Recherche de vulnérabilités logicielles Fuzzing et Optimisation

Rapport technique

Réalisé par :

Khanh Duy NGUYEN TU

Tuteur entreprise : Benjamin DUFOUR

Tuteur école : Angeliki KRITIKAKOU

TABLE DES MATIERES

1 CONTEXTE GENERAL

1.1 FUZZING

Le fuzzing est une technique de test logiciel qui consiste à injecter des données aléatoires inattendues, potentiellement invalides dans un programme pour découvrir les vulnérabilités et les erreurs. En bombardant un système avec une vaste gamme d'entrées, le fuzzing vise à forcer le logiciel à des comportements inattendus, révélant ainsi des vulnérabilités potentielles.

La figure 1 montre le fonctionnement du fuzzing. Les outils de fuzzing génèrent divers types de données d'entrée, comme des fichiers, des paquets réseau ou des entrées utilisateur, qui sont intentionnellement incorrectes ou incomplètes. Ces données mal formées sont ensuite transmises au logiciel testé. Le fuzzer observe attentivement le comportement du programme, à la recherche de tout signe d'instabilité, comme des crashs, des blocages ou des failles de sécurité.

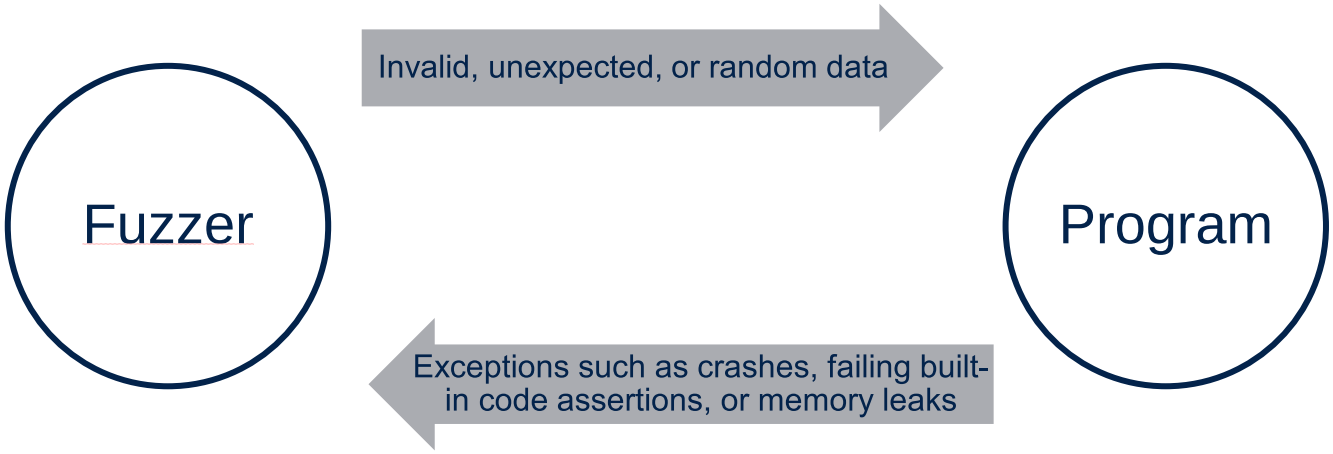

Figure 1 : Le flux de travail de Fuzzing

Le fuzzing est efficace pour découvrir des vulnérabilités cachées qui pourraient échapper à la détection par les méthodes de test traditionnelles. En identifiant et en traitant ces problèmes durant tout le processus de développement, le fuzzing contribue à la création d'un logiciel plus robuste et fiable. De plus, la nature automatisée du fuzzing le rend très efficace pour tester de grandes bases de code.

Il existe principalement trois types de fuzzing :

1. Boîte noire : aucune connaissance du programme cible.
2. Boîte blanche : information complète du programme cible.
3. Boîte grise : combinaison des deux autres types.

Du point de vue de la génération des cas de test, il existe deux types de fuzzing : basé sur la mutation (Mutation-based) et basé sur la génération (Generation-based) (ou basé sur la grammaire : Grammar-based). Le fuzzing basé sur la mutation consiste à modifier les cas de test existants pour en créer de nouveaux. Il commence par un ensemble de cas de test préexistants, connu sous le nom de seed, et applique ensuite diverses mutations pour générer de nouveaux cas de test. Il est principalement utilisé lorsqu'un ensemble bien rempli de cas de test valides est déjà disponible. Cette approche est efficace pour trouver des écarts dans le comportement du logiciel lorsqu'il est soumis à des entrées valides légèrement modifiées.

Le fuzzing basé sur la génération crée des entrées de test à partir d'un modèle ou de spécifications de formats d'entrée valides. Cette approche est particulièrement utile pour tester des systèmes avec des formats d'entrée bien définis et riches, tels que les compilateurs, les interpréteurs ou les implémentations de protocoles.

Selon OSS-Fuzz [1], un service créé par Google qui exécute des fuzzers sur des projets open source, plus de 10 000 vulnérabilités et 36 000 bogues sur 1 000 projets ont été détectés et corrigés à l'aide du fuzzing. J'ai regroupé quelques uns de ces CVEs dans le tableau 1.

CVE-2020-13396	OOB Read in FreeRDP
CVE-2020-13397	NULL dereference in FreeRDP
CVE-2020-13398	Heap Overflow in FreeRPD
CVE-2020-11099	OOB Read in FreeRPD
CVE-2020-11097	OOB Read in FreeRPD
CVE-2020-11098	OOB Read in FreeRPD
CVE-2020-29129	OOB Read in QEMU
CVE-2021-27804	OOB Write in JPEG XL
CVE-2024-22857	Heap Overflow in ZLOG

Tableau 1 : Certains CVEs ont été découverts à l'aide de fuzzing [2]

1.2 AFL++

AFL++ [3] [4] est un framework de fuzzing de pointe qui s'appuie sur le succès de son prédécesseur, American Fuzzy Lop (AFL [5] [6]). Il intègre une multitude d'améliorations, d'optimisations et de fonctionnalités innovantes pour exceller dans la découverte des vulnérabilités logicielles.

L'efficacité de l'AFL+ repose sur son mécanisme de rétroaction utilisant la couverture de code (*coverage-based feedback mechanism*). Non seulement les erreurs ou vulnérabilités sont gardées, mais AFL++ essaie aussi de trouver des résultats "intéressants" pour expérimenter plus loin. Un résultat est considéré comme "intéressant" selon la quantité de code que le programme a exécuté, aussi appelé couverture de code. Si un cas de test permet d'exécuter plus de code qu'auparavant, alors c'est un résultat "intéressant" et il sera conservé pour la mutation et l'expérimentation. Il existe de nombreux critères de couverture tels que :

- Couverture des fonctions (Function Coverage) : les appels fonctions sont suivis.
- Couverture des blocs (Block Coverage) : les blocs de base dans le graphe de flot de contrôle sont suivis.
- Couverture des arêtes (Edge Coverage) : les arêtes dans le graphe de flot de contrôle sont suivies.

AFL++ fonctionne avec l'Edge Coverage qui mesure le pourcentage d'arêtes dans le graphe de flot de contrôle qui ont été exécutées. En informatique, un graphe de flot de contrôle (GFC) est une représentation, utilisant la notion mathématique de graphe, de tous les chemins qui peuvent être traversés par un programme pendant son exécution. Dans la figure 2, je montre deux GFC simples qui décrivent une condition if-then-else (à gauche) et une boucle (à droite).

Figure 2 : Simple GFC d'un if-else-then et d'une boucle

Le processus de fuzzing en résumé implique d'instrumenter le programme cible pour suivre la couverture du code, créer un ensemble initial de fichiers d'entrée (corpus), modifier ces fichiers pour produire de nouveaux cas de test, exécuter les entrées modifiées, et analyser les informations de couverture pour prioriser les meilleures cas de test.

Ce framework offre une grande flexibilité avec ses options d'instrumentation étendues, prenant en charge diverses plates-formes cibles et types de programmes via l'intégration LLVM, QEMU et Unicorn. Pour maximiser les chances de découvrir des vulnérabilités cachées, AFL++ emploie un large ensemble d'opérateurs de mutation pour générer divers cas de test.

De plus, AFL++ va au-delà du fuzzing traditionnel en combinant différentes stratégies, telles que les approches basées sur la mutation et les approches basées sur la génération, dans son mode de fuzzing hybride. Il fournit également des outils pour l'analyse approfondie et la minimisation des crashs, aidant à leur analyse. Les optimisations de performance accélèrent encore le processus de fuzzing, permettant une détection efficace des vulnérabilités.

AFL++ a été reconnue pour sa grande efficacité dans la découverte rapide des vulnérabilités. Sa polyvalence s'étend à la fusion de divers types de logiciels, y compris les protocoles réseau, les formats de fichiers et les applications web. Soutenu par une communauté forte, AFL++ offre un support étendu et est facilement extensible avec de nouvelles fonctionnalités. En conséquence, il est devenu un choix privilégié pour les chercheurs et développeurs de sécurité qui souhaitent améliorer la fiabilité des logiciels grâce à des tests rigoureux.

1.3 EMULATEUR

Un émulateur est un logiciel ou un programme matériel qui reproduit les fonctionnalités d'un autre système informatique ou appareil. Il permet à un ordinateur hôte d'imiter le comportement d'un système invité.

Les émulateurs atteignent cet objectif en simulant les composants matériels du système invité, y compris le processeur, la mémoire, les périphériques d'entrée/sortie et autres éléments critiques. Une fois cet environnement matériel virtuel établi, l'émulateur peut exécuter des logiciels conçus pour le système invité.

