Fuzzing des Extensions Noyau de macOS

Erwan Fasquel[‡], Frédéric Tronel[‡], Yaëlle Vinçont[†]
[‡]CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
[†]Univ Rennes, Inria, CNRS, IRISA

***Abstract*—Le fuzzing est une technique efficace pour identifier des vulnérabilités dans des logiciels complexes, comme les pilotes de systèmes d'exploitation. En particulier, le fuzzing des extensions du noyau de macOS (*Kernel Extensions*, ou *kext*) pose des défis uniques, liés aux nouveaux processeurs ARM d'Apple, aux mesures de sécurité matérielle comme l'authentification de pointeur (PA) et à l'impossibilité d'accéder au code source pour la plupart des *kext*. Cet article examine les principaux obstacles liés au fuzzing des *kext*, tels que l'identification et la création d'une grammaire pour fuzzer les *kext*, l'instrumentation pour obtenir la couverture du code et détecter les bugs, et les limitations inhérentes à l'architecture ARM. En outre, une analyse approfondie des fuzzers existants a permis de mettre en avant leurs avancées et limites. Pour conclure, nous proposons des pistes d'amélioration pour le fuzzing des *kext* de macOS, telles que la coopération de plusieurs fuzzers, l'optimisation de la détection des bugs et l'évaluation des fuzzers sous différents angles.**

***Index Terms*—Kernel Extensions, Fuzzing, Recherche de Vulnérabilités, macOS, Sources Fermées**

I. Introduction

Les systèmes d'exploitation (*OS*) sont des composants critiques pour la sécurité des infrastructures numériques et leurs vulnérabilités ont immédiatement un impact élevé [1]. Ils sont complexes, difficile à maintenir sans bugs et à tester de manière approfondie, tout en fonctionnant avec des privilèges très élevés.

Dans un système d'exploitation, le noyau est responsable de la gestion des ressources, de la communication avec le matériel, de la sécurité et de l'isolation des processus. Il sert d'interface pour les applications issues de l'espace utilisateur qui sollicitent des services, comme l'accès à la mémoire ou à une partie du matériel. La communication entre les applications et le noyau se fait à l'aide des *appels système* soit de manière directe, soit le plus souvent par l'intermédiaire de bibliothèques système. Les pilotes (ou *drivers*) étendent les fonctionnalités du noyau en permettant l'accès à des périphériques matériels spécifiques. En général les pilotes s'exécutent au même niveau de privilège que le noyau. Les pilotes sont d'importantes sources de vulnérabilités, comme l'a démontré l'équipe iDEA dans leur article [2]. Ils ont identifié 74 CVE liées aux pilotes (appelés *Kernel Extensions*, ou *kext* dans macOS) dans les mises à jour de sécurité d'Apple [3], sur les 231 convernant les noyaux d'iOS 8 à iOS 13.4.1

Compte tenu du statut critique des pilotes du noyau, il est impératif de découvrir d'éventuelles vulnérabilités avant de procéder à la publication d'une nouvelle version d'un OS. C'est d'autant plus important pour les OS à source fermée, pour lesquels il est plus compliqué d'identifier des bugs. C'est par exemple le cas de macOS qui n'a rendu accessible qu'une partie de son code source et presque aucune information concernant les *kext*.

Le *fuzz testing*, ou *fuzzing*, est une approche innovante qui permet d'identifier des vulnérabilités dans différents systèmes. Elle consiste à fournir de manière répétée des entrées aléatoires, appelées cas de test, à une cible pour tenter de déclencher un bug. Les fuzzers sont généralement composés d'au moins un module dédié à la génération des cas de test et d'un autre à leur exécution [4]. Une analyse approfondie ou de l'instrumentation de la cible peuvent être effectuées en plus pour obtenir des connaissances supplémentaires au cours de l'exécution des cas de test, telles que la surface de code découverte au cours du fuzzing. Le module de génération oriente alors le fuzzing à l'aide de ces informations vers des directions spécifiques. Le module d'exécution s'en sert plutôt pour détecter les bugs. Le fuzzing s'applique également aux OS, bien que leurs spécificités présentent des défis particuliers.

