05 — Testing & Fuzzing (Trail of Bits)

Handbook de testing et fuzzing (libfuzzer, AFL++, ASan, coverage, etc.).

- □□ Index — 05 — Testing & Fuzzing (Trail of Bits)
- aflpp
- coverage-analysis
- fuzzing-dictionary
- wycheproof
- libfuzzer
- ossfuzz
- ruzzy
- testing-handbook-generator
- atheris
- cargo-fuzz
- address-sanitizer
- libafl
- fuzzing-obstacles
- constant-time-testing
- harness-writing
- property-based-testing

□□ Index — 05 — Testing & Fuzzing (Trail of Bits)

05 — Testing & Fuzzing (Trail of Bits)

> Index des skills regroupés dans ce livre. Clique sur un skill pour ouvrir sa fiche.

16 skills documentés.

Skill	En une phrase
address-sanitizer	AddressSanitizer (ASan) est un "détecteur de fumée" qu'on greffe au code C/C++ pour repérer en temps réel les bugs mémoire…
aflpp	AFL++ est un "fuzzer" (machine à tester) qui bombarde un programme C/C++ avec des milliers d'entrées aléatoires en parallèle sur…
atheris	Atheris est un fuzzer pour Python : il bombarde ton code Python avec des entrées aléatoires pour faire surgir les exceptions non…
cargo-fuzz	cargo-fuzz est l'outil officiel pour fuzzer un projet Rust : il bombarde tes fonctions Rust avec des entrées aléatoires pour…
constant-time-testing	Le constant-time testing vérifie qu'une opération cryptographique met toujours exactement le même temps à s'exécuter — sinon un…
coverage-analysis	L'analyse de couverture mesure quelles lignes de ton code ont vraiment été exécutées par les tests/fuzzers — pour repérer les…
fuzzing-dictionary	Un dictionnaire de fuzzing, c'est une liste de "mots magiques" qu'on donne au fuzzer pour qu'il sache de quoi il parle (mots-clés…

Skill	En une phrase
fuzzing-obstacles	Recueil de techniques pour contourner les "blocages" qui empêchent un fuzzer d'avancer (checksums, horloges aléatoires,...
harness-writing	Le harness (ou "harnais") c'est la petite fonction d'entrée qu'on écrit pour brancher un fuzzer à son code : un mauvais harness =...
libafl	LibAFL est une "boîte à outils" en Rust pour construire ton propre fuzzer sur mesure quand les fuzzers tout faits (libFuzzer,...
libfuzzer	libFuzzer est un "fuzzer" intégré au compilateur LLVM qui bombarde une fonction C/C++ avec des milliers d'entrées aléatoires pour...
ossfuzz	OSS-Fuzz est un service gratuit de Google qui fait tourner du fuzzing 24h/24 sur les projets open source critiques pour trouver...
property-based-testing	Le property-based testing teste des **propriétés universelles** ("encoder puis décoder doit redonner la valeur d'origine") au...
ruzzy	Ruzzy est un fuzzer pour Ruby (créé par Trail of Bits) : il bombarde ton code Ruby et tes extensions C de Ruby avec des entrées...
testing-handbook-generator	Méta-skill qui lit le "Testing Handbook" de Trail of Bits (appsec.guide) et génère automatiquement de nouvelles skills Claude...
wycheproof	Wycheproof est une énorme collection de "pièges crypto" maintenus par Google : des entrées tordues, déjà connues pour casser les...

aflpp

aflpp

> *Catalogue généré le 2026-05-11*

En une phrase

AFL++ est un "fuzzer" (machine à tester) qui bombarde un programme C/C++ avec des milliers d'entrées aléatoires en parallèle sur plusieurs cœurs CPU pour faire sortir tous les bugs et plantages cachés.

Quand l'utiliser

- Tester intensivement un projet écrit en C ou C++ pour traquer les bugs mémoire.
- Faire tourner une grosse campagne de fuzzing sur plusieurs cœurs (très rapide).
- Pousser un test plus loin quand un fuzzer plus simple (libFuzzer) a déjà épuisé ses idées.
- Auditer un logiciel mature avant un release important.
- Auditer une bibliothèque utilisée par beaucoup de monde (sécurité critique).