Les émulateurs servent à diverses fins. Ils permettent aux utilisateurs d'exécuter des logiciels plus anciens sur des systèmes modernes, ce qui facilite la compatibilité. Les développeurs utilisent souvent des émulateurs pour tester des logiciels sur différentes plateformes sans avoir besoin d'accéder physiquement à chaque appareil. De plus, les émulateurs jouent un rôle crucial dans la préservation des logiciels et du matériel informatique historiques à des fins éducatives ou d'archivage. Dans le domaine du divertissement, les

émulateurs permettent aux joueurs de profiter des jeux vidéo classiques sur des plateformes contemporaines.

Les émulateurs sont par exemple ceux conçus pour exécuter des applications Android sur Windows ou macOS, jouer à des jeux Nintendo, PlayStation ou Xbox sur PC et simuler des composants matériels spécifiques à des fins de test ou de débogage.

1.4 UNICORN ENGINE

Unicorn [7] est un framework léger, multi-plateforme et multi-architecture d'émulateur de CPU, basé sur QEMU. Sa vitesse et son efficacité en ont fait un choix populaire pour diverses applications. Les principales caractéristiques de Unicorn comprennent sa prise en charge de plusieurs architectures telles que x86, ARM et MIPS, ainsi que sa compatibilité avec divers systèmes d'exploitation tels que Windows, Linux et macOS. Pour obtenir des performances élevées, Unicorn utilise la compilation Just-in-time des blocs de base lors de l'exécution. Comme QEMU, Unicorn traduit le code assembleur cible en code TCG (Tiny Code Generator), puis traduit le code TCG en code assembleur hôte à exécuter.

Unicorn fournit une interface de bas niveau pour interagir avec le CPU émulé. Pour émuler avec Unicorn, les utilisateurs créent un programme appelé harness. Grâce à ce programme, les utilisateurs peuvent définir l'architecture et le mode, allouer la mémoire pour le programme cible, charger du code en mémoire, configurer des registres et des valeurs de mémoire, démarrer et arrêter l'émulation, et définir des points d'arrêt et des hooks. Hooking est une fonctionnalité importante dans Unicorn qui permet aux utilisateurs d'interagir avec le programme dans son émulation et permet également de combiner Unicorn avec des fuzzers comme AFL++. Je liste les 17 types de hook disponibles dans Unicorn et quand ils sont exécutés dans le tableau 2.

Hook	Exécuté à
UC_HOOK_CODE	Avant d'exécuter chaque instruction d'assemblage
UC_HOOK_BLOCK	Avant d'exécuter chaque bloc de base
UC_HOOK_INTR	Lorsqu'il y a une interruption/syscall
UC_HOOK_INSN	Avant d'exécuter des instructions spécifiques
UC_HOOK_INSN_INVALID	Instructions invalides
UC_HOOK_TCG_OPCODE	Comme INSN mais avec un code de TCG spécifique
UC_HOOK_EDGE_GENERATED	Nouvelle génération de bordures
UC_HOOK_MEM_*	Avant les événements d'accès à la mémoire
UC_HOOK_MEM_*_UNMAPPED, UC_HOOK_MEM_*_PROT	Avant les événements d'accès à la mémoire non valide (utiliser une mémoire non mappée ou protégée)
UC_HOOK_MEM_READ_AFTER	Identique à UC_HOOK_MEM_READ mais seulement déclenché après une lecture réussie

* : READ / WRITE / FETCH

Tableau 2 : Liste des hooks dans Unicorn

1.5 AFL-UNICORN

En combinant AFL++ et Unicorn Engine [8], des binaires de différentes architectures peuvent être testé sur un hôte. Ceci est particulièrement utile pour analyser des logiciels difficiles à déboguer directement sur du matériel, comme les systèmes embarqués ou le code hérité. Normalement, AFL++ a besoin du code source pour instrumenter le programme cible par son propre compilateur. Il a besoin de cette instrumentation pour que le programme cible

calcule lui-même la couverture du code pendant l'exécution. AFL++ peut alors faire le fuzzing sur le programme directement sans aucune autre configuration. En mode Unicorn, il n'y a pas de code source, donc il n'y a pas d'instrumentation. Un programme intermédiaire (*harness*) qui supervise l'émulation du programme cible doit être créé. AFL-Unicorn utilise ensuite un hook_block pour calculer la couverture de code dynamiquement. AFL++ va ensuite fuzzer ce programme.

1.6 SANITIZERS

Les sanitizers sont des applications logicielles conçues pour détecter et signaler les bogues dans un programme en injectant du code supplémentaire pendant le processus de compilation. Ces outils sont indispensables pour améliorer la qualité et la sécurité des logiciels.

Les types courants de sanitizer comprennent :

1. AddressSanitizer (ASan) pour identifier les erreurs de mémoire comme les buffer overflows.
2. MemorySanitizer (MSan) pour détecter les lectures de mémoire non initialisées.
3. ThreadSanitizer (TSan) pour découvrir les situations de compétition et d'autres problèmes liés aux threads.
4. UndefinedBehaviorSanitizer (UBSan) pour repérer les comportements indéfinis dans le code C et C++.

Les sanitizers fonctionnent en instrumentant le programme avec du code supplémentaire qui effectue des vérifications d'exécution pour des conditions spécifiques. Lorsqu'une erreur est détectée, le sanitizer fournit des informations détaillées sur le problème, y compris son emplacement et son type.

Les sanitizers offrent plusieurs avantages, notamment une meilleure qualité des logiciels grâce à la détection et à la correction rapides des bogues. En outre, les sanitizers jouent un rôle crucial dans l'amélioration de la sécurité en identifiant des vulnérabilités comme les buffer overflows qui peuvent être exploitées par des acteurs malveillants. Ces outils accélèrent également le débogage en fournissant des rapports d'erreur précis, favorisant une confiance accrue dans la justesse du code.

Les sanitizers sont largement utilisés pour découvrir des failles de sécurité, améliorer la fiabilité du code et même optimiser les performances en identifiant les goulots d'étranglement potentiels.

2 SUJET DE STAGE

Pour ce stage, l'objectif est d'améliorer le processus de fuzzing de binaires embarqués avec l'outil AFL-Unicorn soit en optimisant les performances, soit en améliorant la précision en développant un sanitizer d'adresse. Ce stage peut être résumé en quatre parties :

1. Comprendre comment AFL-Unicorn fonctionne, aller en profondeur sur certains points clés liés à la performance.
2. A l'aide de cette connaissance, proposer et mettre en œuvre des optimisations.
3. Faire un état de l'art des sanitizers puis implémenter un sanitizer d'adresse.
4. Eventuellement corriger certains bogues importants dans AFL-Unicorn, Unicorn.

Parmi les trois parties liées au développement, la priorité est de mettre en place un sanitizer d'adresse pour les binaires qui peuvent travailler avec AFL-Unicorn, Ce sanitizer se concentrera sur les programmes ARM en mode THUMB, et compilés avec l'optimisation "-Os", qui vise à réduire la taille du code compilé, couramment utilisé pour les programmes embarqués.

3 ÉTAT DE L'ART DES SANITIZERS BINAIRE

3.1 OVERVIEW

Les sanitizers se sont avérés être une aide précieuse pour trouver des vulnérabilités dans les programmes, cependant la plupart des sanitizers nécessitent l'accès au code source. Cela est dû à certains obstacles techniques qui se forment durant le processus de compilation [9]:

1. Perte d'information : Le premier obstacle est la perte d'information causée par le processus de compilation. Les sanitizers binaires doivent reconstruire ces informations perdues pour déterminer les emplacements corrects des objets, tout en maintenant la sémantique du binaire. La reconstruction dépend souvent de l'heuristique, ce qui conduit généralement à des inexactitudes. Les catégories typiques d'information perdues pendant la compilation sont l'information sur le flux de contrôle, les types, l'ordre de mémoire, le signe et l'information de débogage. Cette perte d'information est causée par des différences conceptuelles entre la langue source et l'architecture cible.
2. Différences conceptuelles dans les représentations des programmes. Le deuxième obstacle englobe les différences conceptuelles dans les représentations des programmes. Les différentes représentations d'un programme (code source, représentation intermédiaire (IR) et assembleur) ont des propriétés différentes qui peuvent simplifier ou entraver l'analyse statique et doivent donc être prises en compte lors du développement de sanitizer. Le comportement indéfini, par exemple, est un concept utilisé dans les langages de haut niveau comme C/C++ pour guider les optimisations des compilateurs. Cependant, il n'y a pas de concept de comportement indéfini au niveau binaire, car seules des instructions concrètes sont émises.

En raison de ces problèmes, chaque type de sanitizer (ASan, UBSan, MSan, TSan) aura une faisabilité différente au niveau binaire [9]:

1. ASan : ASan peut être appliqué à des cibles binaires en général, car son instrumentation est relativement légère, et la bibliothèque de runtime ASan implémente la plupart de la logique. Par exemple, la gestion des zones rouges, une technique couramment utilisée dans des implémentations d'ASan, fait partie de la bibliothèque de runtime et ne nécessite pas d'instrumentation binaire. Bien qu'un port binaire d'ASan soit généralement possible, la perte d'information nuit à l'efficacité d'une version binaire d'ASan.
2. UBSan : UBSan est différent de tous les autres sanitizers, car il utilise une multitude de petites vérifications individuelles, qui nécessitent toutes une mise en œuvre indépendante. Cependant, le problème majeur avec une version binaire d'UBSan est que bien que des comportements non définis puissent être utilisés pour l'optimisation lors de la compilation, ce concept n'existe plus au niveau binaire. Il

faut plutôt reconstruire l'intention d'origine pour déterminer si le comportement du code source n'était pas défini. C'est une tâche difficile et sujette aux erreurs.