Nous nous concentrerons sur le fuzzing des extensions du noyau de macOS et aborderons les contributions suivantes :

- Les défis du fuzzing des extensions du noyau de macOS et les techniques pour les traiter dans les outils récents ;
- Une discussion sur des pistes d'amélioration du fuzzing des *kext* de macOS: la collaboration de plusieurs fuzzers, la détection intelligente de bugs, l'évaluation du coût de l'instrumentation en termes de performances, le test des fuzzers sur les dernières versions de macOS et l'évaluation d'un fuzzer sur des bugs connus.

II. Contexte

A. Fuzzing

Le fuzzing est utilisé dans le domaine de la sécurité depuis plus de 30 ans [5]. Il s'agit d'une méthode permettant de tester la robustesse d'un logiciel en fournissant des entrées aléatoires ou malformées aux programmes. Le programme utilisé générait des flux aléatoires de caractères et les transmettait à des utilitaires de ligne de commande UNIX, découvrant ainsi de nombreux bugs. La technique a été affinée au fil des ans et les fuzzers sont maintenant généralement classés en fonction de la manière dont les cas de test sont générés.

1) Les fuzzeurs à génération: Ils créent de nouveaux cas de test à partir d'une grammaire ou d'une heuristique qui spécifie le format des cas de test. L'efficacité de ces fuzzer est limitée par la capacité à exprimer précisément le format attendu pour générer des cas de test valides pour la cible [4]. C'est pourquoi certains fuzzers tentent d'automatiser la

génération de la grammaire. IMF [6], par exemple, déduit une grammaire d'appel d'API pour macOS en se basant sur les traces d'exécution de programmes réels. Radamsa [7] est un fuzzer capable de fuzzer tout type de cible. Pour cela, il génère les cas de test à partir d'un simple fichier contenant des entrées valides de la cible qu'il va modifier.

2) Les fuzzers à mutation: Ils créent les cas de test en mutant d'autres cas de test, soit des graines initiales fournies en entrée, soit des cas de test obtenus par des mutations précédentes. Il est possible d'utiliser une grammaire pour filtrer les mutations générées afin de préserver la structure des données de la cible. Plusieurs stratégies sont employées dans la littérature, telles que la suppression, l'épissage, l'insertion d'appel système ou l'écrasement/mutation des arguments de certains appels système [8]. AFLFast [9], une optimisation d'AFL [10], propose de moins sélectionner les mutations des cas de test menant à des chemins souvent parcourus.

3) Les fuzzers évolutifs: Les fuzzers évolutifs se basent sur les fuzzers à mutation ou à génération. Ils s'appuyent sur des informations obtenues lors de l'exécution pour sélectionner les cas de test à muter ou la technique de génération à prioriser. Par exemple, LibFuzzer [11] s'appuie sur AddressSanitizer [12] pour obtenir la couverture des blocs de code et privilégier les mutations atteignant de nouveaux blocs.

B. Apple Silicon

En 2020, Apple a annoncé utiliser dorénavant ses propres processeurs, conçues sur la base de processeurs ARM, et qui se distinguent par leurs performances et leur efficacité énergétique [13]. En effet, les processeurs ARM ont un coût réduit, une conception simple, une faible consommation d'énergie et un faible dégagement de chaleur, ce qui les rend particulièrement adaptés aux appareils légers et portables tels que les smartphones et les ordinateurs portables [14].

Ces processeurs introduisent de nouvelles mesures de sécurité, telles que l'authentification des pointeurs (PA), qui implique l'ajout d'une signature cryptographique aux bits de poids fort inutilisés d'un pointeur avant son stockage en mémoire. Cette signature est ensuite vérifiée et supprimée lors de la lecture du pointeur dans la mémoire [15]. Ainsi, le compilateur ajoute les instructions de PA pour protéger les pointeurs, sans pour autant affecter l'exécution du programme.

