-SJTU/SFuzz **DOI:** https://doi.org/10.1145/3548606.3559367					

SFuzz används för att hitta sårbarheter i realtidssystem (eng. Real-Time Operating System, RTOS), vilket är en vanlig typ av inbyggt system [24]. SFuzz använder sig av en metod som verktygets författare benämner som delbaserad fuzzing (eng. slice-based fuzzing). Metoden går ut på att hitta, emulera och fuzztesta de delar av systemet som används vid ett anrop istället för att emulera och testa ett helt inbyggt system.

SFuzz använder statisk analys för att hitta delar av systemet som är intressanta för fuzzning. Symbolisk exekvering används också för att få en mer robust analys i de fall som fuzzern fastnar i sin analys. Bakom SFuzz står samma grupp av författare som utvecklat verktyget SaTC som även det beskrivs i den här rapporten.

SFuzz är baserad på Ghidra och Angr för analys samt Driller och UnicornAFL[30] för fuzzning [42].

Publikationen visar att SFuzz hittat 77 nolldagarssårbarheter i 35 realtidssystem, varav 67 av dessa blivit tilldelade CVE- eller CNVD-nummer.

[30] https://github.com/Battelle/afl-unicorn

VUzzer

Nyckelord: Binärer, applikationsmedveten

Commits	9	**Första commit**	2017-02-28	**Uppdaterat senast (år)**	2019
Contributors	3	**Forks**	108	**Stars**	384
Releases	0	**Branches**	1		
Listade CWE/CVE: Inte beskrivet i publikationen.					
Projekt tillgängligt: https://github.com/vusec/vuzzer **DOI:** http://dx.doi.org/10.14722/ndss.2017.23404					

VUzzer benämns som en applikationsmedveten fuzzer [8]. Författarna menar att många fuzzers tar inte hänsyn till vilken typ av mjukvara som testas och har svårigheter att hitta djupa och komplexa exekveringsvägar. VUzzer angriper problemet genom att skapa indata optimerad för mjukvarans struktur. Indata genereras genom en återkopplingscykel för mutering, som bland annat består av dynamisk dataflödesanalys av osäker data i kombination med en probabilistisk modell. Muteringsmetoden tillåter fuzzern att prioritera intressanta vägar och därmed hitta djupare och mer svåråtkomliga sårbarheter.

Publikationen visar att VUzzer har utvärderats på flera dataset och produktionsmjukvaror. Författarna menar att sårbarheter har hittats och rapporterats.

Zest

Nyckelord: Komplex indata

Commits	856	**Första commit**	2017-02-23	**Uppdaterat senast (år)**	2025
Contributors	22	**Forks**	120	**Stars**	709
Releases	16	**Branches**	18		
Listade CWE/CVE: Hittade buggar finns listade på projektets Github inklusive några CVEs.					
Projekt tillgängligt: https://github.com/rohanpadhye/jqf **DOI:** https://doi.org/10.1145/3293882.3330576					

Zest är en fuzzer för Java-mjukvaror som genererar indata av mer komplexa typer, exempelvis olika filformat [28]. Verktyget kombinerar fördelarna med både genereringsbaserade och täckningsstyrda fuzzers.

Genereringsbaserade fuzzers skapar ofta syntaktiskt korrekta filer, men kan missa semantiska fel, medan täckningsstyrda fuzzers använder återkoppling för att hitta nya kodvägar, men tenderar att producera syntaktiskt felaktig indata. Zest använder återkoppling från en täckningsstyrd fuzzer för att göra en QuickCheck-liknande generator deterministisk och parameterstyrd i stället för slumpmässig.[31] På så sätt styrs indatageneratorn att producera mer semantiskt korrekta filer.

Verktyget har utvärderats med vad som i publikationen beskrivs som välkända Java-benchmarks. Författarna menar att resultaten visar att Zest har en hög kodtäckning och lyckas hitta semantikbaserade buggar.

[31] https://pholser.github.io/junit-quickcheck/site/1.0/

3.3.3 Maskininlärning

Maskininlärningsbaserade tekniker är en övergripande kategori där någon form av maskininlärningsmodell används. I den tidigare studien användes begreppet mönsterigenkänning (eng. pattern recognition) för att beskriva tekniker där en maskininlärningsmodell drar slutsatser baserat på funna mönster. I den här studien används istället begreppet maskininlärning, då det är ett bredare begrepp som bättre stämmer överens med forskningslitteraturen.

Maskininlärning används ofta tillsammans med andra tekniker så som olika varianter av statisk och dynamisk analys. Djupinlärning (eng. deep learning) och graf-neurala nätverk (eng. graph-neural networks) används av verktyg beskrivna i denna studie och är maskininlärningsbaserade tekniker baserade på neurala nätverksarkitekturer.

FUNDED

Nyckelord: Flera språk, funktionsnivå

Commits	161	**Första commit**	2020-11-24	**Uppdaterat senast (år)**	2023
Contributors	5	**Forks**	36	**Stars**	128
Releases	0	**Branches**	1		
Listade CWE/CVE: Kan hitta sårbarheter av de 30 vanligaste CWE-kategorierna. Listas utförligt i publikationen.					
Projekt tillgängligt: https://github.com/HuantWang/FUNDED_NISL **DOI:** https://doi.org/10.1109/TIFS.2020.3044773					

Flow-sensitive vulnerability code detection (FUNDED) hittar mjukvarusårbarheter med styrda grafbaserade neurala nätverk (eng. Gated Graph Neural Network, GGNN) [20]. Författarna kombinerar två metoder: ett utökat grafbaserat neuralt nätverk samt en automatiserad metod för att hitta träningsdata. FUNDED bearbetar först koden till ett abstrakt syntaxträd vilket sedan används som indata till ett grafbaserat neuralt nätverk. I varje steg av bearbetningen utökas grafens kopplingar för att modellera en allt större mängd beroenden och därmed ge ett bättre utgångsläge för analysen.

Den automatiska metoden att hitta träningsdata till modellen bygger på probabilistiskt lärande och statistisk utvärdering.[32] Författarna till FUNDED beskriver att det är svårt att hitta bra träningsdata för maskininlärning och har därför valt att använda denna metod för att hitta sårbar kod i Github-projekt samt ur NVD:s sårbarhetsdatabas.[33]

FUNDED är baserat på Soot för att bygga syntaxträd för java-kod. För övriga språk används andra analysverktyg för att bygga syntaxträd, exempelvis Joern för C/C++.[34]

FUNDED har utvärderats på mjukvara skriven i C, Java, PHP och Swift.

[32] Träningsdata har samlats in med hjälp av en Mixture-of-experts-metod.
[33] NVD är en databas med mer omfattande information om olika CVE:er.
[34] https://joern.io/

SySeVR

Nyckelord: C/C++, källkod

Commits	175	**Första commit**	2018-07-18	**Uppdaterat senast (år)**	2024
Contributors	0	**Forks**	137	**Stars**	341
Releases	0	**Branches**	1		
Listade CWE/CVE: Modellen är tränad på CWE:er uppdelade i fyra kategorier kopplade till: API funktionsanrop, array-användning, pekare och aritmetriska uttryck.					
Projekt tillgängligt: https://github.com/SySeVR/SySeVR **DOI:** https://doi.org/10.1109/TDSC.2021.3051525					

Syntax-based, Semantics-based and Vector Representations (SySeVR) använder djupinlärning för att hitta sårbarheter i källkod [13]. SySeVR fokuserar på C/C++-kod, som konverteras till en vektorrepresentation för att bearbetas av en språkmodell. SySeVR använder så kallade Syntax-based Vulnerability Candidates (SyVC) och Semantics-based Vulnerability Candidates (SeVC). SyVC identifierar syntaktiska mönster som kännetecknar sårbar kod, medan SeVC utökar dessa genom att beakta semantiska beroenden i data- och kontrollflöden. SySeVR använder Joern som analysverktyg för bland annat C och C++.