3. MSan : MSan dispose d'une grande bibliothèque de runtime pour la détection de mémoire non initialisée. Cependant, l'instrumentation de MSan est assez complexe. La raison principale est que la mémoire non initialisée doit être correctement suivie tout au long de l'exécution du programme. L'état de la mémoire est stocké dans ce qu'on appelle la shadow memory. La propagation de shadow sert à mettre à jour l'état de la mémoire si celle-ci est utilisée ou altérée. Au niveau binaire, c'est encore plus complexe puisque le contenu des registres peut influencer l'état d'initialisation de la mémoire. Un exemple typique est le chargement d'une mémoire éventuellement non initialisée dans un registre et le stockage du contenu du registre dans un autre emplacement de mémoire non initialisée. Ainsi, pour mettre en œuvre la propagation, l'état de chaque registre et son contenu doivent être pris en compte.
4. TSan : L'instrumentation du TSan est assez légère, car seuls les accès à la mémoire doivent être instrumentés et la logique de détection est déléguée à une bibliothèque de runtime. Cependant, l'ordre de mémoire des opérations atomiques est perdu pendant la compilation. Par conséquent, il faut développer des heuristiques pour identifier les variables atomiques afin de compenser la perte d'information.

Pour ce stage, je vais me concentrer sur la recherche et le développement de sanitizer d'adresse parce qu'il est plus facile à implémenter au niveau binaire et les erreurs de mémoire sont plus communnes.

3.2 SANITIZER D'ADRESSE

Le sanitizer d'adresse est un programme qui détecte simplement les violations de mémoire (Overflow, Underflow, Memory Leak, Use-after-free...). La plupart des sanitizers d'adresse se classent dans l'une des catégories suivantes :

1. Location-based
2. Identity-based

3.2.1 Location-based sanitizers

Un location-based sanitizer fonctionne en insérant des régions de mémoire non valides, appelés des zones rouges, entre les objets déclenchant des erreurs en cas d'accès. Ils sont principalement utilisés pour détecter les dépassements. Pour les objets dans le tas, chaque fois qu'ils sont libérés, les régions de mémoire deviennent invalides. Cela permet donc aux sanitizers de détecter également les Use-After-Free et Double Free. En dehors des zones rouges, une shadow memory est également utilisée pour afficher l'état de chaque octet de mémoire (adressable ou non). Chaque octet de shadow memory représente l'état de 8 octets de la mémoire réelle. Ceci permet de vérifier rapidement l'état de la mémoire.

Avantages

1. Vérification rapide de l'accès
2. Mise en œuvre simple

Inconvénients

1. Nécessite une grande mémoire pour la shadow memory et les zones rouges, cependant dans le contexte du logiciel embarqué, la taille de la mémoire cible est si petite qu'elle ne pose pas de problème.
2. Il a des problèmes pour détecter certains dépassements.

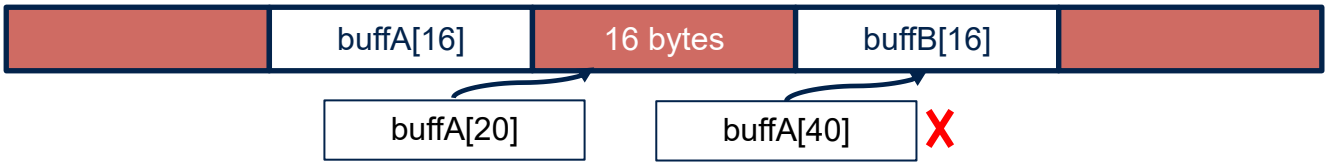

Figure 3 : Problème de Location-based sanitizer, la croix rouge indique que le dépassement n'est pas détecté.

En effet, dans la situation de la figure 3, le sanitizer ne va pas détecter le dépassement du tampon A s'il ne tombe pas sur les zones rouges (buffA[40]). Des zones rouges plus grandes pourraient être utilisées pour minimiser ce problème, mais cela ne fait que rendre le premier problème plus critique.

3.2.2 Identity-based sanitizers

L'identity-based sanitizer est conçu pour résoudre les problèmes liés au Location-based sanitizer. Comme le montre la figure 4, L'identity-based attribue une identité à chaque pointeur et à la zone de mémoire vers laquelle il pointe. Les identités du pointeur et de la mémoire doivent être identiques pour qu'un accès se produise. Cela permet au sanitizer de détecter plus d'infractions.

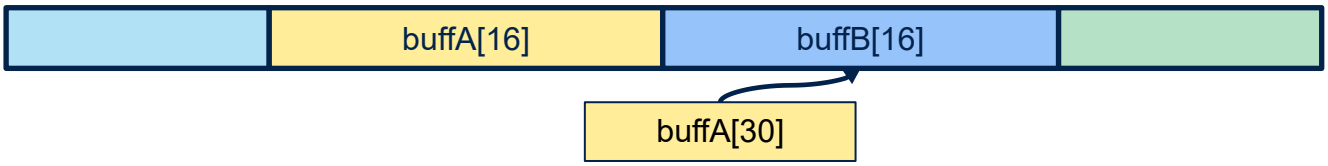

Figure 4 : Identity-based sanitizer, chaque couleur correspond à une identité différente.

Avantages

1. Couvre plus de violations de mémoire.
2. Nécessite moins de mémoire.

Inconvénients

1. Vérification d'accès plus compliquée et plus lente.

3.2.3 Méthode d'instrumentation

En ce qui concerne la méthode d'instrumentation pour le sanitizer binaire, il y a deux façons de faire :

1. Instrumentation binaire dynamique (Dynamic Binary Instrumentation) : instrumenter le binaire pendant qu'il est en cours d'exécution. Cette méthode est la plus utilisée en raison de sa flexibilité et de sa simplicité, cependant cette méthode pose un certain nombre de problèmes de surconsommation de runtime.

2. Réécriture binaire (Binary Rewriting) : Cette méthode réduit les surconsommations de runtime puisque l'instrumentation est faite de façon statique. Cependant, elle est plus complexe à mettre en œuvre.

3.2.4 Les défis de l'élaboration d'un sanitizer binaire

1. Manque d'information : beaucoup d'informations sur les objets, en particulier sur la pile, sont rejetées après la compilation rendant l'identification du pointeur vraiment difficile.
2. Surconsommation de runtime : l'instrumentation d'un binaire à l'exécution peut entraîner des coûts de performance importants, ce qui peut affecter la facilité d'utilisation de l'application.
3. Surconsommation de mémoire : le sanitizer d'adresse nécessite généralement une mémoire supplémentaire pour stocker les métadonnées sur les allocations de mémoire et les accès.
4. Gestion du code optimisé : les compilateurs appliquent souvent diverses optimisations qui peuvent obscurcir la structure de code d'origine, ce qui rend difficile l'insertion correcte de l'instrumentation.
5. Gestion des formats binaires différents : les binaires peuvent être proposés dans divers formats (p. ex., ELF, PE, Mach-O), chacun ayant sa propre structure et ses propres conventions.
6. Compatibilité avec le code existant : veiller à ce que le binaire instrumenté reste compatible avec les bibliothèques et appels système.

3.3 ÉTAT DE L'ART DES SANITIZERS

3.3.1 QASan

QASan [10] est un outil spécialisé qui combine les capacités de l'émulateur QEMU avec les fonctionnalités de détection d'erreurs de mémoire de AddressSanitizer (ASan).

Comme ASan, QASan est un location-based sanitizer. Il intercepte les appels à malloc / calloc / realloc pour insérer des zones rouges entre les objets du tas afin de détecter une erreur de dépassement. Il suit également les appels à free pour transformer la mémoire allouée en zone rouge afin de détecter les Use-After-Free et Double Free. En raison de la perte d'information, QASan ignore les erreurs dans les variables globales et dans la pile.

La figure 5 montre l'architecture de QASan. QASan a deux composants principaux, le runtime QASan et l'extension QASan, tous deux intégrés dans QEMU. Le runtime QASan réside dans la cible émulée. Il intercepte les appels à malloc, free et les remplace avec ses propres implémentations de malloc et free pour insérer et gérer les zones rouges. QASan extension est en dehors de la cible émulée et il stocke la shadow memory. Il contient également des fonctions de génération TCG modifiées pour pouvoir accéder à la mémoire instrumentée pendant la compilation du code TCG vers le code hôte. Les deux composants communiquent par des *hypercalls* personnalisés.

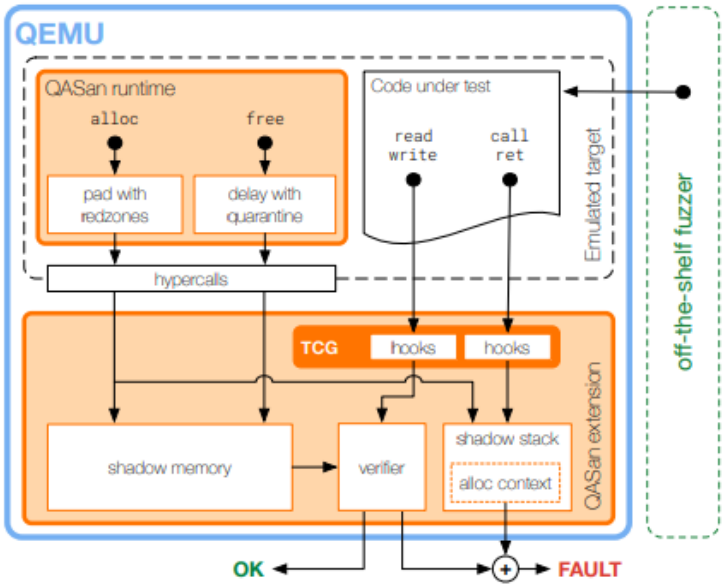

Figure 5 : Architecture de QASan [10] .