Comment l'invoquer

- **Slash command** : `/aflpp`
- **Phrases déclencheurs (texte)** : "fuzz this C/C++ project", "multi-core fuzzing", "AFL++"
- **Auto-invocation** : Sur demande explicite

Description détaillée

AFL++ (American Fuzzy Lop, version "plus plus") est un fuzzer de référence pour le code C et C++. Un fuzzer, c'est un outil qui prend ta fonction et lui balance des millions d'entrées bizarres et aléatoires jusqu'à ce qu'elle plante ou se comporte mal. C'est un peu comme tester une porte en lui donnant des coups dans tous les sens pour voir si elle craque.

Son gros atout par rapport à ses concurrents : il sait fuzzer sur plusieurs cœurs CPU en parallèle de façon stable. Du coup, il est idéal pour des grosses campagnes (laisser tourner une nuit ou un week-end sur 8 cœurs). Il est plus complexe à installer que libFuzzer (il faut Clang ou GCC, parfois Docker), mais il offre des "mutations" plus variées et un mode CMPLOG qui aide à franchir les vérifications difficiles dans le code.

Quand tu auditerais ce qu'il a trouvé, AFL++ te livre des "crashes" (entrées qui ont fait planter), des "hangs" (entrées qui ont fait freezer) et des stats de couverture (quelles parties du code ont été visitées). On le combine souvent avec AddressSanitizer pour détecter les corruptions mémoire silencieuses.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `aflpp`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/aflpp/SKILL.md`

coverage-analysis

coverage-analysis

> *Catalogue généré le 2026-05-11*

En une phrase

L'analyse de couverture mesure quelles lignes de ton code ont vraiment été exécutées par les tests/fuzzers — pour repérer les "zones aveugles" que personne ne teste.

Quand l'utiliser

- Évaluer si une campagne de fuzzing visite vraiment le code intéressant (ou bloque ailleurs).
- Identifier les parties d'un programme jamais touchées par les tests.
- Comparer deux versions d'un harness pour voir laquelle explore plus de code.
- Détecter les "fuzzing blockers" (un checksum, une validation qui bloque tout).
- Prioriser les améliorations de tests (sur les zones non couvertes).

Comment l'invoquer

- **Slash command** : `/coverage-analysis`
- **Phrases déclencheurs (texte)** : "code coverage", "fuzzing coverage", "harness effectiveness"
- **Auto-invocation** : Sur demande explicite

Description détaillée

La couverture de code, c'est l'art de répondre à la question : "quelles lignes/branches de mon code sont vraiment exécutées quand je lance mes tests ?". On instrumente le code à la compilation (avec `-fprofile-instr-generate` par exemple), on le lance avec ses tests/fuzzers, et on en sort un rapport qui te montre en vert les lignes visitées et en rouge celles jamais touchées.

En fuzzing, c'est doublement utile. D'abord pour évaluer ton harness : si après plusieurs heures de fuzzing tu n'as couvert que 30 % du code de ta cible, c'est probablement que ton harness est mal pensé ou qu'un obstacle (checksum, validation) bloque tout. Ensuite pour suivre les progrès : tu modifies le harness, tu rajoutes un dictionnaire, tu changes de fuzzer, et la couverture te dit immédiatement si ça aide ou pas.

La couverture n'est pas un indicateur parfait de la qualité du fuzzing (un mauvais fuzzer peut atteindre une bonne couverture sans trouver de bugs), mais c'est le meilleur indicateur disponible et facile à mesurer. Les outils classiques : `llvm-cov` (pour le code compilé avec Clang), `gcov` (pour GCC), `lcov` pour générer des rapports HTML, et des plateformes comme Codecov pour suivre la couverture dans le temps. À utiliser systématiquement quand on prend le fuzzing au sérieux.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `coverage-analysis`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/coverage-analysis/SKILL.md`

fuzzing-dictionary

fuzzing-dictionary

> *Catalogue généré le 2026-05-11*

En une phrase

Un dictionnaire de fuzzing, c'est une liste de "mots magiques" qu'on donne au fuzzer pour qu'il sache de quoi il parle (mots-clés HTTP, balises XML, octets de header PNG…) et trouve des bugs plus vite.

Quand l'utiliser

- Fuzzer un parseur de format binaire (PNG, ZIP, PDF) qui a des "magic bytes" en en-tête.
- Fuzzer un parseur de format texte (JSON, XML, YAML, configs).
- Fuzzer un protocole réseau (HTTP, DNS, protocole maison).
- Quand ton fuzzer plafonne sans trouver de nouveaux chemins (il bloque sur une validation).
- Pour rendre n'importe quelle campagne de fuzzing plus efficace (format-aware fuzzing).