Statiska algoritmer används för att bygga de syntax- och semantikbaserade kandidaterna och för att koda dem till vektorer. Därefter har ett dubbelriktat återkommande neuralt nätverk (eng. Bidirectional recurrent neural network, BiRNN) tränats på vektorerna. Eftersom SySeVR har tränats på vektorer behöver alla mjukvaror som ska analyseras konverteras till motsvarande vektorer.

I publikationen visas att SySeVR har hittat 15 sårbarheter som inte tidigare rapporterats.

3.3.4 Symboliska exekveringsverktyg

Symbolisk exekvering (eng. symbolic execution) analyserar målsystem genom att ersätta indata med variabler (symboler) som representerar godtyckliga värden [35]. Symbolisk exekvering kan vara såväl statisk som dynamisk [43].

Vid symbolisk exekvering modelleras målsystemet genom formler som beskriver vägen från en startpunkt till andra punkter i källkoden och där formlerna opererar på de symboliska värdena. Lösningen på respektive formel beskriver de potentiella indata som kan användas för att nå respektive punkt. Tekniken kan därmed användas för att visa att en given punkt kan nås med ogiltig indata.

Symbolisk exekvering är en mycket populär teknik eftersom den kan identifiera brister som ligger djupt i ett systems logik och kod. Tekniken är dock väldigt beräkningsintensiv, vilket gör den kostsam att använda.

Angr*

Nyckelord: Binärer, större ramverk

Commits	13650	**Första commit**	2013-08-16	**Uppdaterat senast (år)**	2025
Contributors	255	**Forks**	1100	**Stars**	8200
Releases	243	**Branches**	199		
Listade CWE/CVE: Inte listade i publikationen. Ramverket är stort och innehåller många olika metoder för analys.					
Projekt tillgängligt: https://github.com/angr/angr **Projekthemsida:** https://angr.io/ **DOI:** https://doi.org/10.1109/SP.2016.17					

Angr är en plattform för analys av binärer och erbjuder många olika funktioner, till exempel kontrollflödesanalys, återskapande av krascher samt generering av angreppskod. Huvudsakligen erbjuder Angr sårbarhetsanalys genom olika symboliska exekveringsmetoder. Bland annat har Angr ursprungligt stöd för dynamisk symbolisk exekvering och fuzzning med stöd av symbolisk exekvering. Författarna har konstruerat plattformen för att möjliggöra tillägg av andra analysmetoder. Tanken är att olika metoder ska kunna jämföras på ett likvärdigt sätt.

Publikationen beskriver att Angr har få beroenden av andra verktyg. Istället har inspiration tagits av andra arbeten för egna implementationer. Ett exempel är en subkategori av symbolisk exekvering (under-constrained symbolic execution) där inspiration tagits från verktyget KLEE. Fuzzern som används i Angr är Driller. Driller är också utvecklad av samma författargrupp men presenteras i en annan publikation. Angr använder biblioteket libVEX från Valgrind för att lyfta binärkod till en intermediärrepresentation.[35]

[35] https://valgrind.org/

Inception*

Nyckelord: Inbyggda system

Commits	59	**Första commit**	2018-06-20	**Uppdaterat senast (år)**	2022
Contributors	2	**Forks**	1	**Stars**	25
Releases	0	**Branches**	1		
Listade CWE/CVE: Inga CVE:er listas.					
Projekt tillgängligt: https://github.com/Inception-framework/ **Projekthemsida:** https://inception-framework.github.io/inception/					

Inception är ett ramverk för att analysera inbyggd mjukvara [26]. Författarna framhåller att inbyggda system bör studeras ur ett helhetsperspektiv, där källkod och binärkod analyseras gemensamt tillsammans med tänkt hårdvara. Inception möjliggör detta genom att översätta källkoden till LLVM:s intermediärrepresentation och genom att lyfta assembler- och binärkod till samma format. För detta används en egenutvecklad intermediärrepresentation [33].

Verktyget använder sig av KLEE för dynamisk symbolisk exekvering. Författarna har dock utökat KLEE för minneshantering och för att hantera hårdvarans avbrottsrutiner samtidigt som analysen görs. Inception använder även Clangs kompilator.[36]

Publikationen visar att Inception hittat åtta krascher och två nya sårbarheter.

[36] Clang är en öppen källkodskomponent i LLVM-projektet och används ofta i forskningsverktyg för programanalys och fuzzning. Se https://clang.llvm.org/.

Sydr*

Nyckelord: C/C++, hybridfuzzer

Commits	589	**Första commit**	2021-10-11	**Uppdaterat senast (år)**	2025
Contributors	20	**Forks**	36	**Stars**	141
Releases	0	**Branches**	3		
Listade CWE/CVE: Inga CVE:er är listade i publikationen, men författarna skriver att verktyget kan hitta nullpekarderefererингar, division med noll, åtkomst utanför minnesgränser och heltalsöverflödssårbarheter.					
Projekt tillgängligt: https://github.com/ispras/oss-sydr-fuzz **Projekthemsida:** https://sydr-fuzz.github.io/ **DOI:** https://doi.org/10.1109/ISPRAS53967.2021.00016					

Sydr är ett verktyg som använder sig av dynamisk symbolisk exekvering för att testa mjukvara. Det är uppbyggt av två andra verktyg: Triton[37] och DynamoRIO.[38]

Triton används i Sydr för den symboliska analysen, medan DynamoRIO står för den dynamiska exekveringen av programmet som analyseras. Författarna har integrerat dessa komponenter och menar att de har vidareutvecklat Tritons symboliska exekvering med nya tekniker som syftar till att förbättra analysens precision och prestanda

Det har publicerats flera uppföljande studier av Sydr. Som exempel publicerades 2021 en studie som går igenom hur författarna vidareutvecklat Sydr för att upptäcka fler sårbarhetstyper [15].

Verktyget utvärderades i den ursprungliga publikationen utifrån hur korrekta genererade indata var, hur många kodgrenar som verktyget utforskat och hur lång tid analysen tog [14]. Under uppföljningsstudien genomfördes kompletterande tester på hur väl verktyget upptäckte sårbarheter i en testsvit [15].

[37] https://triton-library.github.io/
[38] https://dynamorio.org/

3.3.5 Övriga dynamiska analysverktyg

Avsnittet beskriver de dynamiska verktyg som inte använder sig av dynamisk flödesanalys, fuzzning eller dynamisk symbolisk exekvering.

Flera av verktygen använder sig av instrumentering. Det innebär att segment läggs till i koden för att övervaka vad som sker i den mjukvara som testas. Ett enkelt exempel är att infoga kodrader som loggar när en viss funktion startar och slutar, för att kunna mäta hur ofta den körs och hur lång tid den tar.