Bien que QASAN améliore significativement la détection d'erreurs de mémoire dans les environnements virtualisés, il est essentiel de considérer les surcoûts potentiels en termes de performances introduits par QEMU et ASan. De plus, QASan ne détecte que les erreurs de tas.

3.3.2 µSBS

µSBS [11] , ou micro-SBS, est un outil de sécurité spécialement conçu pour protéger les dispositifs embarqués bare-metal. Ces périphériques, fonctionnant directement sur du matériel sans système d'exploitation. Par conséquent, ces périphériques ne disposent pas de mécanismes de sécurité tels que MMU (Memory Management Unit). Les violations de mémoire dans ces périphériques ne seront pas observables sous forme de crashs ou d'erreurs. µSBS est un sanitizer binaire qui répond à ce défi en utilisant la réécriture binaire.

µSBS fonctionne en trois phases principales :

1. Static disassembling : désassemble le binaire et décode les instructions.
2. Binary instrumentation : instrumente les binaires selon la spécification du sanitizer (ASan, location-based).
3. Reassembling : Réassemble le binaire.

µSBS est inspiré par ASan. Il instrumente chaque instruction de mémoire (c'est-à-dire charge et stockage), entoure chaque valeur de mémoire avec des zones rouges. Ce qui est spécial, c'est la façon dont il instrumentalise le binaire. Au lieu de modifier le code, il réécrit tout avec l'instrumentation dans une autre section. Il crée également une table de mappage pour toutes les instructions de branche indirectes. Ceci est montré plus clairement sur la figure 6. En utilisant la méthode de réécriture binaire, µSBS peut réduire le temps d'exécution.

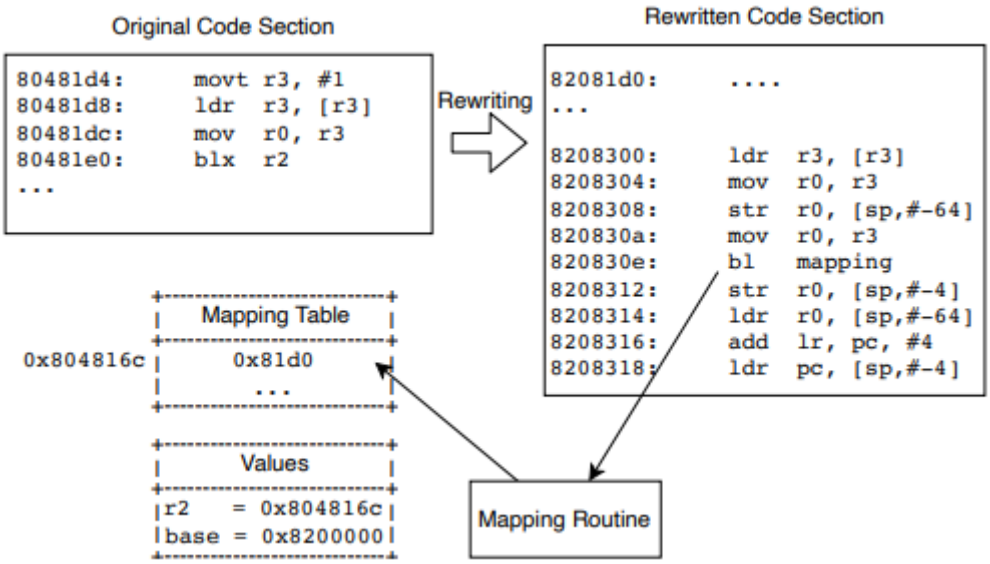

Figure 6 : Rewriting et Mapping [11]

3.3.3 IdSan

IdSan [12] est un sanitizer identity-based permettant d'identifier les vulnérabilités liées à la mémoire. Il est créé pour se concentrer sur les erreurs mémoire dans les variables globales et la pile.

La figure 7 montre le principe d'IdSan. Son concept de base consiste à attribuer des identités uniques aux régions mémoire et pointer puis suivre leur utilisation dans les registres tout au long de l'exécution du programme. IdSan analyse chaque instruction binaire pendant l'exécution et la met en correspondance avec une opération qui assigne des identités ou interagit avec les métadonnées. Pour résoudre le problème de la perte d'information dans l'identification des pointeurs de pile et globaux, il utilise une heuristique qui fait l'hypothèse que certains ensembles d'instructions sont préférentiellement utilisés pour accéder et manipuler ces pointeurs.

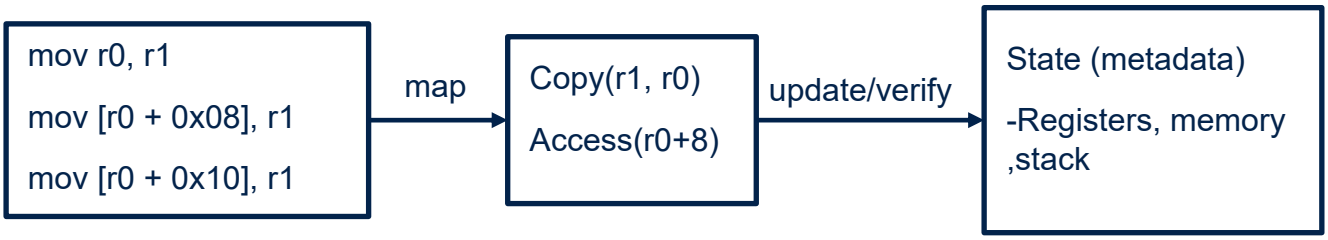

Figure 7 : Principe d'IdSan

Cette heuristique perd en efficacité lorsque le compilateur optimise les accès à la mémoire et fonctionne dans un comportement qui est différent.

3.3.4 MTSan

MTSan [13] est un sanitizer de mémoire conçu pour détecter les violations de sécurité de la mémoire spatiales et temporelles lors de l'exécution du programme. Il représente une avancée significative dans la détection d'erreurs de mémoire en combinant plusieurs techniques innovantes.

MTSan est un sanitizer basé sur l'identité qui utilise les fonctionnalités d'accélération matérielle ARM Memory Tagging Extension (MTE) pour réduire le temps d'exécution. Le marquage de la mémoire (Memory Tagging) ajoute des balises uniques (sous forme de bits)

aux pointeurs et à l'espace mémoire. Ila été d'abord introduit par ARM dans ARMv8.5-A, et a commencé à être intégré dans les processeurs conformes à ARMv9. MTE utilise la fonction Top Byte Ignore (TBI) qui a été introduite dans ARMv8.1. TBI permet aux processeurs ARM d'ignorer le premier octet de chaque pointeur. Cet octet ignoré peut alors être utilisé pour stocker des métadonnées supplémentaires. Ces balises sont propagées par les processeurs ARM sans surcout en temps d'exécution.

Pour améliorer son efficacité, MTSan utilise un système de récupération progressive des objets pour les reconstruire. MTSan déduit d'abord les propriétés des objets comme les limites et la durée de vie. Ensuite, tout au long du processus de fuzzing, ces propriétés seront de plus en plus raffinées grâce à des exécutions uniques.

MTSan introduit également le concept de violations non critiques. Il s'agit de violations mineures de propriétés présumées (p. ex., objets empilés en raison d'une perte d'information). Ces violations sont considérées comme des faux positifs, mais le testcase associé est conservé dans l'espoir de déclencher une véritable violation ultérieurement. Ce processus est appelé Record, Resume et Regression.

La figure 8 résume les composants, les fonctionnalités et le fonctionnement de MTSan

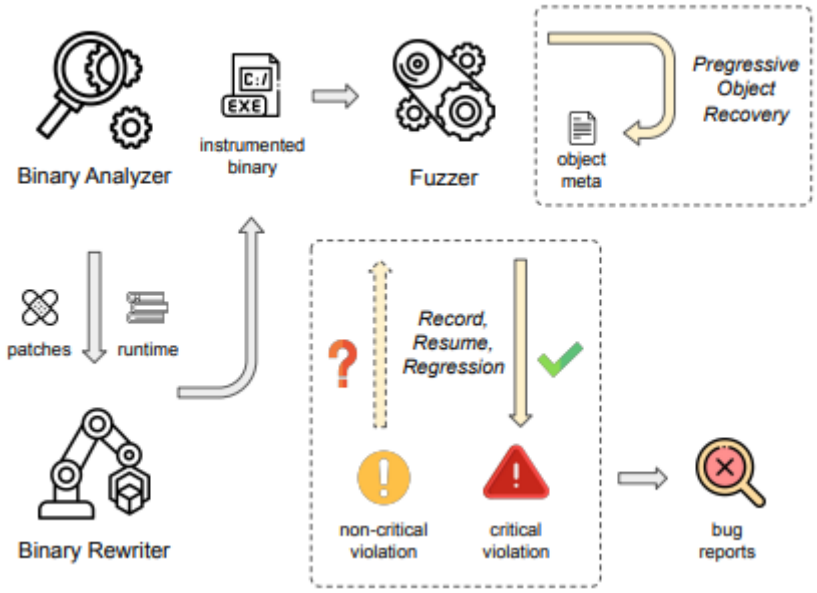

Figure 8 : Fonctionnement de MTSan

En combinant la récupération d'objets, la réécriture binaire et l'accélération matérielle, MTSan offre une approche complète pour détecter les problèmes de sécurité de la mémoire. Cela en fait un outil précieux pour les développeurs et les chercheurs en sécurité qui souhaitent améliorer la fiabilité et la sécurité des logiciels.