Comment l'invoquer

- **Slash command** : `/fuzzing-dictionary`
- **Phrases déclencheurs (texte)** : "fuzzing dictionary", "format-aware fuzzing", "magic bytes"
- **Auto-invocation** : Sur demande explicite

Description détaillée

Quand un fuzzer génère des entrées au hasard, il a très peu de chances de produire spontanément la string `"Content-Type"` ou les 4 octets magiques `89 50 4E 47` qui ouvrent un fichier PNG. Du coup, il reste bloqué dans les premières lignes du parseur ("le format est invalide, retour") sans jamais explorer le cœur du code.

Un dictionnaire de fuzzing résout ce problème. C'est un simple fichier texte avec des "tokens" entre guillemets, par exemple `"GET "`, `"Content-Length:"`, `kw="\xFF\xD8\xFF\xE0"` (entête JPEG). Tu le donnes au fuzzer avec l'option `-dict=`, et il va insérer ces morceaux dans ses mutations. D'un coup, ses entrées ressemblent vaguement à du vrai protocole/format, et il franchit la couche de validation pour aller explorer en profondeur.

Le format est universel : un même dictionnaire marche avec libFuzzer, AFL++ et cargo-fuzz. Tu peux trouver des dictionnaires prêts à l'emploi sur le repo AFL++ (HTTP, JS, XML, etc.) ou en construire un toi-même en extrayant les chaînes de la doc du format. C'est une des optimisations à plus fort retour sur investissement pour booster une campagne de fuzzing — souvent x10 ou x100 sur la vitesse de découverte de bugs.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `fuzzing-dictionary`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/fuzzing-dictionary/SKILL.md`

wycheproof

wycheproof

> *Catalogue généré le 2026-05-11*

En une phrase

Wycheproof est une énorme collection de "pièges crypto" maintenus par Google : des entrées tordues, déjà connues pour casser les implémentations bancales, qu'on jette à ta lib crypto pour voir si elle tient le coup.

Quand l'utiliser

- Tester une implémentation maison ou tierce d'un algo crypto (RSA, AES, ECDSA, HMAC…).
- Auditer une bibliothèque crypto avant intégration dans un produit.
- Vérifier qu'une lib supporte correctement les courbes elliptiques (curves).
- Tester un wallet ou un client blockchain qui valide des signatures.
- Vérifier que ta lib rejette bien les cas vicieux (zéro, courbes invalides, paddings cassés…).

Comment l'invoquer

- **Slash command** : `/wycheproof`
- **Phrases déclencheurs (texte)** : "crypto test vectors", "Wycheproof tests", "cryptographic edge cases"
- **Auto-invocation** : Sur demande explicite

Description détaillée

Wycheproof, c'est un trésor pour qui audite de la crypto. C'est une collection massive de "test vectors" — des couples entrée/sortie connus qui correspondent à des cas d'attaque réels documentés au fil des années. Originalement développé par Google, c'est maintenant un projet communautaire où chercheurs et industriels contribuent leurs trouvailles.

Pourquoi c'est précieux ? Implémenter de la crypto correctement, c'est terriblement dur. Un détail oublié (un padding mal géré, un point sur une courbe invalide accepté, un IV réutilisé…) suffit à briser toute la sécurité. Wycheproof contient des entrées qui ressemblent à de la crypto valide mais qui exploitent des bugs subtils. Si ta lib accepte une signature falsifiée, déchiffre quelque chose qu'elle devrait refuser, ou plante sur un input border-line — Wycheproof le révèle.

Concrètement, tu télécharges les fichiers JSON de test vectors correspondant à ton algo (par exemple "ECDSA secp256k1"), tu écris un petit harness qui boucle dessus et applique ta lib, et tu compares le résultat attendu (`valid`, `invalid`, `acceptable`) avec ce que ta lib renvoie. Tout écart = bug potentiel. C'est devenu un standard de validation crypto, notamment dans l'écosystème blockchain où les bugs de validation de signature peuvent coûter des millions.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `wycheproof`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/wycheproof/SKILL.md`

libfuzzer

libfuzzer

> *Catalogue généré le 2026-05-11*

En une phrase

libFuzzer est un "fuzzer" intégré au compilateur LLVM qui bombarde une fonction C/C++ avec des milliers d'entrées aléatoires pour trouver les bugs et plantages cachés — c'est l'outil le plus simple pour démarrer.

