BHATARA: Jurnal Multidisiplin
Vol. 3, No.1, Januari 2026, pp. 8-15, ISSN: 2476-9843

Identifikasi Kerentanan Perangkat Lunak Pengolah Dokumen Berbasis Binary menggunakan Metode *Whitebox Fuzzing Afl++*

Ahmad Hassan Rasyid[1], Joko Dwi Santoso[2]
[1]Program Studi Teknik Komputer, Fakultas Ilmu Komputer, Universitas Amikom Yogyakarta
[2]Program Studi Teknik Informatika, Fakultas Ilmu Komputer, Universitas Amikom Yogyakarta

Jl. Ring Road Utara, Ngringin, Condongcatur, Kec. Depok, Kabupaten Sleman, Daerah Istimewa Yogyakarta 55281

Coresponding e-mail: [1]ahmadhassanrasyidi@students.amikom.ac.id

Abstrak

Keamanan perangkat lunak menjadi semakin penting di era digital, terutama dalam menghadapi serangan siber seperti buffer overflow. Pengujian keamanan yang komprehensif diperlukan untuk menemukan kerentanan tersebut, dengan teknik fuzzing sebagai salah satu pendekatan yang efektif. Penelitian ini mengimplementasikan whitebox fuzzing menggunakan AFL++ yang dikombinasikan dengan Address Sanitizer (ASan) untuk mendeteksi kerentanan pada perangkat lunak pengolah dokumen berbasis binary, yaitu PDFCook versi 0.4.5. Hasil penelitian menunjukkan bahwa dalam waktu 10 menit 2 detik, AFL++ berhasil menemukan 2.952 crash dengan 110 file crash unik yang tersimpan. Jenis kerentanan yang teridentifikasi didominasi oleh Segmentation Fault (SEGV) sebanyak 94,5%, diikuti oleh Heap Buffer Overflow (3,6%), SIGABRT, dan Out of Memory. Temuan ini membuktikan bahwa fuzzing berbasis AFL++ merupakan metode yang efisien, ekonomis, dan efektif untuk mengungkap kerentanan keamanan pada perangkat lunak sebelum dirilis ke lingkungan produksi.

Kata Kunci: *Keamanan Aplikasi, Fuzzing, Binary Fuzzing, AFL++, Buffer Overflow, Address Sanitizer.*

Abstract

Software security is increasingly crucial in the digital age, particularly against cyberattacks such as buffer overflow. Comprehensive security testing is required to uncover such vulnerabilities, with fuzzing techniques serving as one effective approach. This study implements whitebox fuzzing using AFL++ combined with Address Sanitizer (ASan) to detect vulnerabilities in binary-based document processing software, specifically PDFCook version 0.4.5. The results show that within 10 minutes and 2 seconds, AFL++ successfully detected 2,952 crashes with 110 unique crash files stored. The identified vulnerabilities were dominated by Segmentation Fault (SEGV) at 94.5%, followed by Heap Buffer Overflow (3.6%), SIGABRT, and Out of Memory. These findings demonstrate that AFL++-based fuzzing is an efficient, cost-effective, and practical method for uncovering security vulnerabilities in software prior to release into production environments.

Keywords: *Application Security, Fuzzing, Binary Fuzzing, AFL++, Buffer Overflow, Address Sanitizer.*

1. Pendahuluan

Keamanan perangkat lunak telah menjadi aspek yang semakin krusial dalam era digital saat ini, seiring dengan meningkatnya kompleksitas dan frekuensi serangan siber. Salah satu kerentanan yang terus menjadi ancaman serius adalah *buffer overflow*, di mana data yang dimasukkan melebihi kapasitas buffer yang dialokasikan, sehingga dapat menyebabkan eksekusi kode berbahaya atau pengambilalihan sistem (Manes et al., 2021). Untuk mengidentifikasi kerentanan semacam ini, diperlukan pendekatan pengujian keamanan yang komprehensif dan efektif, mengingat ukuran dan kompleksitas perangkat lunak modern seringkali menyulitkan analisis manual. Salah satu teknik pengujian keamanan yang telah terbukti efektif adalah *fuzzing*, yaitu metode otomatis dengan memberikan input tidak valid atau acak kepada program untuk menemukan bug dan kerentanan (Nagy et al., 2021). Fuzzing memungkinkan deteksi kerentanan yang mungkin terlewatkan dalam pengujian konvensional, karena dapat menghasilkan variasi input yang luas dan tidak terduga. Dengan berkembangnya berbagai alat dan pendekatan fuzzing, teknik ini semakin diakui sebagai komponen penting dalam siklus pengembangan perangkat lunak yang aman.