C. Les extensions du noyau de macOS

Les extensions du noyau (*kext*) sont des modules chargés dynamiquement lors du lancement de macOS qui étendent les fonctionnalités du noyau XNU de macOS. Les *kext* fournissent des fonctionnalités système de bas niveau, telles que la prise en charge des pilotes de périphériques ou l'intégration du système de fichiers. Il est à noter que les *kext* s'exécutent avec le même niveau de privilèges que le noyau XNU, et ont donc un accès direct à la mémoire du système et au matériel. Cependant, Apple a entrepris une transition progressive des *kext* vers les *Extensions Systèmes*, afin de réduire leurs privilèges et d'éviter les crashs causés par des erreurs dans les *kext*.

Les *kext* communiquent de plusieurs façons avec de multiples composants. En premier lieu, à l'instar du noyau XNU, une application de l'espace utilisateur interagit avec les *kext* par le biais d'appels système de la bibliothèque I/O Kit. Les appels système envoyés aux *kext* sont effectués principalement avec la méthode générique `IOConnectCallMethod()`, qui prend comme argument n'importe quelle structure de données complexe [16]. Il est à souligner que ces arguments sont souvent non documentés. En deuxième lieu, un *kext* peut envoyer des *entitlements* au noyau XNU afin de vérifier si un programme exécuté en espace utilisateur peut invoquer le code privilégié du *kext*. Cette vérification a pour effet d'empêcher certains utilisateurs d'accéder aux *kext*. En outre, les *kext* reçoivent passivement les événements du système, tels que l'alimentation, les interruptions ou la synchronisation [17], et adaptent leur comportement en conséquence.

III. Les défis du fuzzing des extensions du noyau de macOS et outils existants

Le fuzzing d'OS à source fermée est étroitement lié au fuzzing de binaires. Pour créer une grammaire décrivant des appels valides au *kext*, il faut identifier quels sont les moyens de communications, malgré le manque de documentation. De plus, cette documentation limitée complique l'extraction de la structure du code et de la logique nécessaire pour diriger le fuzzer vers les fonctions critiques, et rend l'instrumentation plus difficile, tant pour obtenir la couverture de code que pour détecter des bugs. Enfin, les nouvelles processeurs ARM d'Apple, sur lesquelles est exécuté macOS, ajoutent des défis aux fuzzers, notamment de par le jeu d'instructions utilisé qui est de taille fixe et le manque d'outils d'analyse matures.

A. Identifier les communications et créer une grammaire

Pour fuzzer des *kext*, deux défis majeurs se présentent pour identifier les communications et obtenir une grammaire. Tout d'abord, il y a le manque de documentation. Ensuite, l'accès au code "profond", protégé par des privilèges ou seulement accessible dans certains états du système.

Plusieurs fuzzers ont tenté de construire une grammaire des appels système sur macOS. C'est le cas notamment des fuzzers tels que IMF [6] et Syzgen [16] pour les versions Intel de macOS, et KextFuzz [17] sur ARM.

IMF propose de créer une grammaire qui servira à générer une séquence d'appels système valide et qui atteindra un code profond. Le fuzzer déduit les dépendances explicites entre les appels systèmes en traçant l'exécution de programmes réels. Cette démarche implique la création d'une liste de fonctions d'API, qui est ensuite utilisée pour comparer les traces obtenues. Cela permet de déduire l'ordre des appels système et leurs dépendances. Cependant, cette approche est limitée car elle repose sur l'exécution manuelle de programmes réels, qui ne déclenchent pas tous les appels système. De plus, elle nécessite un travail d'analyse complexe pour annoter les fonctions de l'API avec une description des paramètres d'entrée et de sortie. Par ailleurs, les appels système déduits par IMF ne concernent que ceux relatifs au noyau et non

aux *kext*. Cette méthode ne peut donc pas être directement appliquée aux *kext*, qui utilisent un appel système avec des paramètres génériques [16].

Pour répondre à ces limitations, **SyzGen** améliore l'idée proposée par IMF en l'adaptant aux *kext*. En effet, SyzGen trace les séquences d'appels système vers les *kext*, mais traite le `void*` générique en supposant que les dépendances entre les arguments des appels systèmes peuvent exister n'importe où dans l'objet. SyzGen tente de coupler les octets identiques dans les entrées et sorties des différentes interfaces, puis trace les API de macOS utilisées afin de reconstruire les pointeurs contenant les données envoyées aux *kext*. La grammaire, générée pour s'interfacer sur Syzkaller [8], est améliorée en appliquant une passe d'exécution symbolique dynamique aux *kext*, jusqu'aux contrôles d'intégrité des paramètres fournis dans les appels système. De plus, SyzGen identifie les services exposés et les clients accessibles depuis l'espace utilisateur à l'aide d'une autre passe d'exécution symbolique dynamique.