3.3.5 Comparaison

Le tableau 3 résume et compare les quatre sanitizers en fonction de leur type, de la méthode d'instrumentation, des architectures qu'ils supportent et de leur capacité à détecter les erreurs sur le tas, la pile et les variables globales :

Nom	Fonctionnement	Instrumentation	Architecture	Pile	Tas	Globales
QASan	Location	Dynamique	x86, x86-64, ARM, AArch64	Oui	Non	Non
µSBS	Location	Binary Rewriting	ARM	Oui	Oui	Pas indiqué
IdSan	Identity	Dynamique	AArch64	Non	Oui	Oui
MTSan	Identity	Binary Rewriting	AArch64	Oui	Oui	Oui

Tableau 3 : Comparaison des sanitizers binaires existants.

4 CONTRIBUTIONS

En résumé, à date du présent rapport, j'ai apporté les contributions suivantes.

1. Proposer, mettre en œuvre et tester une optimisation dans AFL-Unicorn
2. Analyser un bogue dans Unicorn
3. Mise en œuvre d'un sanitizer d'adresse binaire sur l'émulateur Unicorn Engine

4.1 OPTIMISATION D'AFL-UNICORN

AFL-Unicorn a été créé pour permettre aux utilisateurs de fuzzer des binaires. En utilisant un émulateur, il compromet ses performances. Après avoir analysé AFL-Unicorn et aussi Unicorn, j'ai identifié deux points clés compromettant les performances :

1. L'organisation des hook dans Unicorn.
2. Le processus de rétroaction sur la couverture du code dans AFL-Unicorn (Code Coverage Feedback).

4.1.1 Structure de données de hook

Comme le montre la figure 9, les hooks dans Unicorn sont stockés dans une combinaison de tableau et de liste chaînée. D'abord il y a un tableau de taille 17, chaque case permet de stocker un type de hook. Tous les hooks d'un même type sont chaînés ensemble, de sorte que nous avons seulement besoin de stocker le premier dans le tableau.

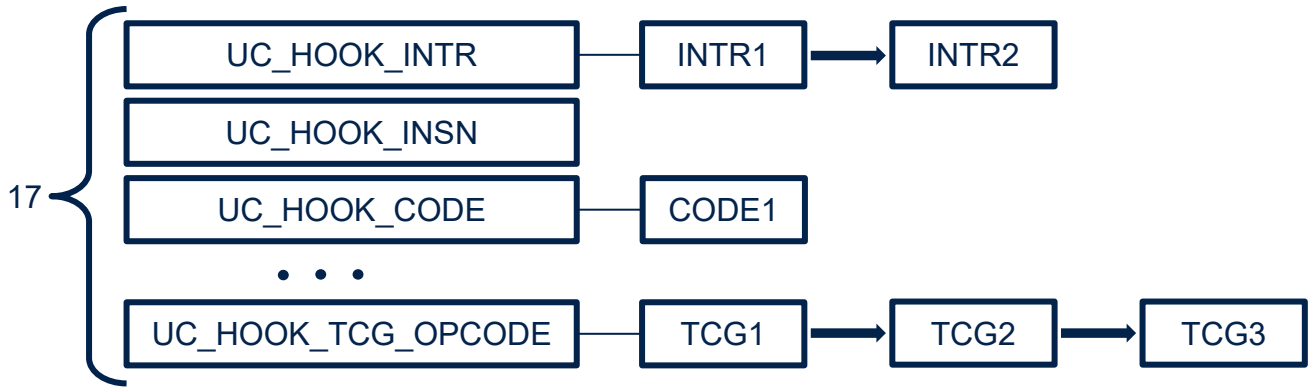

Figure 9 : Comment les hooks sont stockés dans Unicorn

Pour cette raison, la complexité temporelle pour trouver un hook est O(n), l'ajout d'un hook est O(1), le retrait d'un hook est O(n). Ceci est en contradiction avec le fonctionnement normal dans lequel nous ajoutons une fois au début mais accédons aux hooks plusieurs fois tout au long de l'exécution. De plus, lorsque nous supprimons un hook, Unicorn ne le supprime pas de la liste mais le met dans un état "supprimé". De cette façon, le temps de parcours de la liste reste le même après avoir enlevé ce hook.

4.1.2 Code Coverage Feedback

Dans AFL-Unicorn, les informations sur la couverture du code sont stockées dans une table de hachage où l'index est l'identifiant d'une arête et la valeur est le nombre de fois qu'elle a été exécutée. Chaque fois qu'un bloc de code est rencontré, un hook_block réservé à la couverture du code est appelé. Ce hook calcule l'identifiant du bloc en hashant son adresse. Cet identifiant sera ensuite xorée (fonction OU exclusif) avec l'identifiant du dernier bloc, créant ainsi l'identifiant de l'arête entre les deux blocs. La valeur indexée par cet identifiant dans le tableau "__afl_area_ptr" sera augmentée d'un. Le seul problème de performance lié à ce processus est que l'identifiant du bloc est calculé à chaque fois, même si le bloc a déjà été rencontré.

4.1.3 Propositions et essais

Certaines propositions ont été faites pour améliorer les performances d'AFL-Unicorn :

1. Cache d'identifiant de bloc : en théorie, en mettant en cache les identifiants des blocs, nous n'avons pas à les recalculer si nous rencontrons d'anciens blocs. Par conséquent, nous devrions réduire le temps d'exécution.
2. Changer la structure de données des hooks: comme expliqué au §4.1.1, trouver les hooks est une opération qui est répétée très souvent alors qu'il est rare d'ajouter ou de supprimer un hook. Un tableau avec la complexité de O(1) ou un arbre binaire de complexité O(log2n) pour la recherche d'élément pourrait être un meilleur candidat que la liste chaînée.

Parmi ces propositions, le cache d'identité a été mis en œuvre et testé, cependant il n'a pas montré d'amélioration de la performance. Pour le deuxième cas, je n'ai pas assez de temps pour le tester

4.2 BUG UNICORN

Il y a un bogue intéressant qui se rapporte aux hooks Unicorn [14]. Dans une situation où nous souhaitons effectuer un Man-in-the-middle sur l'accès mémoire tel que décrit dans la figure 10, nous ne stockons rien dans la mémoire, la mémoire est marquée comme non inscriptible (par le programme lui-même, les utilisateurs peuvent toujours écrire dessus en utilisant la fonction d'écriture mémoire de Unicorn). Ce n'est que lorsqu'un pointeur a besoin de lire une mémoire qu'on écrit une valeur à cette adresse.

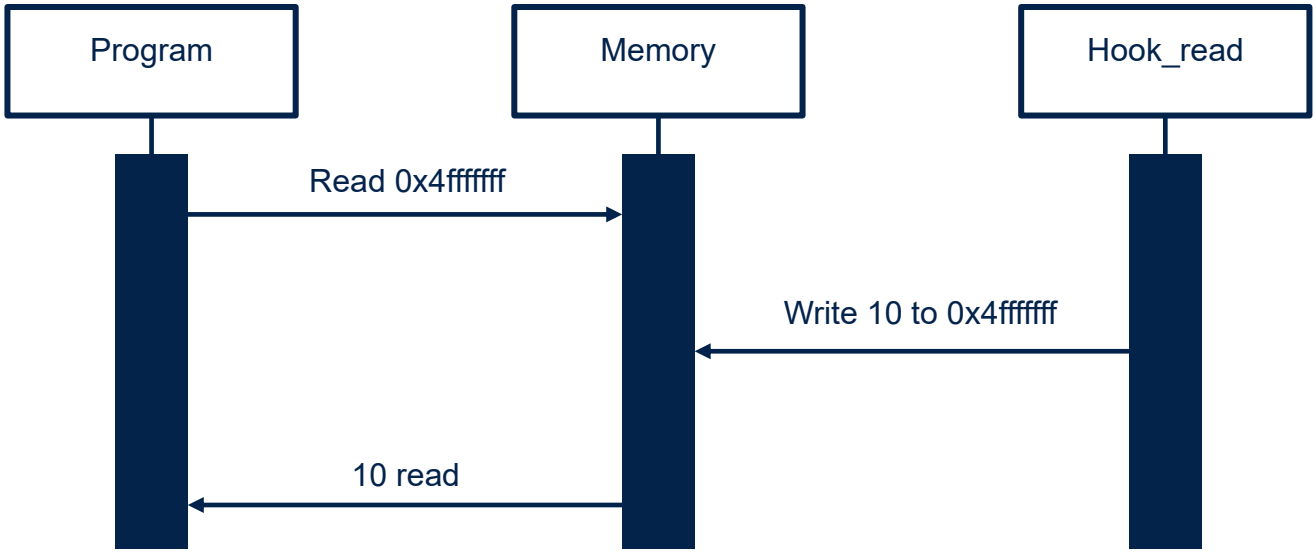

Figure 10 : Man-in-the-middle lecture dans Unicorn

Cette idée pourrait être implémentée dans Unicorn en configurant d'abord un hook_mem_read. Puisque le hook_mem_read est exécuté avant la lecture, nous pouvons l'utiliser pour effectuer la fonction mem_write à l'adresse où il va être lu. Cependant, lors des tests, cette méthode entraîne une erreur de segmentation : l'adresse de lecture a été modifiée.