Quand l'utiliser

- Tester une fonction C ou C++ avec un setup minimal et rapide.
- Démarrer du fuzzing sans s'embêter avec des configurations complexes.
- Fuzzer un projet compilé avec Clang.
- Quand un seul cœur CPU suffit (pas de campagne géante).
- Prototyper avant de basculer plus tard sur AFL++ si besoin.

Comment l'invoquer

- **Slash command** : `/libfuzzer`
- **Phrases déclencheurs (texte)** : "fuzz C/C++ with libFuzzer", "quick fuzzing setup", "LLVM fuzzer"
- **Auto-invocation** : Sur demande explicite

Description détaillée

libFuzzer est le fuzzer "starter pack" pour le C et C++. Un fuzzer, c'est un robot qui essaie de casser ton programme en lui passant des entrées tordues, aléatoires, jamais prévues. Le but : voir s'il plante, accède à de la mémoire interdite ou se met dans un état bizarre. libFuzzer est fait par les équipes LLVM (les mêmes qui font le compilateur Clang), donc c'est super intégré au pipeline de compilation moderne.

Tu écris juste une petite fonction appelée `LLVMFuzzerTestOneInput` qui reçoit des octets aléatoires, tu compiles avec `-fsanitize=fuzzer` et c'est parti. C'est le plus rapide à mettre en route, mais il a deux limitations : il tourne sur un seul cœur CPU et il est en mode "maintenance" depuis fin 2022 (pas de nouvelles features).

Bonne nouvelle : les harnesses (les petits programmes d'entrée écrits pour libFuzzer) sont compatibles avec AFL++. Donc tu commences avec libFuzzer pour aller vite, et si tu veux scaler sur plusieurs cœurs plus tard, tu réutilises le même code avec AFL++. C'est un excellent premier pas avant les outils plus avancés.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `libfuzzer`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/libfuzzer/SKILL.md`

ossfuzz

ossfuzz

> *Catalogue généré le 2026-05-11*

En une phrase

OSS-Fuzz est un service gratuit de Google qui fait tourner du fuzzing 24h/24 sur les projets open source critiques pour trouver des bugs avant les hackers.

Quand l'utiliser

- Inscrire un projet open source important dans le programme de fuzzing continu de Google.
- Mettre en place une infrastructure de fuzzing continu pour ton équipe.
- Quand un projet est utilisé par beaucoup de monde et mérite un audit continu (curl, OpenSSL, etc.).
- Pour automatiser le fuzzing dans un pipeline CI/CD.
- Pour publier des coverage reports automatiques.

Comment l'invoquer

- **Slash command** : `/ossfuzz`
- **Phrases déclencheurs (texte)** : "continuous fuzzing", "OSS-Fuzz enrollment", "Google fuzzing service"
- **Auto-invocation** : Sur demande explicite

Description détaillée

OSS-Fuzz est un service open source de Google qui fournit gratuitement une infrastructure massive (des milliers de cœurs CPU) pour faire tourner du fuzzing en continu sur les projets open source qui acceptent de s'inscrire. L'idée : les projets critiques pour internet (curl, OpenSSL, SQLite, Linux kernel…) tournent 24h/24 sur les serveurs Google, et dès qu'un bug est trouvé, Google notifie les mainteneurs en privé pour corriger avant que ce soit public.

Pour rejoindre OSS-Fuzz, tu dois préparer trois fichiers : un `project.yaml` (metadata du projet), un `Dockerfile` (l'image avec les dépendances de build) et un `build.sh` (le script qui compile les fuzzers). Google fait passer tes harnesses à travers libFuzzer, AFL++, Honggfuzz et les sanitizers (ASan, UBSan, MSan). Si quelque chose crashe, tu reçois un rapport détaillé.

Les projets acceptés sont évalués selon leur "criticality score" (importance pour l'écosystème). Tu peux aussi héberger ton propre instance d'OSS-Fuzz pour tes projets privés — le code de base est open source. C'est devenu LE standard de l'industrie pour le fuzzing continu sérieux.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `ossfuzz`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/ossfuzz/SKILL.md`

ruzzy

ruzzy

> *Catalogue généré le 2026-05-11*

En une phrase

Ruzzy est un fuzzer pour Ruby (créé par Trail of Bits) : il bombarde ton code Ruby et tes extensions C de Ruby avec des entrées aléatoires pour traquer les bugs.