Instrumentering kan implementeras på olika sätt och användas under olika faser av testningen, till exempel i binärkoden eller i kodens intermediärrepresentation. Den kan även utföras dynamiskt under körning [36]. Ett av de mest välkända verktygen som använder instrumentering är Valgrind.[39]

[39] https://valgrind.org/

C11Tester

Nyckelord: C/C++, trådningsbuggar

Commits	2088	**Första commit**	2012-04-08	**Uppdaterat senast (år)**	2024
Contributors	3	**Forks**	4	**Stars**	11
Releases	0	**Branches**	1		
Listade CWE/CVE: Inga CVE:s är listade i publikationen men verktyget letar sårbarheter gällande trådning.					
Projekt tillgängligt: https://github.com/c11tester/c11tester **DOI:** https://doi.org/10.1145/3445814.3446711					

C11Tester letar efter trådrelaterade buggar i C/C++ mjukvaror [16], såsom trådkapplöpning (eng. race condition) och andra samtidighetsproblem (eng. concurrency bugs). Verktyget använder sig av en restriktionsbaserad algoritm (eng. constraint-based algorithm) där en graf konstrueras med alla möjliga relationer mellan mjukvarans trådoperationer. Grafen används sedan för att schemalägga alla giltiga trådarna och identifiera situationer där trådkapplöpning uppstår.

Verktyget använder ett eget bibliotek för trådhantering där instrumentering är inbyggd för exempelvis övervakning. Det använder LLVM och Clang för kompilering och instrumentering.

Verktyget har utvärderats på produktionsmjukvaror i en jämförelse med andra verktyg. I samtliga produktionsmjukvaror hittades buggar varav flertalet bara hittades av C11Tester. Publikationen om C11Tester nämner inte några explicita sårbarheter utan fokuserar på samtidighetsproblem.

Fuzztruction

Nyckelord: Komplex/krypterad/komprimerad indata

Commits	11	**Första commit**	2022-10-11	**Uppdaterat senast (år)**	2024
Contributors	0	**Forks**	23	**Stars**	132
Releases	0	**Branches**	1		
Listade CWE/CVE: Verktyget har hittat nya sårbarheter som har tilldelats CVE-nummer: 2021-4217, 2022-0530, 2022-0529 och 2022-1304.					
Projekt tillgängligt: https://github.com/fuzztruction/fuzztruction					

Fuzztruction fuzztestar mjukvara som tar emot komplex, krypterad eller komprimerad indata [32]. Författarna bakom Fuzztruction menar att många sådana mjukvaror (mottagande) får sin indata från andra mjukvaror (sändande). Verktyget utnyttjar detta genom att modifiera den sändande mjukvaran för att skapa mutationer i den data som skickas vidare till det mottagande mjukvaran.

Fuzztruction injicerar fel i den sändande mjukvaran, där typen och omfattningen av mutationer kan variera. Fuzzningen består i att upprepade gånger se hur små förändringar påverkar sändarens utdata och därmed den mottagande mjukvarans funktion. Verktyget använder LLVM-moduler och Just-In-Time (JIT)-kompilering för instrumentering, att införa mutationer och övervakning i mjukvaran.

Fuzztruction har testats på produktionsmjukvaror och jämförts med andra fuzzers. Publikationen visar att verktyget rapporterat 27 buggar och hittat fyra nya sårbarheter som tilldelats CVE-nummer. I jämförelserna visar Fuzztruction större kodtäckning, särskilt för mjukvara som använder kryptering.

Fuzztrunction är utvecklat av samma forskargrupp som utvecklat Fuzzware och Houdur.

MOVEC

Nyckelord: C, minnessårbarheter

Commits	6	**Första commit**	2021-04-28	**Uppdaterat senast (år)**	2024
Contributors	0	**Forks**	1	**Stars**	22
Releases	2	**Branches**	1		
Listade CWE/CVE: Inte listade i artiklarna. Vektyget fokuserar på minnessårbarheter.					
Projekt tillgängligt: https://github.com/drzchen/movec **DOI:** https://doi.org/10.1145/3460319.3464807, https://doi.org/10.1109/TSE.2022.3210580					

MOVEC är ett dynamiskt analysverktyg som används för att identifiera minnesrelaterade fel och minnesrelaterade sårbarheter i C-kod [5]. Verktyget använder sig av dynamisk instrumentering för att introducera övervakande kod i mjukvaran.

MOVEC infogar övervakningskod direkt i källkoden för att möjliggöra analys oberoende av plattform, kompilator och optimeringsnivå [5]. Instrumenteringen utförs med Clang. Författarna kontrasterar detta mot många andra verktyg som infogar övervakningskod i binären eller i den intermediära representationen, vilket kan göra analysen känslig för kompilatorval, optimeringsgrad och målplattform.

Publikationen visar hur MOVEC utvärderas på flera dataset med goda resultat. Verktyget hittade fler minnesfel vid jämförelse med andra verktyg.

3.3.6 Övriga statiska analysverktyg

Sektionen beskriver övriga statiska verktyg, de som inte huvudsakligen bygger på symbolisk exekvering eller dataflödesanalys.

MIRCHECKER

Nyckelord: Rust, minnessårbarheter

Commits	11	**Första commit**	2021-10-25	**Uppdaterat senast (år)**	2024
Contributors	4	**Forks**	27	**Stars**	154
Releases	0	**Branches**	1		
Listade CWE/CVE: Anger inte.					
Projekt tillgängligt: https://github.com/lizhuohua/rust-mir-checker **DOI:** https://doi.org/10.1145/3460120.3484541					

MIRCHECKER använder abstrakt tolkning för att hitta minnessårbarheter i Rust-kod. Tekniken använder mjukvarans kontrollflödesgraf, där varje nod har en förflyttningsmetod som tar ett approximerat tillstånd som indata och returnerar ett nytt tillstånd [44]. Abstrakt tolkning överapproximerar de möjliga programtillstånden i varje steg och utgör grunden för dataflödesanalys, som kan ses som en specialiserad form av abstrakt tolkning.

Verktyget analyserar Rust-kod under kompilering genom att operera på språkets mellannivårepresentation, MIR (Mid-level Intermediate Representation) [18]. Det kombinerar numerisk statisk analys med en variant av symbolisk exekvering. Den numeriska analysen används för att undersöka koden, medan den symboliska exekveringen fördjupar analysen av de koddelar som identifierats som relevanta. Enligt författarna är denna kombination mer resurseffektiv än traditionell symbolisk exekvering, eftersom numeriska, abstrakta domäner vanligtvis är enklare att lösa än symboliska formler.

Mirchecker har testats på över 1 000 Rust-baserade mjukvaror, där 16 tidigare okända minnessårbarheter identifierats i 12 av dem.

4 Diskussion

Kapitlet diskuterar verktygsforskningens begränsningar, brister i tillgängliga sårbarhetssamlingar, tekniktrender inom området samt vad som behövs för att de ska bli praktiskt användbara. Kapitlet avslutas med en metoddiskussion.

4.1 Verktygsforskningens begränsningar

Trots ett mycket aktivt forskningsområde och många vetenskapliga publikationer är det få publicerade verktyg som identifierats som intressanta för fördjupade studier i projektet. De verktyg som identifierats, antingen för att de är mogna eller där relevant utvecklingsarbete skett, täcker också bara ett relativt litet område av möjliga tekniker, målsystem och sårbarhetstyper.