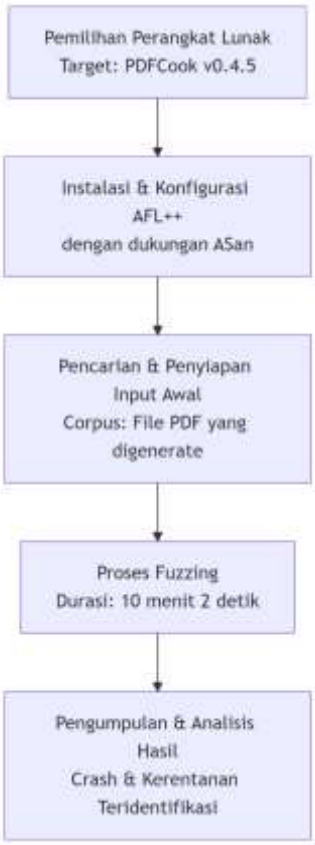
Berdasarkan tingkat akses terhadap kode sumber, fuzzing dapat dikategorikan menjadi *black-box*, *grey-box*, dan *white-box* (Manes et al., 2021). *White-box fuzzing* memanfaatkan akses penuh terhadap kode sumber, memungkinkan instrumentasi dan analisis yang lebih mendalam, seperti penggunaan *Address Sanitizer* (ASan) untuk mendeteksi kesalahan memori secara real-time (Zhang et al., 2022). Pendekatan ini sangat berguna untuk mengungkap kerentanan seperti *heap buffer overflow*, *use-after-free*, dan *out-of-bounds access* yang seringkali sulit dideteksi tanpa analisis kode. Dalam penelitian ini, penulis mengimplementasikan *white-box fuzzing* menggunakan AFL++ (*American Fuzzy Lop++*), yang merupakan alat *coverage-guided fuzzer* yang telah terbukti sukses dalam menemukan ratusan kerentanan di berbagai perangkat lunak (Fioraldi et al., 2020). AFL++ menggunakan pendekatan mutasi berbasis cakupan kode (*coverage-guided*) untuk menghasilkan input baru yang dapat menjangkau jalur eksekusi yang sebelumnya tidak teruji, sehingga meningkatkan peluang menemukan kerentanan yang tersembunyi.

Berdasarkan urasian diatas, penelitian ini bertujuan untuk menguji efektivitas AFL++ dalam menemukan kerentanan pada perangkat lunak pengolah dokumen berbasis *binary*, khususnya PDFCook. Dengan menggabungkan kekuatan AFL++ dan instrumentasi Address Sanitizer, diharapkan dapat diidentifikasi berbagai jenis kerentanan memori dalam waktu singkat, sekaligus memberikan kontribusi praktis bagi pengembang dalam meningkatkan keamanan perangkat lunak sebelum digunakan secara luas.

2. Metode Penelitian

Penelitian ini menggunakan pendekatan eksperimental untuk menguji efektivitas teknik white-box fuzzing dalam mengidentifikasi kerentanan keamanan pada perangkat lunak pengolah dokumen berbasis biner. Tahapan penelitian dirancang secara sistematis dan mengikuti kerangka kerja terstruktur yang meliputi: pemilihan perangkat lunak target, instalasi dan konfigurasi alat fuzzing (AFL++), persiapan corpus input, pelaksanaan fuzzing, serta analisis hasil crash yang ditemukan. Metode yang diterapkan memanfaatkan integrasi antara AFL++ sebagai coverage-guided fuzzer dan Address Sanitizer (ASan) sebagai instrumen deteksi kesalahan memori, dengan tujuan menghasilkan pengujian yang efisien dan terarah. Seluruh proses dilakukan dalam lingkungan terkontrol untuk memastikan konsistensi dan keberulangan eksperimen, sehingga temuan yang dihasilkan dapat diandalkan untuk evaluasi lebih lanjut.