KextFuzz propose une approche différente pour créer une grammaire similaire. Il utilise l'analyse de teinte au niveau des *wrappers* de l'espace utilisateur qui effectuent des appels systèmes aux *kext*. Ces *wrappers* sont des couches abstraites pour les services du noyau dans l'espace utilisateur, telles que des frameworks, des bibliothèques ou des démons.

KextFuzz ne se contente pas d'inférer la grammaire des appels de service; il parvient également à accéder au code privilégié. Pour cela, KextFuzz réécrit le symbole de la fonction de vérification des *entitlements* [18] et le remplace par celui d'une fonction située dans un *kext* développé par leurs soins. Ainsi, chaque *entitlement* est directement validé sans être envoyé au noyau XNU. Pour cela, KextFuzz réécrit le symbole de la fonction de vérification des *entitlements* [18] et le remplace par celui d'une fonction située dans un *kext* développé par leurs soins. Ainsi, chaque *entitlement* est directement validée sans être envoyé au noyau XNU.

KextFuzz gère les événements d'alimentation (`sleep`, `turn-off`, etc.) en interfaçant le fuzzer (Syzkaller), en espace utilisateur, avec un *kext* intermédiaire connecté au gestionnaire d'événements. Cependant, il est important de noter que KextFuzz ne gère pas d'autres types d'événements comme les interruptions ou les évènements temporels.

B. Instrumenter les kext

L'instrumentation consiste à modifier la cible pour récupérer de l'information au cours de son exécution. Cela permet notamment de collecter la couverture de code pendant l'exécution des cas de test, mais aussi de détecter les bugs.

1) Collecte de la couverture du code: Dans le cadre du fuzzing des *kext*, il existe de telles solutions pour les processeurs Intel [16] et ARM [17], [19], [20].

Pour les processeurs Intel, SyzGen s'appuie sur un débogueur de noyau pour collecter la couverture de code que Syzkaller utilise pour générer les cas de test [16]. D'autres solutions ont également été utilisées pour fuzzer Linux x86-64. Par exemple, kAFL [21] trace l'exécution des appels systèmes dans le noyau avec Intel Processor-Trace (Intel-PT).

Toutefois, il convient de noter que ces solutions ne peuvent pas être directement transférées aux nouvelles processeurs d'Apple, dont les capacités de virtualisation et de débogage du noyau sont limitées. Par conséquent, des solutions alternatives ont été identifiées. Tout d'abord, KextFuzz [17] exploite l'authentification des pointeurs (PA) dans macOS qui consite à ajouter des instructions pour par exemple s'assurer que l'addresse de retour de chaque fonction n'a pas été modifiée. Pour cela, KextFuzz remplace certaines instructions PA par un appel à une fonction qui enregistre l'adresse du bloc de base instrumenté, puis retourne au programme d'origine. Les instructions PA ne permettent toutefois que la collecte partielle des blocs de base [19].

Pour y remédier, Pishi [19] propose de remplacer la première instruction non relative au Compteur Ordinal (*Program Pointer*) par un saut vers un trampoline. Des trampolines sont placés dans un *kext* tierce qui enregistre l'adresse du bloc de base instrumenté avant d'exécuter l'instruction remplacée et de revenir au code cible. Autrement, le projet Google Zero [20] suggère de transférer le code des *kext* évalués dans l'espace utilisateur à l'aide d'IDA Pro, puis, d'utiliser un débogueur pour collecter la couverture. Toutefois, comme de nombreuses fonctions ne peuvent pas être exécutées en dehors de l'espace du noyau, la création de harnais pour ces fonctions est nécessaire et demande beaucoup de travail manuel.