Après avoir analysé le code source de Unicorn, j'ai constaté que parce que la mémoire est marquée comme non inscriptible, lorsque nous écrivons dessus en utilisant mem_write function, un ensemble de fonctions dans Unicorn est appelé pour marquer temporairement la mémoire comme inscriptible. Dans cette chaîne de fonctions, il existe une fonction appelée tlb_flush qui purge le cache TLB (Translation Lookaside Buffer). Dans Unicorn, ce cache contient des décalages pour le calcul d'adresses pour les accès mémoire. Avant d'exécuter le hook, Unicorn assigne un pointeur à un enregistrement sur le TLB. Après l'exécution du hook, le TLB est vidé, le pointeur pointe maintenant sur des données non valides. Par conséquent, le calcul effectué par la suite est incorrect, ce qui entraîne l'accès à une mauvaise adresse.

L'analyse du bogue [14] a été notifiée aux développeurs de Unicorn, cependant aucune réponse n'a été reçue à ce jour.

4.3 YUDSAN

4.3.1 Overview

YudSan est conçu pour être un sanitizer d'adresse basé sur l'identité qui est compatible avec Unicorn. Il est inspiré par IdSan [12] et MTSan [13]. Il analyse le code assembleur du programme cible pour ensuite créer, mettre à jour des métadonnées ou vérifier les conditions d'accès. Il peut détecter les violations sur le tas, la pile et les variables globales. Actuellement, il ne prend en charge que les binaires en mode ARM-THUMB.

4.3.2 Architecture

YudSan est utilisé depuis le programme de harness pour l'émulation et le fuzzing. Comme le montre la figure 11, YudSan a six composants : Translator, Sanitizer, Function, Heap, Stack et Global. Translator est chargé de créer des objets qui représentent une instruction ARM-THUMB. Sanitizer manipule les métadonnées et vérifie l'accès à la mémoire en fonction de ces informations. Heap, Stack et Global maintiennent et mettent à jour les métadonnées

relatives respectivement aux mémoires de tas, de pile et globale. Heap est aussi une implémentation simple du tas linux dans Unicorn. Function est réservée au développement de vérifications sur les appels aux fonctions des bibliothèques comme memcpy. Je fournis également un script pour analyser le binaire afin de trouver des informations utiles comme :

1. Adresse de début et de fin de la fonction “main”.
2. Les adresses des appels aux fonctions des bibliothèques standard comme malloc, free, memcpy, printf…
3. Région et objets globaux.

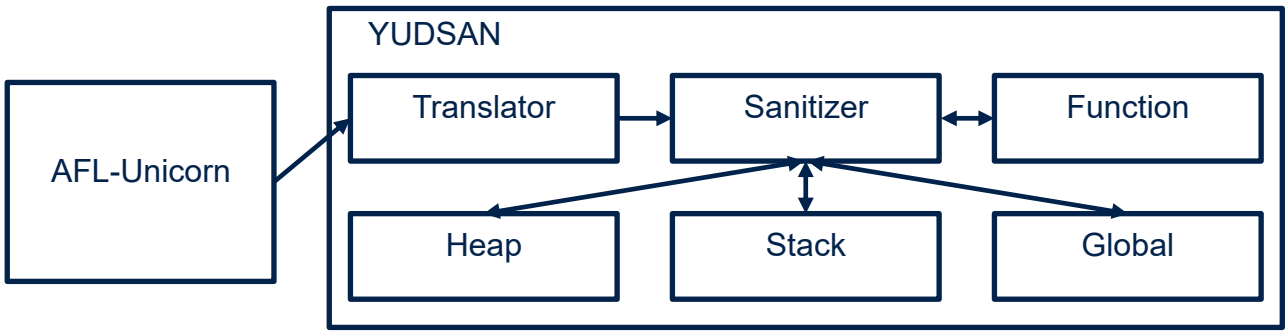

Figure 11: Architecture de YudSan

Translator

YudSan fonctionne en analysant les instructions assembleur. Dans ce cas, chaque instruction ARM-THUMB est représentée par un objet appelé Instruction (Imm, Rd, Rt, Rn, Rm, P, U, W, use_imm, shift_type, op)

1. Rd, Rt, Rn, Rm : les registres utilisés dans une instruction, la valeur varie de 0 à 15, plus de 15 indiquent que le registre n’est pas dans l’instruction. Normalement, Rt est le registre source, Rd est la destination, Rn est la base et Rm est le décalage.
2. Imm: valeur immédiate utilisée au lieu de Rm.
3. P : généralement utilisé dans les instructions STR et LDR pour indiquer si base ou base + offset est utilisé comme adresse.
4. U : généralement utilisé dans les instructions STR et LDR pour indiquer si base + offset ou base - offset est utilisé.
5. W : généralement utilisé dans les instructions STR et LDR pour indiquer si la valeur calculée est réécrite dans le registre de base.
6. use_imm : utilisé pour indiquer si la valeur immédiate ou le contenu de Rm est utilisé.
7. shift_type : utilisé pour indiquer l’opération de bit shift effectuée dans l’instruction, sa plage de valeurs va de 0 à 2 ; 0 pour aucun décalage, 1 pour un décalage à gauche, 2 pour un décalage à droite.
8. op : opération de l’instruction (LDR, STR, MOV, ADD...)

Exemple : LDR r0 [r1, r2 lsl 2]!

L’instruction ci-dessus chargera la mémoire à l’adresse appelée A dans r0, le signe ! indique que la valeur A sera écrite en retour dans r1. On calcule en décalant à gauche le contenu de r2 par 2 puis en ajoutant au contenu de r1. L’objet représente donc cette instruction :

1. Rd = 16 (non utilisé), Rt = 0, Rn = 1, Rm = 2

2. Imm = 2 (utiliser pour le shift)
3. P = 0
4. U = 1
5. W = 1
6. use_imm = 0
7. shift_type = 1
8. op = LDR

Comme Unicorn a aussi un processus de traduction d'instructions, j'ai choisi d'en utiliser les résultats pour éviter de devoir faire deux fois ce travail. Pour obtenir ce résultat dans Unicorn, j'ai fait quelques patchs. Dans Unicorn, les instructions sont également représentées par des objets. Mais à la différence de nos objets qui représentent toutes les opérations, Unicorn a beaucoup d'objets. J'ai donc créé une valeur intermédiaire qui permet aux objets Unicorn d'être traduits en objets YudSan avec la structure affichée dans la figure 12.

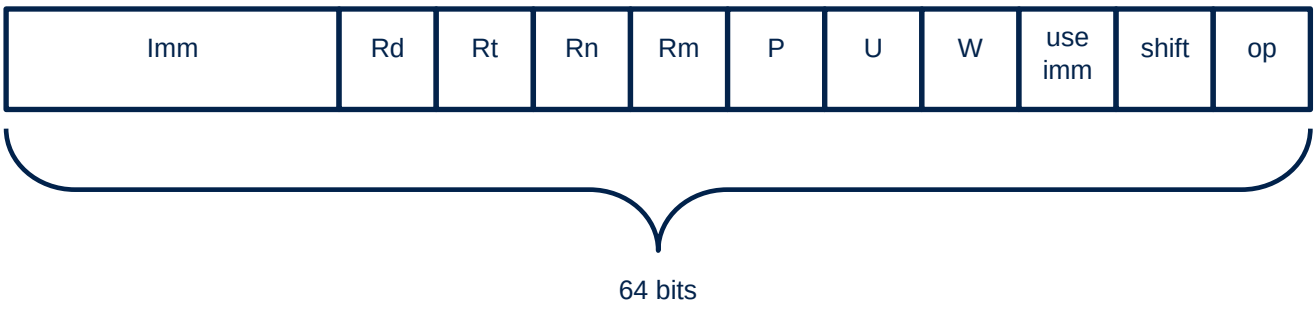

Figure 12 : Valeur de 64-bit représente les Instruction.

Lors de la traduction des instructions Unicorn, je calcule une valeur 64 bits qui stocke :

1. Imm dans ses premiers 32 bits.
2. Rd, Rt, Rn, Rm dans cet ordre dans les 20 bits suivants (5 bits chacun).
3. P, U, W, use_imm dans cet ordre dans les 4 bits suivants (1 bit chacun).
4. shift_type dans les 2 bits suivants.
5. op dans le reste.

Dans Unicorn, lorsqu'il traduit une instruction, s'il y a un hook_code, il instrumentera l'instruction traduite avec un appel à la fonction de rappel associée à ce hook_code. Par conception, cette fonction de rappel n'a que l'adresse de l'instruction et sa taille comme paramètres. J'ai modifié le hook_code pour ajouter un autre paramètre à la valeur de l'instruction. Puisque Unicorn génère l'appel au hook_code avant le processus de traduction, j'ai également patché la fonction de traduction pour traduire d'abord, puis générer l'appel au hook_code avec la valeur de l'instruction, et enfin générer le code traduit.