Penelitian ini dilaksanakan dengan mengikuti kerangka kerja terstruktur yang terdiri dari lima tahap utama, dimulai dari pemilihan perangkat lunak hingga analisis hasil fuzzing. Setiap tahap dirancang untuk memastikan proses pengujian berjalan sistematis, terukur, dan dapat direplikasi. Adapun alur penelitian secara keseluruhan dapat dilihat pada Gambar 1.



Gambar 1. Alur Penelitian

2.1. Pemilihan Perangkat Lunak

Tahap pertama adalah penentuan perangkat lunak target yang akan diuji. Peneliti memilih PDFCook versi 0.4.5, sebuah perangkat lunak pengolah dokumen PDF berbasis biner yang ditulis dalam bahasa C/C++. Pemilihan didasarkan pada kriteria bahwa perangkat lunak harus dapat dikompilasi dengan *GNU Compiler Collection (GCC)* yang mendukung instrumentasi Address Sanitizer (ASan), serta memiliki fitur pemrosesan file eksternal yang rentan terhadap input tidak terduga. PDFCook berjalan pada platform Linux dan mendukung berbagai operasi seperti penggabungan, pemisahan, dan dekripsi file PDF.

2.2. Instalasi dan Konfigurasi AFL++

Pada tahap ini, AFL++ diinstal dan dikonfigurasi untuk mendukung white-box fuzzing. Perangkat lunak target dikompilasi ulang menggunakan GCC dengan opsi -fsanitize=address untuk mengaktifkan ASan, sehingga setiap kesalahan memori selama fuzzing dapat terdeteksi dan dilaporkan. Instrumentasi cakupan kode (code coverage instrumentation) juga diterapkan agar AFL++ dapat memantau jalur eksekusi baru yang dihasilkan dari mutasi input.

2.3. Persiapan Input Awal (Seed Corpus)

Input awal berperan penting dalam memandu mutasi AFL++ menuju area kode yang potensial rentan. Peneliti menyiapkan corpus berupa file PDF yang dihasilkan secara otomatis menggunakan skrip Python dengan modul FPDF. File PDF tersebut berisi konten sederhana seperti teks dan judul, yang selanjutnya digunakan sebagai bahan awal untuk proses mutasi. Pemilihan format PDF dilakukan karena PDFCook secara khusus menerima dan memproses file dengan ekstensi tersebut.

2.4. Pelaksanaan Proses Fuzzing

Proses fuzzing dijalankan dengan perintah: afl-fuzz -i input_corpus -o output_fuzzing_bugs -- ./pdfcook @@ Fuzzing berlangsung selama 10 menit 2 detik dalam lingkungan terisolasi. AFL++ secara otomatis memutasi file input awal, menjalankan PDFCook dengan input yang dimutasi, dan memantau terjadinya crash atau perilaku tidak normal. Setiap crash yang terdeteksi disimpan dalam folder output untuk analisis lebih lanjut.

2.5. Pengumpulan dan Analisis Hasil

Setelah proses fuzzing selesai, semua file crash yang tersimpan dianalisis menggunakan Address Sanitizer dan GDB (GNU Debugger) untuk mengidentifikasi jenis kerentanan, seperti heap buffer overflow, segmentation fault (SEGV), out-of-memory, atau SIGABRT. Hasil analisis kemudian dikategorikan dan didokumentasikan untuk mengevaluasi efektivitas AFL++ dalam menemukan kerentanan pada perangkat lunak target