2) Détecter les bugs: Les fuzzers récents ciblant macOS détectent uniquement les crashs [6], [16], [17], [19], [20]. C'est pertinent et simple à implémenter, car le code exécuté en espace noyau crash lorsqu'une erreur survient. Cependant, cette solution ne permet pas la détection de bugs plus complexes, comme les fuites de mémoire qui ne déclenchent pas nécessairement d'erreur immédiatement. Par ailleurs, Apple procède actuellement à la transition des *kext* vers les Extensions Système afin de réduire les privilèges de certains *kext* pour éviter les paniques du noyau [22]. Cette sécurité supplémentaire devrait empêcher les plantages du noyau lors d'une erreur dans un *kext*, ce qui rend nécessaire d'avoir de nouveaux mécanismes de détection des bugs.

Pour fuzzer Linux, des outils tels que Thunderkaller [23] s'appuient sur Kernel AddressSanitizer (KASAN) [24] pour détecter divers types de bugs liés à la mémoire. Sur macOS, KASAN est disponible pour quelques *kext* sur les processeurs Intel [16], mais il n'a pas été utilisé avec les récents fuzzers sur ARM. Néanmoins, ArmWrestling [25] a développé une version ARM d'AddressSanitizer. Elle est basée sur la *symbolisation* [26] qui consiste à transformer les références constantes aux tables de sauts et pointeurs en étiquette assembleur qui permettent de se libérer des contraintes liées aux calculs des décalages relatifs et ainsi pouvoir insérer des instructions au besoin. À notre connaissance, cette méthode ne s'applique que sur des binaires en espace utilisateur et ne couvre pas les exceptions en C++, mais elle pourrait permettre de créer une version d'AddressSanitizer et détecter des bugs de mémoire. Une autre idée serait d'appliquer la méthode décrite dans Pishi [19] pour collecter la couverture de code comme décrit précédemment et d'ajouter un saut avant les fonctions allouant

et désallouant la mémoire, à l'instar de KASAN.

C. Analyser des binaires sur Apple Silicon

Les plus récentes versions de macOS fonctionnent sur les nouvelles processeurs d'Apple, ce qui pose des problèmes pour l'analyse en raison de la taille fixe du jeu d'instructions et du manque de maturité des outils.

Les processeurs sur ARM, contrairement à x86-64, utilisent un jeu d'instructions où chaque instruction a une taille de 4 octets. L'inconvénient majeur de cette conception est qu'une seule instruction ne peut pas contenir un pointeur, qui nécessite 8 octets. Par conséquent, les pointeurs doivent être construits, ce qui complique leur récupération par les analystes lorsqu'ils tentent d'analyser un binaire. En outre, les tables de sauts sont stockées comme des listes de décalages par rapport à une base, qui peuvent être comprimées pour optimiser l'espace utilisé. Cette optimisation rend difficile l'identification des tables de sauts et l'évaluation de leurs emplacements [25]. Cependant, ArmWrestling [25] a relevé ces défis et peut instrumenter les binaires ARM avec une symbolisation analogue à celle de RetroWrite [26]. Cependant, les techniques décrites dans e9Patch [27], comme l'*instruction punning* ou l'*instruction padding*, ne peuvent pas être utilisées pour ARM. En effet, elles exploitent les différentes tailles du jeu d'instructions x86 pour modifier les binaires, ce qui est impossible sur ARM.

Peu d'outils d'analyse fonctionnent correctement sur ARM, manquant encore de maturité. Les versions récentes de macOS fonctionnant sur Apple Silicon manquent d'outils de virtualisation et de débogage du noyau [17]. En outre, les auteurs d'ArmWrestling ont découvert des bugs dans le désassembleur Capstone [28], et le débogueur GDB. Néanmoins, le fuzzer développé par le Google Zero Project [20] a tiré parti d'un débogueur pour l'espace utilisateur, TinyInst, qui pourrait être utile pour l'analyse dynamique des *kext*. Ses auteurs ont également utilisé IDA Pro pour le désassemblage, tandis que Pishi s'est appuyé sur Ghidra [19].