Quand l'utiliser

- Fuzzer une application ou bibliothèque Ruby.
- Tester un gem Ruby qui contient des extensions natives (C/C++).
- Trouver des corruptions mémoire dans des gems avec du code natif.
- Quand tu veux du fuzzing guidé par couverture pour Ruby.
- Tester un parseur Ruby ou une fonction de désérialisation.

Comment l'invoquer

- **Slash command** : `/ruzzy`
- **Phrases déclencheurs (texte)** : "fuzz Ruby code", "Ruzzy fuzzer", "Ruby C extensions"
- **Auto-invocation** : Sur demande explicite

Description détaillée

Ruzzy est le fuzzer Ruby développé par Trail of Bits (une boîte de sécurité reconnue). À ce jour, c'est le seul fuzzer "production-ready" guidé par couverture pour Ruby. Comme Atheris (pour

Python), il s'appuie sur libFuzzer en moteur, mais sait analyser quelles parties du code Ruby ont été exécutées pour guider intelligemment ses mutations.

Pourquoi c'est utile ? L'écosystème Ruby contient beaucoup de "gems" (bibliothèques) qui ont une partie écrite en C natif pour la performance — par exemple Nokogiri pour le XML, ou les parseurs JSON rapides. Ces parties natives peuvent contenir des bugs mémoire (buffer overflow, use-after-free) que Ruby tout seul ne peut pas détecter. Ruzzy + AddressSanitizer = la combinaison idéale pour traquer ces bugs.

Tu écris un petit harness Ruby qui prend des bytes aléatoires en entrée et appelle ta fonction à tester. Ruzzy lance des milliers d'itérations, et garde précieusement chaque entrée qui fait planter quelque chose. C'est l'outil parfait pour les mainteneurs de gems et les équipes Rails qui veulent durcir leurs APIs.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `ruzzy`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/ruzzy/SKILL.md`

testing-handbook-generator

testing-handbook-generator

> *Catalogue généré le 2026-05-11*

En une phrase

Méta-skill qui lit le "Testing Handbook" de Trail of Bits (appsec.guide) et génère automatiquement de nouvelles skills Claude Code à partir de son contenu — un assistant pour bâtir ton propre catalogue de skills de sécurité.

Quand l'utiliser

- Créer une nouvelle skill basée sur un chapitre du Testing Handbook de Trail of Bits.
- Faire un refresh en masse de toutes les skills générées (mise à jour avec la nouvelle version du Handbook).
- Convertir un outil ou une technique de testing en skill exploitable par Claude.
- Quand quelqu'un mentionne "testing handbook" ou "appsec.guide".
- Pour produire en masse des fiches skills cohérentes à partir d'une source de vérité.

Comment l'invoquer

- **Slash command** : `/testing-handbook-generator`
- **Phrases déclencheurs (texte)** : "generate handbook skill", "appsec.guide", "testing handbook skills"
- **Auto-invocation** : Sur demande explicite

Description détaillée

Cette skill est un "méta-outil" : son rôle n'est pas de tester du code, mais de **générer** d'autres skills. Trail of Bits maintient un excellent ouvrage en ligne, le Testing Handbook (publié sur appsec.guide), qui couvre tous les outils et techniques de testing sécurité (fuzzers, sanitizers, analyse de couverture, audits crypto, etc.). Ce générateur lit ce handbook chapitre par chapitre et produit, pour chaque sujet, un fichier `SKILL.md` formaté que Claude Code peut ensuite invoquer.

Concrètement, tu pointes la skill vers le dépôt du handbook (`./testing-handbook`, `~/testing-handbook`, ou en clonant le repo `github.com/trailofbits/testing-handbook`), et tu lui demandes de générer ou rafraîchir une skill spécifique. Elle parse le markdown du handbook, extrait les sections pertinentes (when to use, quick start, examples) et produit une skill prête à l'emploi, conforme au format attendu.

C'est l'outil que Trail of Bits utilise en interne pour maintenir leur collection de skills (toutes celles que tu vois dans ce catalogue : libfuzzer, aflpp, address-sanitizer, etc.) en synchro avec leur documentation officielle. Tu peux l'utiliser pour personnaliser le catalogue à ton contexte ou pour ajouter de nouvelles skills si Trail of Bits publie de nouveaux chapitres. Très utile aussi si tu veux générer un catalogue similaire pour ta propre documentation interne.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `testing-handbook-generator`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/testing-handbook-generator/SKILL.md`

atheris

atheris

> *Catalogue généré le 2026-05-11*

En une phrase

Atheris est un fuzzer pour Python : il bombarde ton code Python avec des entrées aléatoires pour faire surgir les exceptions non gérées et les bugs cachés — c'est du fuzzing pour les pythonistes.