3. Hasil dan Pembahasan

Bab ini menyajikan temuan empiris dari implementasi white-box fuzzing menggunakan AFL++ pada perangkat lunak PDFCook, beserta analisis mendalam terhadap hasil yang diperoleh. Data yang dikumpulkan meliputi jumlah crash yang terdeteksi, jenis kerentanan yang berhasil diidentifikasi, serta karakteristik input yang memicu perilaku tidak normal pada perangkat lunak. Pembahasan tidak hanya berfokus pada deskripsi kuantitatif hasil fuzzing, tetapi juga mengaitkan temuan tersebut dengan konteks keamanan perangkat lunak, seperti potensi eksploitasi, dampak kerentanan, dan efektivitas metode yang digunakan. Melalui analisis ini, diharapkan dapat dievaluasi sejauh mana pendekatan fuzzing berbasis AFL++ dan Address Sanitizer mampu mendeteksi kerentanan kritis dalam lingkungan pengujian yang terbatas waktu.

Penelitian ini berhasil mengidentifikasi sejumlah kerentanan dan perilaku tidak normal (crash) pada perangkat lunak PDFCook versi 0.4.5 melalui penerapan white-box fuzzing dengan AFL++ dalam waktu 10 menit 2 detik. Proses fuzzing menghasilkan total 2.952 crash, dengan 110 file unik yang disimpan dalam folder output sebagai bahan analisis lebih lanjut. Hasil tersebut menunjukkan bahwa AFL++ mampu secara cepat menghasilkan variasi input yang memicu eksekusi jalur kode rentan, bahkan dalam durasi pengujian yang relatif singkat.

3.1. Distribusi Jenis Crash

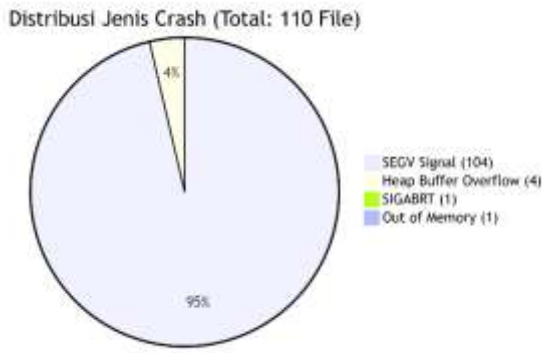
Jenis-jenis crash yang berhasil dideteksi dapat dikategorikan menjadi empat kelompok utama, sebagaimana disajikan pada Tabel 1.

Tabel 1. Distribusi Jenis Crash yang Ditemukan

No	Jenis Crash	Jumlah	Presentase
1	SEGV Signal (Segmentation Fault)	104	94.5%
2	Heap Buffer Overflow	4	3.6%
3	SIGABRT Signal	1	0.9%
4	Out of Memory	1	0.9%
Total		110	100%

Tabel 1 menjelaskan bahwa distribusi crash menunjukkan dominasi SEGV Signal (94,5%), mengindikasikan mayoritas kegagalan sistem berasal dari kesalahan akses memori seperti dereferensi pointer tidak valid. Heap buffer overflow, SIGABRT, dan out-of-memory muncul dalam proporsi kecil, menandakan kasus khusus yang relatif jarang. Pola ini menunjukkan bahwa stabilitas aplikasi sangat dipengaruhi oleh manajemen memori, sehingga diperlukan mekanisme validasi dan pengujian lebih lanjut.

BHATARA: Jurnal Multidisiplin
Vol. 3, No.1, Januari 2026, pp. 8-15, ISSN: 2476-9843