En delförklaring till verktygens begränsningar går troligtvis att finna i att studien avgränsar sig till vetenskapligt publicerade verktyg. Till skillnad från ett kommersiellt utvecklat verktyg syftar inte nödvändigtvis en akademisk publicering till att utvecklas bortom att experimentellt bevisa ett koncept (TRL 3), än mindre att nå marknaden. Att verktyg överges efter publicering kan möjligtvis förklaras av samma skäl: Att de har utvecklats inom ett begränsat forskningsprojekt med andra syften än att påbörja ett långsiktigt utvecklingsarbete.

Det finns noterbara undantag från observationen om begränsningen med vetenskapligt publicerade verktyg. Det tydligaste är PhASAR, utvecklat av en grupp forskare som tidigare presenterat verktyg för att identifiera sårbarheter i Java. I PhASAR har de tagit en bredare ansats, med en modulär konstruktion och där målmjukvaran omtolkas till LLVM:s intermediärrepresentation innan analys. Deras mål har varit ett verktyg som kan analysera en större bredd av sårbarheter och målmjukvaror.

4.2 Högproduktiva forskargrupper

Vid analys av publikationerna kan noteras att ett antal forskargrupper är mycket produktiva i sin utveckling av verktyg för att identifiera sårbarheter och buggar i mjukvara. Dessa forskargrupper är verksamma vid institutioner i USA, Kina, Tyskland, Singapore, Australien och Ryssland.

Det finns många kopplingar mellan olika arbeten inom området, exempelvis genom referenser till centrala arbeten samt att nya verktyg ofta bygger vidare på andras verktyg. I publikationer om fuzzerverktyg refereras nästan alltid publikationer om Driller, LibFuzzer, Klee och Peach, men även AFLGo, AFLSmart och AFLNet. LLVM-tekniken och verktyg som AFL, Angr, Driller, Klee och Soot används återkommande som byggstenar för nya verktyg.

Givet att de dominerande forskargrupperna fortsätter att vara produktiva kan en bevakning av deras produktion, och av arbeten där de refereras, antagligen ge en översiktlig bild av forskningsområdet och dess utveckling.

4.3 Sårbarhetssamlingarnas brister

En begränsade faktor för utvecklingen av verktyg är tillgången till publika, uppdaterade och relevanta testfall för utvärdering. I en inventering av testfallssamlingar som genomförts i projektet konstateras att det finns en omfattande kritik mot bland annat samlingarnas täckningsgrad [45]. Samlingarnas sammansättning styr sannolikt forskningen mot programspråk som C/C++ snarare än Java, Python och Rust. De inkluderar också en begränsad mängd av sårbarhetstyper och sårbarheter.

Att bygga upp nya och bättre, publika sårbarhetssamlingar, med en högre kvalitet och en större täckningsgrad, är resurskrävande och utan uppenbara vinster för den som investerar i arbetet. Det är sannolikt dock en förutsättning för att bredda den akademiska forskningens inriktning mot nya tekniker, målsystem och sårbarhetstyper.

4.4 Tekniktrender inom området

I den tidigare studien noterades att det fanns två nya, växande, forskningstrender under senare år [2]. För det första fanns det en stark tillväxt av hybridverktyg, där flera tekniker kombineras för att nå bättre resultat. För det andra så fanns det ett experimenterande med maskininlärningsmetoder. I den här rapporten kan ytterligare två trender utrönas. Den första är fuzzningsteknikens dominans inom forskningsområdet. Den andra är att verktygsutveckling ofta bygger vidare på befintliga verktyg.

Nästan alla verktyg som identifierats som intressanta för fortsatta studier bygger på någon form av hybridansats. Att kombinera olika tekniker, exempelvis att använda en maskininlärningsmodell som input till en fuzzer, är vanligt. Genom att kombinera tekniker så når verktygen längre än vad enstaka tekniker gör på egen hand. Hybridverktyg har gått från experimenterande till att bli det nya normala.

Utfallet av intressanta verktyg som primärt använder maskininlärning (ML), som mer än input till ett annat verktyg, är begränsat. Inget av de verktyg som bedömts med hög mognadsgrad är ett rent ML-baserat verktyg. I den här studien har dock FUNDED och SySeVR identifierats som intressanta baserat på projektens aktivitetsnivå efter ursprunglig publicering. Användningen av ML-tekniker för att identifiera mjukvarusårbarheter får ännu betraktas som omogen. Det kan ta tid att bygga välfungerande verktyg och det finns en risk att utvecklingen drivs av forskare som fokuserar mer på ML-modellerna än förmågan att identifiera mjukvarusårbarheter.

Sårbarhetssamlingarnas brister kan särskilt påverka utvecklingen av ML-verktyg, då de utgör modellernas träningsdata. Felklassificerade dataset är vanligt [45]. Det behövs sannolikt sårbarhetssamlingar av en bättre kvalitet och med en bättre täckning av sårbarheter och sårbarhetstyper för att ML-verktygen ska ta nästa steg i sin utveckling.

Fuzzningstekniken dominerar inte bara forskningsområdet, utan också rapportens lista över verktyg som bedömts intressanta för fortsatta studier. Tekniken är särskilt lämpad för att automatiskt identifiera kraschrelaterade sårbarheter, minnesfel och andra oväntade programbeteenden, men mindre lämpad för att upptäcka logiska eller kontextberoende brister som kräver djupare semantisk förståelse av koden. Dess starka ställning innebär därför att forskningsfältet i hög

grad fokuserar på just dessa typer av sårbarheter. En möjlig förklaring till fuzzerdominansen kan vara att utveckling av nya verktyg inom forskningen ofta bygger vidare på existerande verktyg från andra forskargrupper.

Att det finns bra verktyg att använda som utgångspunkt för forskningsarbete verkar bidra positivt till framväxten av nya verktyg anpassade mot nya sårbarheter. Bland studiens verktyg med hög mognadsgrad (TRL 8-9) bygger alla vidare på andra verktyg. Det tydligaste exemplet är fuzzerverktyget AFL som är utgör grunden för AFLSmart, AFLGo, AFLNet och AFL++. Fuzzware bygger på både AFL och AFL++. AFL++ utgör i sin tur också grunden för PolyFuzz. Även om bottenplattan, AFL, är gemensam så finns det en stor spridning i funktion hos de vidareutvecklade verktygen, där exempelvis Fuzzware fokuserar på inbyggd mjukvara medan AFLNet används för att testa servrar.

4.5 Vad behövs för att ett verktyg ska vara användbart?

Användbarhet är en faktor för att verktyg ska få praktiskt genomslag. Det är dock svårt att entydigt definiera vad användbarhet är. Ett verktyg behöver ha en hög teknisk förmåga när det gäller att identifiera en bredd av sårbarhetstyper och sårbarheter. Det behöver också upplevas som lätt att hantera och vara integrerbart i utvecklarens arbetsflöde. Verktygen behöver alltså klara komplexa uppgifter och samtidigt vara enkla, vilket kan utgöra motsatsförhållanden.

I studien är det få verktyg som är designade med användaren i fokus. Det är också få publikationer som diskuterar verktygen i termer av användbarhet. De är ofta prototyper eller forskningsverktyg som inte är ämnade att bli färdiga produkter. Teknisk funktionalitet, inom sitt begränsade område, prioriteras framför enkelhet och användarvänlighet. Utifrån publikationerna är det därför svårt att dra tydliga slutsatser om verktygens praktiska värde.