Quand l'utiliser

- Tester un parseur Python (JSON, XML, configs maison).
- Tester une bibliothèque Python qui traite des données utilisateur (souvent vulnérables).
- Fuzzer une extension C de Python (la partie écrite en C pour la perf).
- Trouver des plantages "AttributeError", "KeyError" non prévus dans un script.
- Tester un projet Python critique avant déploiement.

Comment l'invoquer

- **Slash command** : `/atheris`
- **Phrases déclencheurs (texte)** : "fuzz Python code", "Atheris fuzzer", "Python C extensions"
- **Auto-invocation** : Sur demande explicite

Description détaillée

Atheris est le fuzzer fait par Google pour le langage Python. Il s'appuie sur libFuzzer (le moteur de LLVM) mais sait parler Python : il guide ses mutations en fonction de la couverture de code Python (quelles lignes ont été exécutées) et pas seulement la couverture C.

Concrètement, tu écris une petite fonction `TestOneInput` qui reçoit des bytes aléatoires, tu les convertis dans ce que ta cible attend (un dict, une string, un objet…) et tu appelles ta fonction. Atheris boucle en générant des millions d'entrées en quelques minutes, et si une seule fait planter ton code (exception non rattrapée, segfault dans une extension C…), il te garde l'entrée fautive pour rejouer le bug.

Atheris brille particulièrement pour les extensions C de Python (les modules écrits en C/C++ pour la rapidité). Ces extensions peuvent avoir des bugs mémoire que Python pur n'a pas. Atheris peut être combiné avec AddressSanitizer pour détecter ces corruptions mémoire. Pour le Python pur, c'est aussi très utile pour traquer les exceptions oubliées et les cas limites jamais testés.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `atheris`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/atheris/SKILL.md`

cargo-fuzz

cargo-fuzz

> *Catalogue généré le 2026-05-11*

En une phrase

cargo-fuzz est l'outil officiel pour fuzzer un projet Rust : il bombarde tes fonctions Rust avec des entrées aléatoires pour trouver les bugs cachés, en une seule commande Cargo.

Quand l'utiliser

- Fuzzer un projet Rust qui utilise Cargo (l'outil de build standard).
- Tester un crate Rust avant publication sur crates.io.
- Tester du code Rust qui contient des blocs `unsafe` (sensibles à la mémoire).
- Tester un parseur Rust ou une fonction de désérialisation.
- Quand tu veux activer un sanitizer (AddressSanitizer) facilement avec Rust.

Comment l'invoquer

- **Slash command** : `/cargo-fuzz`
- **Phrases déclencheurs (texte)** : "fuzz Rust project", "cargo fuzz target", "Rust fuzzing"
- **Auto-invocation** : Sur demande explicite

Description détaillée

cargo-fuzz, c'est le standard de fait pour fuzzer du code Rust quand tu utilises Cargo (le gestionnaire de paquets et de build de Rust). Il s'utilise comme une sous-commande Cargo : `cargo`

`fuzz init`, `cargo fuzz run mon_test`, et voilà. Sous le capot, il utilise libFuzzer comme moteur, mais tout est branché automatiquement avec les bons flags de compilation pour Rust.

Rust est censé être un langage "memory safe" par défaut, donc tu pourrais te dire "à quoi bon fuzzer ?". Réponse : Rust empêche les bugs mémoire classiques, mais il reste plein d'autres bugs possibles. Tes parseurs peuvent paniquer sur une entrée tordue, ta logique de validation peut avoir des trous, et si tu utilises des blocs `unsafe`, là toutes les protections sautent. cargo-fuzz t'aide à trouver tout ça.

Tu écris un petit fichier `fuzz_target!` qui reçoit des bytes aléatoires, tu lances la commande, et l'outil tourne en boucle en générant des entrées de plus en plus exotiques jusqu'à trouver un crash. Le support des sanitizers (comme AddressSanitizer) est intégré, ce qui est très utile pour le code `unsafe`.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `cargo-fuzz`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/cargo-fuzz/SKILL.md`

address-sanitizer

address-sanitizer

> *Catalogue généré le 2026-05-11*

En une phrase

AddressSanitizer (ASan) est un "détecteur de fumée" qu'on greffe au code C/C++ pour repérer en temps réel les bugs mémoire (buffer overflow, use-after-free) que l'œil ne voit jamais.