IV. Travaux futurs

Les extensions du noyau de macOS n'ont pas été analysées de manière aussi approfondie dans la littérature que les pilotes Linux. Le fuzzing d'un OS sans accès au code source constitue un défi majeur, avec notamment moins d'outils d'analyse. Par conséquent, il reste encore beaucoup à faire pour trouver des vulnérabilités sur les *kext* à l'aide du fuzzing.

a) Utiliser plusieurs fuzzers ensemble: L'idée principale d'EnFuzz [29] est d'utiliser plusieurs fuzzers différents en même temps, sur les mêmes binaires. Contrairement au mode parallèle d'AFL, EnFuzz emploie des fuzzers distincts et synchronise les cas de test intéressants entre eux. Cette approche permet de tirer parti des atouts de chaque fuzzeur.

Cette idée pourrait être adaptée au fuzzing de *kext*, en faisant travailler plusieurs fuzzers en parallèle sur des versions émulées de macOS et en partageant les cas de test. Pour l'émulation, nous pourrions nous baser sur LibAFL-QEMU [30] par exemple qui est capable d'émuler des cibles sur ARM et qui a été utilisé pour fuzzer le noyau de Windows.

b) Détection intelligente: Comme dans l'étude présentée dans [4], nous voudrions détecter des bugs spécifiques, plutôt que les crashes qui se produisent parfois après un bug. La détection des fuites de mémoire ou des débordements de tampon serait utile pour fuzzer les *kext*. Bien que les fuzzers actuels aient trouvé un certain nombre de vulnérabilités, il en reste probablement qui n'ont pas été détectées faute de crash immédiat.

c) Évaluer la surcharge introduite par l'instrumentation: À notre connaissance, les techniques les plus prometteuses pour instrumenter les *kext* sont celles développées par Pishi [19] et ArmWrestling [25]. Cependant, le coût de cette instrumentation n'a pas été évalué. Il pourrait donc être intéressant de le faire et d'étudier la couverture de code récupérée par les deux outils sur plusieurs *kext*, afin de décider quelle technique choisir pour instrumenter un *kext*.

d) Exécuter les fuzzers existants sur des versions récentes de macOS: Les fuzzers de *kext* ont été développés pour cibler des versions et des architectures macOS spécifiques. Il serait donc intéressant de tester s'ils sont toujours utilisables sur les dernières versions de macOS et s'en servir comme base de comparaison d'un futur fuzzer.

e) Déclencher des bugs connus: Une autre piste d'évaluation est de trouver des vulnérabilités existantes affectant les *kext* de la version de macOS ciblée, et d'essayer de les détecter par le fuzzing. Cette méthode permettrait de s'assurer que le fuzzer peut trouver des bugs réels. De plus, une suite de *kext* avec des bugs synthétiques serait utile pour évaluer un fuzzer en se basant sur le nombre de bugs synthétiques trouvés. Néanmoins, il faudrait ajouter à cette suite synthétique des bugs réels pour s'assurer que le fuzzer n'est pas entraîné à atteindre uniquement les bugs d'une suite de tests synthétiques et qu'il détecte aussi des bugs réels [31].

V. Conclusion

Fuzzer les extensions du noyau de macOS (*kext*) présente des défis uniques, comme l'absence de code source, la transition vers Apple Silicon et le manque de maturité des outils d'analyse sur cette nouvelle architecture. Ces contraintes compliquent l'identification des mécanismes de communication avec les *kext* et la création d'une grammaire des cas de tests. De plus, les difficultés liées à l'instrumentation, telles que la collecte de la couverture de code et la détection de bugs complexes, limitent l'efficacité du fuzzing.

Bien que certains de ces défis aient été partiellement addressés, les recherches futures devraient se concenter sur l'amélioration des techniques de génération des cas de tests afin d'atteindre des bugs subtils et difficiles à déclencher. La collaboration de plusieurs fuzzers pourrait tirer parti de stratégies variées pour maximiser l'exploration du code. Par ailleurs, une évaluation approfondie des fuzzers de *kext* serait essentielle, notamment en analysant les coûts des différentes techniques d'instrumentation par rapport aux bénéfices obtenus. Enfin, il serait pertinent de les confronter aux dernières versions de macOS, et d'étudier leurs capacité à détecter des bugs récents et connus.

References

[1] C. Fitzl. "Uncovering apple vulnerabilities: The diskarbitrationd and storagekitd audit story part 1." (Nov. 2024), [Online]. Available: https://www.kandji.io/blog/macos-audit-story-part1 (visited on 01/17/2025).