Två verktyg som exemplifierar den möjliga motsättningen mellan komplexitet och användarvänlighet är QuickFuzz och Zest. Verktygen använder liknande tekniker för att lösa ett delvis liknande problem, men deras angreppssätt är varandras motsatser. QuickFuzz har ett stort fokus på att automatisera arbetet med en lägre

tröskel för användaren. Zest tillåter mer detaljstyrning och kräver därför mer av användaren.

På Github har Zest, det verktyg som erbjuder en mer komplex detaljstyrning, tre gånger så många stjärnmarkeringar[40] som QuickFuzz. Det går inte att dra långtgående slutsatser av antalet stjärnor för två verktyg på Github, men det verkar dock indikera att användarna inte skyr möjligheten att styra och förstå verktyget i detalj, även när det går ut över användarvänligheten.

Verktyget PhASAR är också ett undantag på användbarhetsområdet [4]. Publikationen är nästan uteslutande skriven utifrån användarens perspektiv. Verktyget beskrivs som flexibelt och anpassningsbart till respektive användares behov. Det använder också ett gränssnitt som avsiktligt ska reducera den komplexitet som användaren ställs inför.

Svårigheten med att entydigt beskriva vad som är användbarhet i termer av kodanalysverktyg har visats också i tidigare arbeten från FOI. I litteraturstudien *Varför har mjukvaror sårbarheter?* av Karlzén m.fl. lyfts flera problemområden relaterat till utvecklares användning av kodanalysverktyg [46]. Här beskrivs bl.a. utvecklarens erfarenhet som viktig för om kodanalysverktyg ska göra nytta, men också problemet med att inget verktyg kan hitta sårbarheter av alla typer. I den uppföljande enkätstudien *Mjukvarors säkerhet beror på utvecklarnas motivationer och hinder* av Karlzén m.fl., riktad mot svenska utvecklare inom säkerhetskritisk verksamhet, bekräftas däremot inte den bilden [47]. Av enkätstudien är det svårt att dra någon slutsats om utvecklares syn på kodanalysverktyg och deras användbarhet.

Vad som gör verktyg användbara och vad som skulle kunna öka deras användbarhet är svårt att utröna från den forskning som denna studie bygger på. Användbarhet verkar dock bygga på en balans mellan enkel användning, möjlighet till detaljerad anpassning, bredd av upptäcktsförmåga samt integration i utvecklingsprocessen. Ett stärkt fokus på mognad kan bidra till att verktyg utvecklas mot en mer produktliknande form, vilket i sin tur kan öka deras användbarhet. Samtidigt är det troligt att ett direkt fokus på användbarhet är avgörande för att fler verktyg ska nå hög mognad och accepteras av fler utvecklare. Mognad och användbarhet påverkar sannolikt varandra ömsesidigt.

[40] Att en användare gillar eller vill spara projektet.

4.6 Metoddiskussion

Den tidigare studien tog sin ansats i att bedöma ett verktygs teknologiska mognad baserat på dess akademiska publikation. Publikationen ger en indikation på ett verktygs kvalitet, men det är en bedömning som begränsar sig till tillfället för publicering. Mognadsskattningen tar inte hänsyn till om hur verktyget utvecklats, eller inte utvecklats, efter publiceringen.

Den här utökade studien tar istället fasta på verktygens utveckling. En urvalsprocess tillämpades på de verktyg som skattades med medelhög mognad (TRL 5-7) för att identifiera verktyg som utvecklats meningsfullt sedan deras ursprungliga publicering. Genom att betrakta andra perspektiv än en publikations mognad så breddas utfallet av potentiellt intressanta verktyg.

Urvalsprocessens resultat ska inte övertolkas. Att ett verktyg uppfyller processens kriterier visar enbart att det är tillgängligt och att det bedrivits en aktiv vidareutveckling. Urvalsprocessen är ett filter för att identifiera verktyg som uppvisar tecken på att ha ökat i mognadsgrad sedan ursprunglig publicering och därmed är särskilt intressanta för vidare utvärdering. Den utgör i sig inte en bedömning av verktygets kvalitet.

Det går heller inte att med säkerhet uttala sig om kvaliteten för de verktyg som exkluderas i urvalsprocessen. Att ett verktyg är otillgänglig i dag behöver inte betyda att det är ett dåligt verktyg. Ett verktyg kan också ha utvecklats i en miljö som inte är tillgänglig via Github, exempelvis i en kommersiell kontext.

Beslutskriterier som baseras på datum medför skarpa gränser som inte tar hänsyn till om en uppdatering hamnat någon dag på fel sida om gränsen. I urvalsprocessen valdes gränser som var väl tilltagna, men det kan inte uteslutas att enstaka verktyg exkluderats trots att de varit relevanta för fortsatta studier.

4.7 Framtida arbete

För att med någon säkerhet kunna uttala sig om ett verktygs förmåga och praktiska nytta behövs praktisk testning, vilket är nästa steg i projektet. Utgångspunkten för testerna är den lista med 27 verktyg som identifierats genom

mognadsskattningar och urvalsprocessen. Då kapaciteten att testa verktyg är begränsad vore det önskvärt att hitta ett objektivt mått som rangordnar potentialen för dessa verktyg. En ansats till metod för en sådan rangordning beskrivs i bilaga B. Det kan ifrågasättas om jämförande verktygstester ens är relevanta i praktisk mening, åtminstone för att rangordna verktygen sinsemellan. Identifierade verktyg i studien använder olika tekniker och fokuserar på olika målsystem. Att jämföra verktyg är möjligt, inte minst inom olika teknikgrupper. Att avgöra vad som är det överlag bästa verktyget kan däremot vara som att försöka svara på vad som är bäst av en hammare och en skruvmejsel, utan att ta hänsyn till uppgiften som ska lösas.

5 Slutsatser

Projektet *Tekniker för att identifiera mjukvarusårbarheter* har i två rapporter granskat vetenskapligt publicerade verktyg som kan identifiera mjukvarusårbarheter. Forskningsområdet bedöms som mycket aktivt, med 237 publicerade verktyg under perioden 2014–2024. Fokus för den här rapporten handlar om vilka av dessa verktyg projektet ska studera djupare, genom att besvara forskningsfrågan:

Vilka vetenskapligt publicerade verktyg som identifierar mjukvarusårbarheter är intressanta att utvärdera praktiskt?

Rapporten beskriver 27 verktyg som bedöms intressanta för fortsatta studier genom praktisk utvärdering. Dessa har identifierats genom att granska vetenskapligt publicerade verktyg från perioden 2014-–2024. Projektets granskningsprocess har följt två spår fördelat över två studier:

- Det första spåret, som beskrivs i den tidigare studien, fokuserar enbart på verktygens bedömda teknologiska mognadsgrad. I forskargruppens arbete skattades 7 verktyg som att de höll en hög teknologisk mognadsgrad (TRL 8–9). Dessa är intressanta för fortsatta studier.
- Det andra spåret, som redovisas i den här rapporten, genomförs en förnyad granskning av de verktyg som skattats till en medelhög teknologisk mognadsgrad (TRL 5–7). Arbetet studerar hur verktygen utvecklats efter sin initiala akademiska publicering, genom att bedöma aktivitetsnivån i respektive Github-projekt. Till listan av intressanta verktyg fogas 20 verktyg där det bedöms att en relevant utveckling skett.