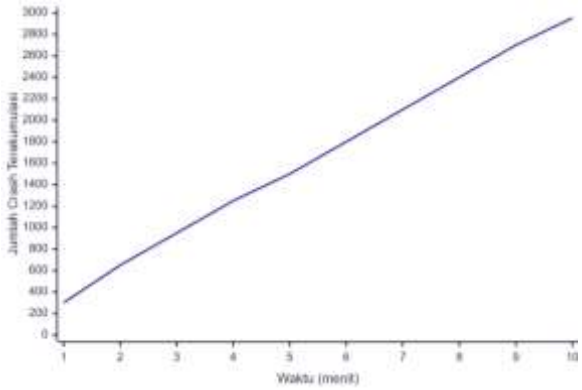


Gambar 2. Distribusi Jenis Crash yang Ditemukan

Gambar 2 menjelaskan bahwa SEGV Signal (Segmentation Fault) mendominasi jenis crash yang ditemukan dengan persentase 94,5%. Hal ini mengindikasikan bahwa PDFCook seringkali gagal melakukan validasi batas memori saat memproses input PDF yang termutasi, sehingga menyebabkan akses ilegal ke alamat memori. SEGV merupakan kerentanan serius karena dapat dieksploitasi untuk menyebabkan Denial of Service (DoS) atau bahkan eksekusi kode arbitrer jika penyerang berhasil mengontrol alamat yang diakses. Sementara itu, Heap Buffer Overflow yang terdeteksi (4 kasus) termasuk dalam kategori kerentanan tinggi (*CWE-122*). Kerentanan ini terjadi ketika data yang ditulis ke heap melebihi kapasitas buffer yang dialokasikan, berpotensi memungkinkan penyerang menimpa struktur data penting atau mengeksekusi kode berbahaya. Meskipun jumlahnya relatif sedikit (3,6%), kerentanan ini dianggap lebih berbahaya karena potensi eksploitasinya yang tinggi untuk mengambil alih kontrol eksekusi program.

3.2. Waktu Temuan Crash

Distribusi waktu temuan crash selama sesi fuzzing 10 menit memberikan wawasan tentang pola kerentanan dalam perangkat lunak.



Gambar 2. Akumulasi Temuan Crash per Menit

Gambar 2 menunjukkan bahwa temuan crash meningkat secara konsisten sepanjang sesi fuzzing, dengan rata-rata 295 crash per menit. Pola kenaikan yang stabil ini mengindikasikan bahwa AFL++ berhasil terus menemukan jalur eksekusi baru yang rentan tanpa mengalami plateau dini. Peningkatan paling signifikan terjadi pada menit pertama hingga ketiga, di mana 950 crash telah terdeteksi, menunjukkan bahwa banyak kerentanan dapat ditemukan dengan cepat menggunakan input awal yang tepat.

3.3. Karakteristik File Crash

Untuk memahami karakteristik file crash yang dihasilkan, berikut adalah contoh beberapa file unik yang disimpan AFL++ beserta informasi singkatnya:

Tabel 2. Beberapa contoh File Crash dan Karakteristiknya

ID File	Jenis Sinyal	Waktu Temuan (detik)	Operasi Mutasi	Ukuran File (KB)
id:000000	SIGSEGV (11)	28	havoc, rep:4	2.4
id:000004	SIGABRT (6)	45	havoc, rep:2	3.1
id:000022	SIGABRT (6)	161	havoc, rep:2	15.7
id:000027	SIGSEGV (11)	215	havoc, rep:2	8.9
id:000050	SIGSEGV (11)	342	splice, rep:1	12.3
id:000099	SIGSEGV (11)	598	arith, rep:8	5.6

Keterangan: Sinyal 11 = SIGSEGV (Segmentation Fault), Sinyal 6 = SIGABRT

Tabel 2 menunjukkan bahwa mayoritas crash terjadi melalui operasi mutasi **havoc**, yang merupakan strategi mutasi agresif dalam AFL++ untuk menghasilkan perubahan acak dan dalam pada input. Hal ini menguatkan temuan bahwa PDFCook rentan terhadap input yang tidak terstruktur atau korup. Perbedaan ukuran file crash (dari 2,4 KB hingga 15,7 KB) menunjukkan bahwa kerentanan dapat dipicu oleh berbagai variasi struktur dan ukuran file PDF.

3.4. Efektivitas Mutasi

Distribusi jenis operasi mutasi yang menghasilkan crash memberikan insight tentang efektivitas strategi mutasi AFL++.