Heap

Comme le tas est une fonctionnalité de la bibliothèque standard, il n'existe pas dans Unicorn. Aussi, pour émuler et sanitizer les programmes qui utilisent le tas, j'ai implémenté notre propre tas dans Unicorn. Inspiré par le tas linux, le nôtre est structuré comme une liste chaînée des objets block_header comme affiché dans figure 13, chaque élément représentant un bloc sur le tas avec des attributs :

1. size: la taille.
2. free : si le bloc est libéré ou non.

3. next, last: bloc suivant et dernier bloc.
4. heap_addr: adresse du bloc dans la mémoire Unicorn.

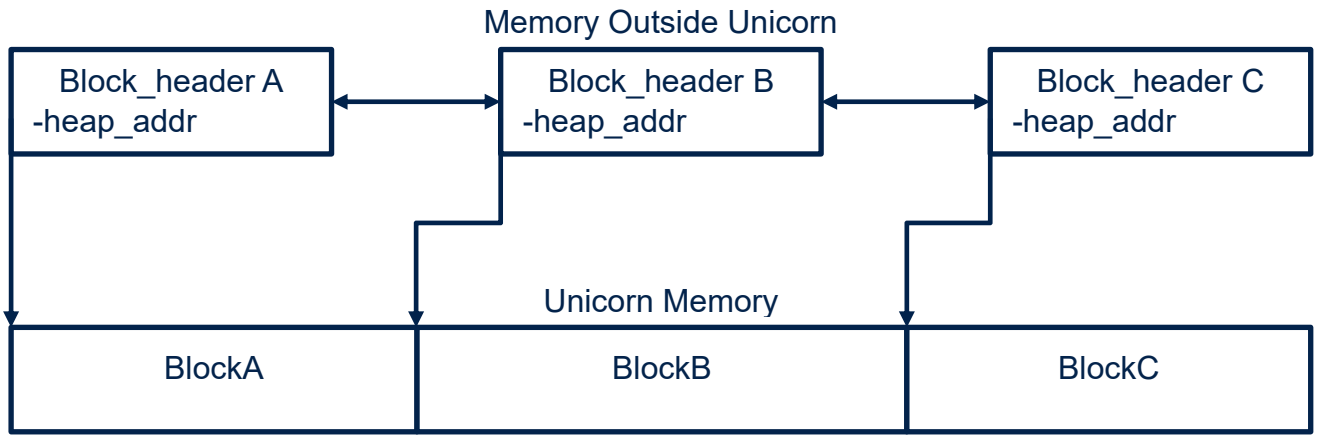

Figure 13 : Le tas de YudSan

Le tas de YudSan supporte malloc, calloc, realloc et free.

La shadow memory pour le tas est la liste chaînée elle-même. L'identité de chaque objet est le block_header.

Stack

Stack contient les métadonnées pour de la pile. Les métadonnées de la pile sont une shadow memory sous forme de tableau qui contient l'identité de chaque mot de 4 octets sur la pile. Comme dans les programmes en mode ARM-THUMB, les objets sur la pile ont un alignement de 4 octets, on peut supposer que tous les mots de 4 octets auront la même identité.

Global

Les métadonnées pour la mémoire globale sont assez similaires à celles de la pile. C'est aussi un shadow memory mais cette fois-ci chaque élément représente seulement 1 octet de mémoire globale. La raison en est que bien que les objets globaux aient aussi un alignement de 4 octets, il est plus facile de trouver la taille réelle des objets globaux par rapport aux objets de la pile. Ainsi, la shadow memory sera plus précise.

Function

Function contient les implémentations des fonctions de librairie standard qui sont reliées aux manipulations de mémoire. Pour le moment, YudSan ne supporte que memcpy. Memcpy est le plus utilisé par les développeurs et les compilateurs optimisent le code en remplaçant les boucles aves les memcpy.

Sanitizer

Sanitizer reçoit des informations sur l'instruction à exécuter. Il met ensuite à jour la shadow memory associée ou vérifie l'accès en fonction de l'instruction :

- ADD Rd, SP, offset : si Rd n'est pas SP, nous identifions un objet de la pile.
- SUB Rd, SP, offset : si Rd est SP, nous identifions un objet de la pile.
- LDR/STR : accès à la mémoire.
- LDMIA/POP : fin d'une fonction, la shadow memory de la pile est effacée.

- STRDB/PUSH : entrée d'une fonction, nous identifions l'adresse de retour.

4.3.3 Identification des objets

Heap

Identifier un objet de tas est assez facile puisque nous contrôlons la fonction malloc : nous connaissons donc la taille de l'objet et décidons de son adresse.

Stack

Les Objets

J'utilise une heuristique pour détecter les pointeurs sur la pile, où je considère toute opération de la forme "ADD Rd SP, offset" comme le calcul d'un pointeur vers un objet sur la pile. Bien que les compilateurs ne soient pas tenus d'utiliser cette opération, et qu'ils puissent utiliser d'autres astuces, il est très probable qu'elle soit utilisée car si un pointeur doit être transmis à une autre fonction ou accédé immédiatement, l'utilisation de sp+N est la façon la plus efficace de le calculer. Les limites de cet objet peuvent être raisonnablement calculées à partir du pointeur vers l'objet suivant identifié. En outre, l'opération de la forme "SUB SP, SP, offset" est utilisée pour allouer une grande mémoire sur la pile. Cela peut être un grand objet ou un groupe d'objets. Dans tous les cas, j'assigne une identité à cette grande zone et elle sera affinée par l'opération ci-dessus.

L'adresse de retour d'une fonction

L'adresse de retour est toujours empilée sur la pile avec toute opération PUSH ou STRDB (Store Multiple Decrement Before ; fonctionne comme PUSH) qui a LR (Link Register ; détient l'adresse à retourner lorsqu'un appel de sous-routine se termine) en tant que paramètre. LR est toujours stocké en premier, donc l'adresse de retour est à l'adresse SP + 4.

Global

Les objets globaux résident dans la section .data (initialisée) ou .bss (non initialisée) et ont également un alignement de 4 octets. Pour localiser des objets globaux, je localise d'abord la région globale. Les sections .data et .bss se concatènent en mémoire, ainsi la région globale est située entre l'adresse de départ de .data à l'adresse de fin du .bss. Cette information peut être obtenue via l'en-tête du programme. Ensuite, en parcourant la section .symtab, je peux trouver les objets globaux via leurs symboles.

4.3.4 Sanitizer check

YudSan considère que tous les accès à la mémoire qui ne sont pas dans une boucle sont de la forme :

LDR/STR Rt, [Rn, Offset]

Où Rn est l'adresse de base ou pointeur et Rn + Offset est l'adresse de destination. YudSan vérifie ensuite sur la shadow memory si l'adresse de base et l'adresse de destination ont la même identité.

Avec la perte d'information du processus de compilation, il est difficile de localiser tous les objets sur la pile. L'utilisation d'une heuristique est utile, mais elle n'est pas parfaite. Les adresses et les tailles trouvées sont toutes issues d'hypothèses, certaines erreurs détectées pourraient être de faux positifs. En outre, je veux me concentrer sur les programmes qui ont été compilés avec l'optimisation -Os puisque cette option est généralement utilisée dans les

systèmes embarqués. Les optimisations sont connues pour être l'ennemi du sanitizer. Pour obtenir de hautes performances, les compilateurs violent parfois l'accès à la mémoire dans un but précis. Ce comportement donne également de nombreux faux positifs.

Inspirés par MTSan, je divise les erreurs en deux catégories : l'erreur réelle et l'erreur potentielle. Seules les erreurs réelles entraînent des crashs. Cela permet de minimiser les faux positifs tout en donnant des indices au fuzzer sur les cas de test qui sont à investiguer.

Erreurs Réelles

1. Tas : Use After Free, Double Free, Memory leak, Overflow qui ne se produit pas dans une boucle.
2. Pile : écraser l'adresse de retour sur la pile.
3. En général : dépassement qui commence à partir de la région globale, du tas ou de la pile et finit dans une autre région.

Erreurs Potentielles

1. Dépassement qui se produit dans une boucle.
2. Dépassement dans la pile.
3. Dépassement dans la région globale.

4.3.5 Evaluation

Dataset

Pour cette évaluation, j'utilise Juliet Test Suite [15]. Juliet est une collection de cas de test en langage C/C++. Il contient des exemples représentant différents CWE (Common Weakness Enumeration). J'ai choisi quelques CWE qui sont liés à la corruption de mémoire :

1. CWE121: Stack Based Buffer Overflow
2. CWE122: Heap Based Buffer Overflow
3. CWE401: Memory Leak
4. CWE415: Double Free
5. CWE416: Use After Free

Il y a plusieurs programmes de test pour chaque CWE. Je retire les programmes qui utilisent des fonctions de bibliothèque ou des fonctionnalités non prises en charge par YudSan, notamment :

1. Les programmes C++ qui utilisent new", "delete", "map", "vector"
2. Les programmes qui utilisent des fonctions de bibliothèque standard liées à la mémoire autres que "memcpy", "malloc", "free", "calloc", "realloc"

Tous les programmes seront compilés en mode ARM-THUMB par "arm-linux-gnueabihf-gcc" avec l'optimisation des options "-Os". Chaque programme a deux versions : une bonne (Good) et une mauvaise (Bad). Les mauvais programmes contiennent une vulnérabilité alors que les bons n'en contiennent pas.

4.3.6 Résultat : Juliet Test Suite

Le tableau 4 montre les résultats des tests de la suite de tests Juliet en indiquant le nombre d'erreurs réelles et d'erreurs potentielles détectées dans les mauvaises et bonnes versions de chaque CWE choisi.