Det är noterbart att forskningsområdet domineras av ett fåtal forskargrupper, vars produktion står för 12 av 27 identifierade verktyg i studien. Två av dessa grupper står även för ytterligare tre verktyg som återkommande används som komponenter i andra identifierade verktyg.

Arbetet med de 27 identifierade verktygen övergår nu i en praktisk fas. Projektets metoder pekar på att de har en potential för praktisk användning. För att verifiera

verktygens kvalitet – deras applicerbarhet, deras effektivitet och deras användbarhet – behöver verktygen testas praktiskt.

Referenser

[1] Försvarsberedningen, “Kraftsamling: Inriktningen av totalförsvaret och utformningen av det civila försvaret”, Försvarsdepartementet, Ds 2023:34, 2023.

[2] C. Gustavsson, C. Vestlund, V. Andersson, D. Eidenskog, L. Nyholm och C. Jensen, “Tekniker och verktyg som identifierar mjukvarusårbarheter: En skanning av forskning publicerad mellan 2014 – 2024”, Totalförsvarets forskningsinstitut, FOI-R--5692--SE, 2024. https://www.foi.se/rapporter/rapportsammanfattning.html?reportNo=FOI-R–5692–SE.

[3] J. C. Mankins, “Technology readiness levels: A white paper”, NASA, White Paper, 1995.

[4] P. D. Schubert, B. Hermann och E. Bodden, “PhASAR: An Inter-procedural Static Analysis Framework for C/C++”, i *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar och L. Zhang, utg., Cham: Springer International Publishing, 2019, s. 393–410.

[5] Z. Chen, Q. Zhang, J. Wu, J. Yan och J. Xue, “A Source-Level Instrumentation Framework for the Dynamic Analysis of Memory Safety”, *IEEE Transactions on Software Engineering*, årg. 49, nr 4, s. 2107–2127, 2023. https://doi.org/10.1109/TSE.2022.3210580.

[6] Z. Chen, C. Wang, J. Yan, Y. Sui och J. Xue, “Runtime detection of memory errors with smart status”, i *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021, Association for Computing Machinery, 2021, s. 296–308. https://doi.org/10.1145/3460319.3464807.

[7] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel och G. Vigna, “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”, i *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, s. 138–157. https://doi.org/10.1109/SP.2016.17.

[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida och H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing", i *Proceedings 2017 Network and Distributed System Security Symposium*, ser. NDSS 2017, Internet Society, 2017. https://doi.org/10.14722/ndss.2017.23404.

[9] 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. https://doi.org/10.1109/SP46214.2022.9833613.

[10] M. Böhme, V.-T. Pham, M.-D. Nguyen och A. Roychoudhury, "Directed Greybox Fuzzing", i *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, ACM, 2017-10, s. 2329–2344. https://doi.org/10.1145/3133956.3134020.

[11] C. T. Tran och S. Kurmangaleev, "Futag: Automated fuzz target generator for testing software libraries", i *2021 Ivannikov Memorial Workshop (IVMEM)*, IEEE, 2021-09. https://doi.org/10.1109/ivmem53963.2021.00021.

[12] J. Chen, J. Wang, C. Song och H. Yin, "JIGSAW: Efficient and Scalable Path Constraints Fuzzing", i *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022-05, s. 18–35. https://doi.org/10.1109/sp46214.2022.9833796.

[13] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu och Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities", *IEEE Transactions on Dependable and Secure Computing*, årg. 19, nr 4, s. 2244–2258, 2022-07. https://doi.org/10.1109/tdsc.2021.3051525.

[14] A. Vishnyakov, V. Logunova, E. Kobrin, D. Kuts, D. Parygina och A. Fedotov, "Symbolic Security Predicates: Hunt Program Weaknesses", i *2021 Ivannikov Ispras Open Conference (ISPRAS)*, 2021, s. 76–85. https://doi.org/10.1109/ISPRAS53967.2021.00016.

[15] A. Vishnyakov, A. Fedotov, D. Kuts, A. Novikov, D. Parygina, E. Kobrin, V. Logunova, P. Belecky och S. Kurmangaleev, "Sydr: Cutting Edge Dynamic Symbolic Execution", i *2020 Ivannikov Ispras Open Conference (ISPRAS)*, 2020, s. 46–54. https://doi.org/10.1109/ISPRAS51486.2020.00014.

[16] W. Luo och B. Demsky, “C11Tester: a race detector for C/C++ atomics”, i *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’21, ACM, 2021-04. https://doi.org/10.1145/3445814.3446711.

[17] M. Cui, C. Chen, H. Xu och Y. Zhou, “SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis”, *ACM Transactions on Software Engineering and Methodology*, årg. 32, nr 4, s. 1–21, 2023-05. https://doi.org/10.1145/3542948.

[18] Z. Li, J. Wang, M. Sun och J. C. Lui, “MirChecker: Detecting Bugs in Rust Programs via Static Analysis”, i *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, s. 2183–2196. https://doi.org/10.1145/3460120.3484541.

[19] 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, 2023-06, s. 1379–1396.

[20] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian och Z. Wang, “Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection”, *IEEE Transactions on Information Forensics and Security*, årg. 16, s. 1943–1958, 2021. https://doi.org/10.1109/tifs.2020.3044773.

[21] S. Chen, Y. Zhang, L. Fan, J. Li och Y. Liu, “AUSERA: Automated Security Vulnerability Detection for Android Apps”, i *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22, Rochester, MI, USA: Association for Computing Machinery, 2023. https://doi.org/10.1145/3551349.3559524.

[22] L. Chen, Y. Wang, J. Linghu, Q. Hou, Q. Cai, S. Guo och Z. Xue, “SaTC: Shared-Keyword Aware Taint Checking for Detecting Bugs in Embedded Systems”, *IEEE Transactions on Dependable and Secure Computing*, årg. 21, nr 4, s. 2421–2433, 2024-07. https://doi.org/10.1109/tdsc.2023.3307430.

[23] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz och A. Abbasi, “Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing”, i *31st USENIX Security*

Symposium (USENIX Security 22), Boston, MA: USENIX Association, 2022-06, s. 1239–1256.

[24] L. Chen, Q. Cai, Z. Ma, Y. Wang, H. Hu, M. Shen, Y. Liu, S. Guo, H. Duan, K. Jiang och Z. Xue, "SFuzz: Slice-based Fuzzing for Real-Time Operating Systems", i *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, ACM, 2022-11, s. 485–498. https://doi.org/10.1145/3548606.3559367.

[25] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel och T. Holz, "HOEDUR: embedded firmware fuzzing using multi-stream inputs", i *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23, Anaheim, CA, USA: USENIX Association, 2023.

[26] 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, 2018-08, s. 309–326.

[27] P. Srivastava, F. Toffalini, K. Vorobyov, F. Gauthier, A. Bianchi och M. Payer, "Crystallizer: A Hybrid Path Analysis Framework to Aid in Uncovering Deserialization Vulnerabilities", i *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '23, ACM, 2023-11, s. 1586–1597. https://doi.org/10.1145/3611643.3616313.

[28] R. Padhye, C. Lemieux, K. Sen, M. Papadakis och Y. Le Traon, "Semantic fuzzing with zest", i *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '19, ACM, 2019-07, s. 329–340. https://doi.org/10.1145/3293882.3330576.