Quand l'utiliser

- Pendant une campagne de fuzzing C/C++ (combo classique fuzzer + ASan).
- Pour tester du code Rust qui utilise `unsafe`.
- Pour traquer un bug mémoire suspect (corruption silencieuse, plantage aléatoire).
- Pour auditer une bibliothèque qui manipule beaucoup de pointeurs.
- Avant un release important d'un projet C ou C++.

Comment l'invoquer

- **Slash command** : `/address-sanitizer`
- **Phrases déclencheurs (texte)** : "ASan", "memory error detection", "AddressSanitizer setup"
- **Auto-invocation** : Sur demande explicite

Description détaillée

AddressSanitizer (ASan pour les intimes) est un outil de "sanitisation" : il instrumente ton code à la compilation pour ajouter des vérifications partout où ton programme touche à la mémoire. Du coup, dès qu'il détecte un accès illégal (lire un octet hors d'un tableau, utiliser un pointeur déjà libéré, double free...), il arrête le programme et te dit exactement où ça s'est passé, avec une stack trace propre.

C'est devenu le standard de l'industrie pour le fuzzing C/C++. Pourquoi ? Parce que beaucoup de bugs mémoire ne font pas planter le programme immédiatement — ils corrompent silencieusement la mémoire et causent des bugs aléatoires plus tard. Sans ASan, ton fuzzer pourrait passer à côté. Avec ASan, chaque accès interdit est attrapé sur le fait.

Le coût : ASan ralentit l'exécution de 2 à 4 fois et réserve 20TB de mémoire virtuelle (oui, terabytes — mais c'est virtuel, pas vraiment consommé). Il faut compiler avec `-fsanitize=address`. ASan détecte les classiques : buffer overflow (lecture/écriture hors zone), use-after-free (utilisation d'un pointeur libéré), double-free, memory leaks. À combiner avec UndefinedBehaviorSanitizer pour une couverture maximale.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `address-sanitizer`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/address-sanitizer/SKILL.md`

libafl

libafl

> *Catalogue généré le 2026-05-11*

En une phrase

LibAFL est une "boîte à outils" en Rust pour construire ton propre fuzzer sur mesure quand les fuzzers tout faits (libFuzzer, AFL++) ne suffisent plus pour ton cas tordu.

Quand l'utiliser

- Construire un fuzzer personnalisé avec ses propres stratégies de mutation.
- Fuzzer une cible exotique (architecture rare, format de données très spécifique).
- Faire de la recherche académique en fuzzing (nouvelles techniques).
- Avoir le contrôle fin sur chaque composant du fuzzing (feedback, scheduler...).
- Remplacer libFuzzer en gardant le même code mais avec plus de puissance.

Comment l'invoquer

- **Slash command** : `/libafl`
- **Phrases déclencheurs (texte)** : "custom fuzzer", "advanced fuzzing", "LibAFL research"
- **Auto-invocation** : Sur demande explicite

Description détaillée

LibAFL n'est pas un fuzzer "clé en main" comme libFuzzer ou AFL++. C'est plutôt une bibliothèque Rust modulaire qui te donne tous les bouts d'un fuzzer (l'observer de couverture, le mutateur, le

scheduler, le feedback...) et te laisse les assembler comme un Lego. Tu as donc une liberté quasi totale sur le comportement de ta machine à tester.

C'est l'outil pour les cas avancés. Par exemple : tu veux tester un firmware embarqué sur une puce inhabituelle, ou tu veux inventer une nouvelle stratégie de mutation qui n'existe nulle part ailleurs. Avec libFuzzer ou AFL++, tu serais coincé. Avec LibAFL, tu codes ce qu'il te faut. La contrepartie : la courbe d'apprentissage est raide (il faut connaître Rust et comprendre comment marche un fuzzer en interne).

Bon plan : LibAFL peut servir de "drop-in replacement" pour libFuzzer. Tu gardes ton harness existant, mais tu profites des features modernes (multi-cœur, mutations avancées) sans tout réécrire. C'est l'outil que les chercheurs en sécurité utilisent pour leurs papiers académiques et les pentesters pour leurs cibles les plus difficiles.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `libafl`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/libafl/SKILL.md`

fuzzing-obstacles

fuzzing-obstacles

> *Catalogue généré le 2026-05-11*

En une phrase

Recueil de techniques pour contourner les "blocages" qui empêchent un fuzzer d'avancer (checksums, horloges aléatoires, validations trop strictes) — en patchant temporairement le code pour qu'il devienne "fuzzable".