CWE		Erreurs Réelles	Erreurs Potentielles
CWE121: Stack Based Buffer Overflow	Bad	228/820 (27.8%)	153/820 (18.65%)
	Good	23/820 (0.028%)	143/820 (17.43%)
CWE122: Heap Based Buffer Overflow	Bad	387/684 (56.57%)	2/684 (0.002%)
	Good	13/684 (0.019%)	115/684 (16.81%)
CWE401 : Memory Leak	Bad	755/755 (100%)	0/755 (0%)
	Good	0/755 (0.01%)	0/755 (0%)
CWE415 : Double Free	Bad	210/210 (100%)	0/210 (0%)
	Good	0/210 (0%)	0/210 (0%)
CWE416 : Use After Free	Bad	84/128 (65.62%)	0/128 (0%)
	Good	0/128 (0%)	57/128 (44.53%)

Tableau 4 : Résultat de test sur Juliet

4.3.7 Observation

1. CWE416 :
    a. Good : Il y a un grand nombre d'erreurs potentielles en raison de l'utilisation de boucles, j'en discute plus en détail dans la limitation.
    b. Bad : Certains cas de faux négatifs sont de raison que le pointeur libéré est passé dans des fonctions de bibliothèque (généralement printf), je n'ai pas d'implémentations de ces fonctions et je ne peux pas supposer si ces fonctions utiliseront le pointeur pour accéder à la mémoire (erreurs) ou simplement pour effectuer des calculs (pas d'erreurs).
2. CWE415 et CWE401 :
    a. En général, YudSan fonctionne bien dans ces deux CWE.
3. CWE121 et CWE122:
    a. Good : Comme CWE416, il y a beaucoup d'erreurs potentielles and raison des boucles.
    b. Bad : Une partie de faux négatifs est de raison que le tampon vulnérable est considéré comme un code mort, il n'est jamais utilisé après le dépassement donc il est supprimé par le compilateur. Pour CWE121, étant donné que le dépassement entre les objets de la pile est une erreur potentielle, une autre partie de faux négatifs est sous forme des erreurs potentielles.

4.3.8 Limites

Voici les situations où il sera difficile pour YudSan de fonctionner correctement :

1. Accéder aux objets de la pile en utilisant SP comme adresse de base : Dans les cas où le programme utilise des opérations LDR/STR R [SP, offset] pour accéder à un objet quelconque de la pile, il y a de fortes chances que SP ne soit pas l'adresse de base de cet objet, rendant impossible de déterminer quel pointeur accède à cet objet.
2. Boucles : Dans les cas où nous itérons un tableau, l'accès à la mémoire se fait habituellement sous la forme LDR/STR Rt [Rn, Imm]! ou LDR/STR Rt, [Rn], Imm. Ces deux termes sont équivalents à Arr[++i] et Arr[i++] en langage C. Pour accéder à Arr[0] avec Arr[++i], i doit être -1, en assembleur il s'agira de LDR/STR Rt [Arr - 4, 4], car Arr est l'adresse de base du tableau, Arr - 4 et Arr auront des identités différentes. Ceci est également vrai pour l'autre cas avec le dernier élément du tableau. Pour minimiser les faux positifs, je marque ceux-ci comme des erreurs potentielles.

3. Accès aux objets globaux optimisé : Comme dans le premier cas, le compilateur optimise les accès aux objets globaux en utilisant toujours l'adresse de base du premier objet pour être l'adresse de base de tous les accès.
4. Stack overflows : les heuristiques implémentées ne permettent pas d'identifier avec certitude tous les objets de la pile. Par conséquent, je marque toutes les violations sur la pile (qui n'écrasent pas l'adresse de retour) comme des erreurs potentielles.
5. Instruction vectorielle : Pour une allocation d'objets vraiment importante, ARM peut utiliser des instructions vectorielles. YudSan ne les prend actuellement pas en charge.
6. Fonctions de bibliothèque standard liées à la mémoire : YudSan ne prend actuellement en charge que les memcpy.

5 PERSPECTIVES

Des améliorations peuvent être apportées à YudSan et à AFL-Unicorn :

1. Patcher AFL-Unicorn pour qu'il prenne en charge les erreurs potentielles de YudSan : Inspiré par la violation non critique dans MTSan, l'erreur potentielle est créée pour minimiser les bruits causés par une identification d'objet incorrecte, elle ne doit donc pas être traitée comme une véritable erreur dans AFL++. L'erreur potentielle peut donner des indications sur l'erreur réelle, elle doit donc être considérée comme un cas de test intéressant et être étudiée plus en détail.
2. Implémenter plus de fonctions de bibliothèque standard liées à la mémoire telles que memmove, memset, fgets.
3. Améliorer l'analyse des instructions : ce projet vise directement les programmes qui utilisent l'optimisation "-Os", je n'analyse donc qu'un nombre limité d'opérations. En ajoutant plus d'analyses, YudSan peut prendre en charge plusieurs options d'optimisation, améliorant ainsi l'efficacité. Il peut également vérifier si le programme exécute une boucle ou non, ce qui aide à réduire les faux positifs.
4. Optimiser AFL-Unicorn : Les propositions mentionnées dans la section 4.1.3 peuvent être testées pour améliorer les performances du fuzzing.

6 CONCLUSION

Ce stage consacré au fuzzing binaire et sanitizer a été une expérience enrichissante qui m'a permis d'apprendre des techniques de sécurité logicielle avancées. À travers ce projet, j'ai eu l'opportunité de comprendre les principes de fuzzing, émulation et sanitizer ; de comprendre des mécanismes internes d'AFL-Unicorn, ce qui m'a amené à proposer des optimisations, puis à déboguer, et enfin à développer YudSan, point clé de ce stage. En travaillant sur YudSan, j'ai également acquis de nombreuses connaissances sur le langage assembleur, en particulier sur l'architecture ARM en mode THUMB, afin d'analyser les instructions.

En résumé, YudSan est un identity-based sanitizer avec instrumentation dynamique binaire. Il cible les programmes ARM en mode THUMB, et compilés avec l'optimisation "-Os". Il est capable de détecter les erreurs dans le tas, la pile et les variables globales et il est compatible avec Unicorn Engine et AFL-Unicorn.

YudSan a démontré une capacité prometteuse à détecter des erreurs d'accès mémoire telles que les débordements de tampon et les fuites de mémoire. Cela s'est avéré crucial pour améliorer la robustesse des tests de fuzzing, en augmentant la précision des détections de bugs critiques.

Toutefois, ce travail n'a pas été exempt de défis. En raison de la perte d'information, l'identification des objets sur la pile était difficile et il a fallu élaborer une heuristique. De plus, l'équilibrage entre une détection fine des erreurs et la minimisation des faux positifs et les erreurs potentielles a été une pointe centrale du développement, nécessitant de nombreuses itérations et ajustements.

En conclusion, ce stage m'a permis de développer des compétences techniques solides en fuzzing binaire. Les outils et méthodes explorés, combinés aux défis surmontés, constituent des connaissances précieuses pour ma carrière future en cybersécurité. Les perspectives offertes par AFL-Unicorn et les sanitizers d'adresse montrent l'importance croissante de ces techniques dans le renforcement de la sécurité des logiciels, en particulier dans les environnements embarqués.

7 BIBLIOGRAPHIE

[1] «OSS-Fuzz,» [En ligne]. Available: https://github.com/google/oss-fuzz.

[2] «Liste de CVEs qui ont été découverts à l'aide de Fuzzing,» [En ligne]. Available: https://github.com/AFLplusplus/AFLplusplus/issues/286.

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

[4] «AFL++ GitHub,» [En ligne]. Available: https://github.com/AFLplusplus/AFLplusplus.

[5] M. Zalewski, «American Fuzzy Lop (AFL) homepage,» [En ligne]. Available: https://lcamtuf.coredump.cx/afl/.

[6] «american fuzzy lop GitHub,» [En ligne]. Available: https://github.com/google/AFL.

[7] «Unicorn Engine GitHub,» [En ligne]. Available: https://github.com/unicorn-engine/unicorn.

[8] «AFL-Unicorn,» [En ligne]. Available: https://github.com/AFLplusplus/unicornafl/tree/4b4fdab161c15529affcc1e785d779e318b882ab.

[9] Schilling, J., Wendler, A., Görz, P., Bars, N., Schloegel, M., & Holz, T, «A Binary-level Thread Sanitizer or Why Sanitizing on the Binary Level is Hard».

[10] Fioraldi, A., D'Elia, D. C., & Querzoni, L., «Fuzzing binaries for memory safety errors with QASan,» *2020 IEEE Secure Development (SecDev),* pp. 23-30, 2020.

[11] Salehi, M., Hughes, D., & Crispo, B, «{µSBS}: Static Binary Sanitization of Bare-metal Embedded Devices for Fault Observability,» *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020),* pp. 381-395, 2020.

[12] J. Craaijo, «IdSan: An identity-based memory sanitizer for fuzzing binaries,» 2020.

[13] Chen, X., Shi, Y., Jiang, Z., Li, Y., Wang, R., Duan, H., ... & Zhang, C, «{MTSan}: A Feasible and Practical Memory Sanitizer for Fuzzing {COTS} Binaries,» *32nd USENIX Security Symposium (USENIX Security 23),* pp. 841-858, 2023.

[14] «Unicorn Bug,» [En ligne]. Available: https://github.com/unicorn-engine/unicorn/issues/1865.

[15] «Juliet Test Suite for C/C++,» [En ligne]. Available: https://github.com/arichardson/juliet-test-suite-c/tree/master.