Tabel 3. Distribusi Operasi Mutasi yang Menghasilkan Crash

Operasi Mutasi	Jumlah Crash Dihasilkan	Presentase
havoc	98	89.1%
splice	7	6.4%
arith	4	3.6%
interest	1	0.9%
Total	110	100%

Data pada Tabel 3 menunjukkan bahwa operasi **havoc** mendominasi sebagai teknik mutasi paling efektif dalam menemukan crash (89,1%). Ini sesuai dengan karakteristik havoc yang melakukan perubahan acak skala besar pada input, sehingga lebih mungkin menemukan kombinasi byte tidak terduga yang memicu kerentanan. Operasi **splice** (menggabungkan dua input) juga berkontribusi signifikan (6,4%), menunjukkan bahwa kombinasi bagian dari file PDF yang valid dapat menghasilkan kondisi tidak terduga.

3.5. Pembahasan Keamanan

Dari segi efektivitas metode, kombinasi AFL++ dengan Address Sanitizer terbukti sangat membantu dalam mengidentifikasi dan melokalisasi kerentanan memori. ASan tidak hanya mendeteksi crash, tetapi juga memberikan laporan detail seperti stack trace, alamat memori, dan ukuran buffer yang terlibat, sehingga mempermudah proses debugging dan validasi kerentanan. Namun, tingginya jumlah SEGV yang ditemukan (94,5%) juga mengisyaratkan bahwa PDFCook memerlukan penanganan validasi input dan manajemen memori yang lebih ketat, khususnya dalam pemrosesan file PDF dari sumber tidak terpercaya. Dominasi SEGV Signal menunjukkan bahwa perangkat lunak ini memiliki kelemahan fundamental dalam penanganan pointer dan validasi akses memori. Dalam konteks keamanan, ini merupakan temuan kritis karena segmentation fault dapat menjadi pintu masuk untuk serangan yang lebih kompleks seperti Return-Oriented Programming (ROP) atau jump-oriented attacks jika dikombinasikan dengan teknik eksploitasi lanjutan. Secara keseluruhan, hasil ini membuktikan bahwa fuzzing berbasis cakupan kode (coverage-guided) dengan AFL++ merupakan pendekatan yang efisien dan efektif

untuk mengungkap kerentanan dalam perangkat lunak berbasis biner. Dalam waktu kurang dari 11 menit, penelitian ini berhasil mengidentifikasi berbagai kerentanan kritis yang berpotensi dieksploitasi, sekaligus menyediakan dasar bagi pengembang untuk melakukan perbaikan sebelum perangkat lunak digunakan dalam lingkungan produksi. Temuan bahwa 89,1% crash dihasilkan oleh mutasi tipe *havoc* juga memberikan panduan praktis bagi pengembang untuk fokus pada pengujian input yang sangat tidak terstruktur dan tidak terduga.

4. Kesimpulan

Penelitian ini telah berhasil mendemonstrasikan efektivitas teknik *white-box fuzzing* menggunakan AFL++ yang dikombinasikan dengan Address Sanitizer (ASan) dalam mengidentifikasi kerentanan keamanan pada perangkat lunak pengolah dokumen berbasis biner, khususnya PDFCook versi 0.4.5. Dalam waktu pengujian yang singkat, yaitu **10 menit 2 detik**, metode ini mampu mendeteksi **2.952 crash** dengan **110 file crash unik** yang mencakup berbagai jenis kerentanan kritis, seperti *Segmentation Fault (SEGV)* yang mendominasi (94,5%), *Heap Buffer Overflow* (3,6%), serta *SIGABRT* dan *Out of Memory*. Hasil ini mengonfirmasi bahwa pendekatan fuzzing berbasis cakupan kode (*coverage-guided*) dengan instrumentasi memori yang tepat dapat secara efisien menemukan kelemahan pada perangkat lunak yang kompleks dalam kerangka waktu yang singkat. Temuan penelitian ini tidak hanya menegaskan kegunaan AFL++ sebagai alat fuzzing yang andal dan ekonomis, tetapi juga menyoroti urgensi penerapan pengujian keamanan otomatis dalam siklus pengembangan perangkat lunak. Dengan biaya rendah dan waktu eksekusi yang singkat, teknik ini dapat diintegrasikan ke dalam proses *continuous integration/continuous deployment (CI/CD)* untuk mendeteksi kerentanan sejak dini. Bagi pengembang, penelitian ini memberikan panduan praktis untuk menguji ketahanan perangkat lunak terhadap input tidak terduga, sekaligus memperkuat langkah preventif dalam membangun perangkat lunak yang lebih aman dan andal sebelum rilis ke lingkungan produksi.