[29] V.-T. Pham, M. Bohme och A. Roychoudhury, "AFLNET: A Greybox Fuzzer for Network Protocols", i *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, 2020-10, s. 460–465. https://doi.org/10.1109/icst46399.2020.00062.

[30] G. Grieco, M. Ceresa och P. Buiras, "QuickFuzz: an automatic random fuzzer for common file formats", i *Proceedings of the 9th International Symposium on Haskell*, ser. ICFP'16, ACM, 2016-09, s. 13–20. https://doi.org/10.1145/2976002.2976017.

[31] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu och A. Roychoudhury, “Smart Greybox Fuzzing”, *IEEE Transactions on Software Engineering*, årg. 47, nr 9, s. 1980–1997, 2021. https://doi.org/10.1109/TSE.2019.2941681.

[32] N. Bars, M. Schloegel, T. Scharnowski, N. Schiller och T. Holz, “Fuzztruction: using fault injection-based fuzzing to leverage implicit domain knowledge”, i *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC ’23, Anaheim, CA, USA: USENIX Association, 2023.

[33] *The LLVM Compiler Infrastructure*, `https://llvm.org/` [Besökt: 2026-09-22].

[34] L. D. Fosdick och L. J. Osterweil, “Data flow analysis in software reliability”, *ACM Computing Surveys (CSUR)*, årg. 8, nr 3, s. 305–330, 1976.

[35] A. Adhikari och P. Kulkarni, “Survey of techniques to detect common weaknesses in program binaries”, *Cyber Security and Applications*, årg. 3, s. 100 061, 2025. https://doi.org/10.1016/j.csa.2024.100061.

[36] T. Sutter, T. Kehrer, M. Rennhard, B. Tellenbach och J. Klein, “Dynamic Security Analysis on Android: A Systematic Literature Review”, *IEEE Access*, årg. 12, s. 57 261–57 287, 2024. https://doi.org/10.1109/ACCESS.2024.3390612.

[37] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu och L. Xu, “An empirical assessment of security risks of global Android banking apps”, i *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20, Seoul, South Korea: Association for Computing Machinery, 2020, s. 1310–1322. https://doi.org/10.1145/3377811.3380417.

[38] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel och G. Vigna, “KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware”, English, i *Proceedings 2020 IEEE Symposium on Security and Privacy, SP 2020*, 41st IEEE Symposium on Security and Privacy 2020 ; Conference date: 18-05-2020 Through 20-05-2020, United States: IEEE, 2020-05,

s. 1544–1561. https://doi.org/10.1109/SP40000.2020.00036. http://www.ieee-security.org/TC/SP2020/.

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

[40] S. Acharya och V. Pandya, "Bridge between black box and white box–gray box testing technique", *International Journal of Electronics and Computer Science Engineering*, årg. 2, nr 1, s. 175–185, 2012.

[41] P. D. Marinescu och C. Cadar, "KATCH: High-Coverage Testing of Software Patches", i *European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)*, Saint Petersburg, Russia, 2013-08, s. 235–245.

[42] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel m. fl., "Driller: Augmenting Fuzzing Through Selective Symbolic Execution", i *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[43] 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. https://doi.org/10.1109/ICDS50568.2020.9268686.

[44] P. Cousot och R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", i *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77, Los Angeles, California: Association for Computing Machinery, 1977, s. 238–252. https://doi.org/10.1145/512950.512973.

[45] C. Jensen, C. Vestlund, C. Gustavsson, V. Andersson, L. Nyholm och D. Eidenskog, "Inventering av testfall för verktyg som identifierar mjukvarusårbarheter", Totalförsvarets forskningsinstitut, FOI Memo 8673, 2024.

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

[47] H. Karlzén, J. Falkcrona, D. Eidenskog och M. Karresand, “Mjukvarors säkerhet beror på utvecklarnas motivationer och hinder - En enkätundersökning”, Totalförsvarets forskningsinstitut, FOI-R--5691--SE, 2024.

[48] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski och A. Wierzbicki, “GitHub Projects. Quality Analysis of Open-Source Software”, 2014-11, s. 80–94. https://doi.org/10.1007/978-3-319-13734-6_6.

[49] G. Blom, J. Enger, G. Englund, J. Grandell och L. Holst, *Sannolikhetsteori och statistikteori med tillämpningar*. Studentlitteratur, 2017.

A Insamlad GitHub-data

Statistik insamlad från verktygens GitHub-projekt presenteras i tabell A.1.

Tabell A.1. Mätvärden insamlade från Github (2025-09-22), sorterade efter TRL och namn. Verktyg markerade med * inkluderas från den tidigare studien [2].

Verktyg	TRL	Forks	Stars	Commits	Contrib.	Releases	Branches
POLYFUZZ*	9	3	27	773	3	6	1
AFLSmart*	8	91	517	257	10	0	2
Angr*	8	1100	8200	13650	255	243	199
Goshawk*	8	15	99	59	4	0	4
Fuzzware*	8	62	350	77	11	2	2
Inception*	8	1	25	59	2	0	1
OSS-Sydr-Fuzz*	8	36	141	589	20	0	3
AFLGO	7	140	536	345	0	1	2
AFLNET	7	203	956	99	38	0	1
CRYSTALLIZER	7	1	14	22	0	0	1
Futag	7	11	52	190	4	17	2
JIGSAW	7	8	69	27	2	0	1
MIRCHECKER	7	27	154	11	4	0	1
MOVEC	7	1	22	6	0	2	1
SaTC	7	59	317	96	3	1	3
Sfuzz	7	11	90	9	4	0	1
AUSERA	6	3	32	21	2	0	1
FUNDED	6	36	128	161	5	0	1
Fuzztruction	6	23	132	11	0	0	1
QuickFuzz	6	45	198	570	16	2	4
SafeDrop	6	27	109	780	12	1	2
SySeVR	6	137	341	175	0	0	1
C11Tester	5	4	11	2088	3	0	1
Hoedur	5	10	66	12	2	1	1
VUzzer	5	108	384	9	3	0	1
Zest	5	120	709	856	22	16	18
PhASAR	-	149	1007	3226	48	4	52

B Ansats till rangordning av verktyg

Att ett verktyg uppfyller kriterierna för ett relevant utvecklingsarbete betyder endast att det finns en potential att verktyget stigit i mognadsgrad. Inför projektets kommande verktygstest vore det önskvärt att kunna rangordna dem inbördes med ett objektivt mått som kvantifierar deras potential.

I arbetet med studien har en ansats undersökts för rangordning av de sammanlagt 27 verktyg som identifierats i projektet. Metoden rangordnar verktygen utifrån hur de uppdaterats efter den initiala akademiska publiceringen. Det sker genom att ett mått beräknas för aktivitetsnivån i respektive GitHub-projekt.

Det visar sig vara svårt att fånga ett projekts utveckling och potential i ett numeriskt värde. Därför redovisas rangordningen enbart som en bilaga – en rapport i rapporten – och är enbart ett av flera kriterier för hur projektets framtida verktygstester prioriteras.

B.1 Metod

För rangordning används ett kvantitativt mått som baseras på publikt tillgänglig metrik från verktygens GitHub-projekt. Metoden som beskrivs är ett tillägg till arbetet som beskrivs i kapitel 2 och tillämpas på inkluderade verktyg.