[2] X. Bai, L. Xing, M. Zheng, and F. Qu, "Idea: Static analysis on the security of apple kernel drivers," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1185–1202.

[3] Apple. "Apple security releases." (2025), [Online]. Available: https://support.apple.com/en-us/100100 (visited on 01/20/2025).

[4] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–38, 2023.

[5] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[6] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[7] A. Helin. "Radamsa." (2018), [Online]. Available: https://gitlab.com/akihe/radamsa (visited on 01/20/2025).

[8] "Syzkaller - kernel fuzzer." (2015), [Online]. Available: https://github.com/google/syzkaller (visited on 10/28/2024).

[9] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[10] "American fuzzy lop." (2013), [Online]. Available: https://github.com/google/AFL (visited on 10/28/2024).

[11] "Libfuzzer - a library for coverage-guided fuzz testing." (), [Online]. Available: https://llvm.org/docs/LibFuzzer.html (visited on 10/28/2024).

[12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, USENIX Association, Jun. 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[13] J. Clover. "Apple silicon: The complete guide." (2024), [Online]. Available: https://www.macrumors.com/guide/apple-silicon/ (visited on 12/10/2024).

[14] E. van der Watt. "Arm vs x86: The future of competing computing architecture." (2024), [Online]. Available: https://versus.com/en/news/arm-vs-x86 (visited on 12/10/2024).

[15] Apple. "Preparing your app to work with pointer authentication." (), [Online]. Available: https://developer.apple.com/documentation/security/preparing-your-app-to-work-with-pointer-authentication (visited on 12/09/2024).

[16] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "Syzgen: Automated generation of syscall specification of closed-source macos drivers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 749–763.

[17] T. Yin, Z. Gao, Z. Xiao, Z. Ma, M. Zheng, and C. Zhang, "Kextfuzz: A practical fuzzer for macos kernel extensions on apple silicon," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[18] Apple. "Entitlements." (2019), [Online]. Available: https://developer.apple.com/documentation/bundleresources/entitlements (visited on 12/09/2024).

[19] M. Firouzi. "Pishi: Coverage guided macos kext fuzzing." (2024), [Online]. Available: https://r00tkitsmm.github.io/fuzzing/2024/11/08/Pishi.html (visited on 01/22/2025).

[20] I. Fratric. "Simple macos kernel extension fuzzing in userspace with ida and tinyinst." (2024), [Online]. Available: https://googleprojectzero.blogspot.com/2024/11/simple-macos-kernel-extension-fuzzing.html (visited on 01/22/2025).

[21] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "Kafl: Hardware-assisted feedback fuzzing for os kernels," in *26th USENIX security symposium (USENIX Security 17)*, 2017, pp. 167–182.

[22] H. Oakley. "How macos is moving away from kernel extensions." (2024), [Online]. Available: https://eclecticlight.co/2024/07/24/how-macos-is-moving-away-from-kernel-extensions/ (visited on 12/09/2024).

[23] Y. Lan, D. Jin, Z. Wang, W. Tan, Z. Ma, and C. Zhang, "Thunderkaller: Profiling and improving the performance of syzkaller," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2023, pp. 1567–1578.

[24] "Kernel address sanitizer (kasan)." (), [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html (visited on 12/09/2024).

[25] L. Di Bartolomeo, "Armwrestling: Efficient binary rewriting for arm," M.S. thesis, ETH Zurich, 2021. [Online]. Available: http://hdl.handle.net/20.500.11850/474088 (visited on 08/21/2024).

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

[27] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, Jun. 2020, pp. 151–163, ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385972. [Online]. Available: https://dl.acm.org/doi/10.1145/3385412.3385972 (visited on 07/24/2024).

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

[29] Y. Chen, Y. Jiang, F. Ma, *et al.*, "EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1967–1983.

[30] R. Malmain, A. Fioraldi, and F. Aurélien, "Libafl qemu: A library for fuzzing-oriented emulation," in *BAR 2024, Workshop on Binary Analysis Research, colocated with NDSS 2024*, 2024.

[31] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.