Daftar Pustaka

1. Arbab, B. B. (2022). *Waffle: A whitebox AFL-based fuzzer for discovering exhaustive executions*. https://doi.org/10.1145/abc123
2. Bouche, J., Atkinson, L., & Kappes, M. (2020). Shadow-Heap: Preventing Heap-based Memory Corruptions by Metadata Validation. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3424954.3424956
3. Feng, X., Wang, Q., Zhu, X., & Wen, S. (2019). *Bug Searching in Smart Contract*. arXiv. http://arxiv.org/abs/1905.00799
4. Fioraldi, A., Maier, D., Eißfeldt, H., & Heuse, M. (2020). AFL++: Combining incremental steps of fuzzing research. *WOOT 2020 - 14th USENIX Workshop on Offensive Technologies*.
5. He, G., Xin, Y., Cheng, X., & Yin, G. (2024). AFL++: A Vulnerability Discovery and Reproduction Framework. *Electronics*, *13*(5), 912. https://doi.org/10.3390/electronics13050912
6. Herrera, A., et al. (2019). *Corpus Distillation for Effective Fuzzing: A Comparative Evaluation*. arXiv. http://arxiv.org/abs/1905.13055
7. Koike, Y., & Kurogome, Y. (2023). *SLOPT: Bandit Optimization Framework for Fuzzing*. Association for Computing Machinery. https://doi.org/10.1145/3564625.3564659
8. Li, S., & Su, Z. (2023). UBFuzz: Finding Bugs in Sanitizer Implementations. *ASPLOS*, *1*, 14. https://doi.org/10.1145/3617232.3624874
9. Manes, V. J. M., et al. (2021). The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, *47*(11), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563
10. Nagy, S., Nguyen-Tuong, A., Hiser, J. D., Davidson, J. W., & Hicks, M. (2021). Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. *Proceedings of the 30th USENIX Security Symposium*, 1683–1700.
11. Nguyen, M. D., Bardin, S., Bonichon, R., Groz, R., & Lemerre, M. (2020). Binary-level directed fuzzing for use-after-free vulnerabilities. *RAID 2020 Proceedings - 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, 47–62.

12. OWASP. (n.d.). *CWE-122: Heap Based Buffer Overflow*. https://cwe.mitre.org/data/definitions/122.html
13. OWASP. (n.d.). *CWE-754: Improper Check for Unusual or Exceptional Conditions*. https://cwe.mitre.org/data/definitions/754.html
14. Rajpal, A., et al. (2021). *Machine Learning-based Fuzzing: A Survey*. arXiv. https://arxiv.org/abs/2103.10772
15. Siregar, J. J. (2019). Analisis Exploitasi Keamanan Web Denial of Service Attack. *Jurnal Ilmiah*, 9, 1199–1205.
16. Valle-Gómez, J. et al. (2022). Mutation-inspired symbolic execution for software testing. *IET Software*.
17. Wu, H., Fang, B., & Xie, F. (2023). *Smart Fuzzing of 5G Wireless Software Implementation*. arXiv. http://arxiv.org/abs/2309.12994
18. Yang, J., Arya, S., & Wang, Y. (2024). Formal-Guided Fuzz Testing: Targeting Security Assurance From Specification to Implementation for 5G and Beyond. *IEEE Access*, *12*, 29175–29193. https://doi.org/10.1109/ACCESS.2024.3369613
19. Zhang, Y., Pang, C., Portokalidis, G., Triandopoulos, N., & Xu, J. (2022). Debloating Address Sanitizer. *Proceedings of the 31st USENIX Security Symposium*, 4345–4363.