Metoden tar inspiration från en studie av Jarczyk m. fl. [48]. Deras arbete handlar om att bedöma kvalitet på öppen källkod utifrån GitHub-projekts metriker. Författarna identifierade sex metriker som korrelerade positivt med ett projekts kvalitet: forks, stars, commits, contributors, releases och branches. Data för dessa metriker samlades in för samtliga verktyg som uppfyllde kriterierna för relevant utvecklingsarbete, samt de verktyg som identifierats genom den tidigare studien. Insamlade metriker redovisas i bilaga A.

Databearbetningen syftar till att beräkna ett kvantitativt mått för aktivitetsnivån i utvecklingen av de identifierade verktygen. För att insamlad mätdata skulle kunna användas som kvalitetsmått genomfördes en initial normaliseringsprocess som hanterade två problem med insamlad rådata:

- Kategorierna av metriker har olika storleksordningar. Antalet forks, stars och commits är normalt avsevärt större än antal medverkande utvecklare i ett projekt.
- Insamlad data innehåller extremvärden för enskilda metriker som omotiverat påverkar helhetsbilden av aktivitetsnivån hos ett projekt.

Initialt logaritmerades värdena för att minska extremvärdens inverkan på helhetsbilden. För att öka jämförbarhet mellan olika kategorier av metriker användes deras Z-Score istället för det insamlade värdet.[41]

Två metoder utforskades för att beräkna en jämförande rang för varje verktyg:

Medelvärde: Beräknas enbart utifrån data som hämtats från GitHub. Varje verktygs värde beräknas som medelvärdet av Z-Score för de sex metrikerna (forks, stars, commits, contributors, releases och branches).

Korrelationsviktat medelvärde: För respektive verktyg representerar det korrelationsviktade medelvärdet ett viktat genomsnitt av verktygets metriker, där metrikerna med starkast korrelation till verktygets mognadsskattning får störst genomslag. Det korrelationsviktade medelvärdet beräknas för varje verktyg enligt ekvation (B.1).

$$m_{w\_corr} = \sum \rho_i z_i \quad , \tag{B.1}$$

där z_i är Z-Score för metrik i och ρ_i är Spearman-korrelationen[42] mellan hela datasamlingen för metrik i och TRL-nivå från den tidigare studien [2].

[41] Z-Score (Z-värde eller standardiserad standardavvikelse) är ett sätt att beskriva hur många standardavvikelser ett värde är från medelvärdet i en fördelning [49, s. 119].

[42] Spearman-korrelation beräknas genom att ta korrelationen mellan två rangordnade datasamlingar [49, s. 235].

B.2 Resultat

Den slutliga rangordningen baseras på ett medelvärde mellan de två måtten, med lika vikt, och redovisas tillsammans med respektive verktygs uppskattade teknikmognadsnivå (TRL) i tabell B.1. Verktyg med samma medelvärde tilldelas delad placering.

Tabell B.1. Rangordning av verktyg utifrån hur de uppdaterats efter den initiala akademiska publiceringen, med tillhörande medelvärden. Verktyg från den tidigare studien är markerade med * [2].

Placering	Verktyg	TRL	Medelvärde	Korrelationsviktat medelvärde
1	Angr*	8	17.83	3.57
2	PhASAR	-	8.82	1.90
3	Zest	5	7.23	1.44
4	QuickFuzz	6	2.70	0.56
5	AFLNET	7	2.73	0.28
6	OSS-Sydr-Fuzz*	8	1.42	0.34
7	AFLSmart*	8	1.71	0.18
7	Fuzzware*	8	1.53	0.19
9	SafeDrop	6	1.14	-0.20
10	Futag	7	0.42	-0.04
10	SaTC	7	0.65	0.01
12	AFLGo	7	0.88	-0.15
13	POLYFUZZ*	9	-1.30	-0.09
14	FUNDED	6	-0.89	-0.26
15	SySeVR	6	-0.71	-0.48
16	Goshawk*	8	-1.39	-0.16
17	VUzzer	5	-1.19	-0.49
18	C11Tester	5	-2.78	-0.34
19	MIRCHECKER	7	-2.40	-0.55
20	Hoedur	5	-3.36	-0.64
20	Sfuzz	7	-3.40	-0.63
22	JIGSAW	7	-3.62	-0.66
23	Fuzztruction	6	-3.83	-0.88
23	AUSERA*	6	-4.77	-0.77
23	Inception*	8	-4.86	-0.71
26	MOVEC	7	-6.00	-1.00
27	CRYSTALLIZER	7	-6.56	-1.07

Resultaten visar att verktygen i både den övre och nedre delen av rangordningen uppvisar stabila positioner oavsett vilket medelvärdesmått som används. De största variationerna återfinns bland verktyg i mellanskiktet, där små skillnader i aktivitetsmönster påverkar placeringen mer påtagligt.

B.3 Diskussion

Rangordningen ger en indikation om vilka verktyg som kan vara bäst lämpade för praktiska tester, då den tar hänsyn till mognadsgrad och att verktyget bärs upp av en aktiv skara utvecklare. Det är sannolikt dock svårt att fånga ett verktygs utveckling och potential i ett numeriskt värde. Inför praktiska tester kan det fungera som ett av flera underlag för prioritering av verktygens testordning.

Det noteras en skillnad mellan rangordningen och den tidigare TRL-bedömningen, vilket framgår av tabell B.1. Verktygen högst i rangordningen har inte nödvändigtvis den högsta mognadsnivån (TRL 8–9). Ett tydligt exempel är Zest, som i den ursprungliga publikationen bedömdes ha en relativt låg mognadsgrad, men som nu rankas högt till följd av ett aktivt utvecklarcommunity som etablerats efter publiceringen.

Resultatet visar att metodansatsen tillför ett perspektiv. Samtidigt är resultatet svårtolkat och dess relevans svår att verifiera utan jämförande, praktiska, tester av verktygen. Rangordningen kan används som ett av flera underlag vid planering av kommande verktygstester.

Metoddiskussion

Den använda metodansatsen hämtar inspiration av Jarczyk m.fl. för indikatorer på kvalitet i GitHub-projekt [48]. Syftet med respektive arbete skiljer sig dock åt, liksom nivån på de mjukvaruprojekt som studeras. Jarczyk m.fl. studerar några av GitHubs mest populära mjukvaruprojekt, medan den här studien granskar nischade forskningsprojekt med begränsad användarinteraktion.

Den metod som tillämpas för rangordning har begränsningar. Att så många verktyg inte är tillgängliga, och därmed exkluderats i rapportens urvalsprocess, innebär att intressanta verktyg sannolikt saknas. Det stora bortfallet av verktyg innebär också ett stort bortfall av data som ligger till grund för beräkning av det korrelationsviktade medelvärdet. Om fullständiga data hade funnits tillgängliga för samtliga verktyg är det möjligt att rangordningen hade sett annorlunda ut.

Rangordningen ger uttryck för ett verktygs popularitet, men säger inte nödvändigtvis någonting om dess kvalitet. Det omvända kan också sägas vara sant.

Att ett verktyg placerat sig långt ned i rangordningen, eller inte är tillgängligt, innebär inte att det är av låg kvalitet.

Försöket med att rangordna mjukvaruprojekt är intressant och lutar sig delvis mot tidigare forskning. Samtidigt finns det goda skäl att förhålla sig skeptisk till resultaten. Att fånga ett GitHub-projekts utveckling och potential med ett entydigt, numeriskt, värde visar sig svårt.

ISSN 1650-1942