Quand l'utiliser

- Quand ton fuzzer stagne et ne trouve plus rien (la couverture ne grimpe plus).
- Quand ton code vérifie un checksum (CRC, SHA) et rejette tout en entrée.
- Quand ton code utilise `time()` ou `random()` (résultat différent à chaque run = fuzzer perdu).
- Quand une validation très stricte bloque toutes les entrées du fuzzer.
- Quand ton code fait des requêtes réseau ou écrit dans une vraie base de données.

Comment l'invoquer

- **Slash command** : `/fuzzing-obstacles`
- **Phrases déclencheurs (texte)** : "fuzzing blocker", "checksum bypass for fuzzing", "anti-fuzzing patterns"
- **Auto-invocation** : Sur demande explicite

Description détaillée

Plein de programmes contiennent des patterns "anti-fuzzing" sans le faire exprès. Par exemple, un format de fichier vérifie son checksum avant traitement : le fuzzer doit deviner la bonne valeur du checksum pour chaque entrée mutée — astronomiquement improbable. Résultat : 99,99 % des entrées sont rejetées immédiatement et le fuzzer n'explore jamais le vrai code.

Autres obstacles classiques : du code qui dépend de l'horloge système (`time()` renvoie une valeur différente à chaque exécution, donc le fuzzer perd le sens du "même input = même résultat"), des générateurs aléatoires non-déterministes, des appels réseau, ou des validations cryptographiques très strictes. Tous ces patterns "cassent" la boucle de feedback du fuzzer.

La solution proposée par cette skill : la **compilation conditionnelle**. Tu ajoutes des `#ifdef FUZZING` autour des morceaux problématiques pour les désactiver uniquement quand on compile pour fuzzer. En production, le code reste 100 % normal. En fuzzing, les checksums sont court-circuités, l'horloge renvoie une valeur fixe, le random est seedé de façon déterministe. D'un coup, le fuzzer voit du progrès et trouve des bugs. C'est une technique reconnue, indispensable pour fuzzer du code "réel".

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `fuzzing-obstacles`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/fuzzing-obstacles/SKILL.md`

constant-time-testing

constant-time-testing

> *Catalogue généré le 2026-05-11*

En une phrase

Le constant-time testing vérifie qu'une opération cryptographique met toujours exactement le même temps à s'exécuter — sinon un attaquant peut deviner ta clé secrète juste en chronométrant le serveur.

Quand l'utiliser

- Auditer une implémentation de crypto maison (RSA, AES, ECDSA…).
- Vérifier qu'une fonction de comparaison de signature ne fuit pas d'info par le temps.
- Auditer une bibliothèque crypto avant déploiement.
- Quand tu fais de la crypto post-quantique (Kyber, Dilithium…).
- Vérifier qu'une fonction `compare_password` ne révèle pas la longueur du bon mot de passe.

Comment l'invoquer

- **Slash command** : `/constant-time-testing`
- **Phrases déclencheurs (texte)** : "timing side channel", "constant-time crypto", "timing attack"
- **Auto-invocation** : Sur demande explicite

Description détaillée

Imagine que tu vérifies un mot de passe. Si ton code dit "non, faux !" plus vite quand la première lettre est fausse que quand les 10 premières lettres sont bonnes mais la 11e fausse, alors un attaquant peut deviner ton mot de passe lettre par lettre en chronométrant. C'est une "attaque par canal auxiliaire temporel" (timing attack). Elle existe depuis 1996 (Kocher) et a déjà cassé en pratique RSA, OpenSSL et même de la crypto post-quantique récente.

Le constant-time testing consiste à vérifier que ton code crypto met toujours **exactement le même temps**, **quel que soit le secret** qu'il traite. Concrètement, il faut s'assurer que les branchements (if/else) et les accès mémoire ne dépendent pas du secret. Ça paraît simple, mais le compilateur peut réintroduire des branches sans prévenir, et les processeurs modernes ont des optimisations (caches, prédictions) qui créent des micro-différences mesurables, même à distance sur un réseau.

Cette skill te guide dans l'audit : repérer les patterns dangereux (early returns sur secret, accès tableau indexé par un secret, divisions/multiplications variables), utiliser des outils comme `dudect`, `ctgrind` ou `timecop` pour mesurer expérimentalement le temps, et appliquer des patterns "constant-time" reconnus. Indispensable pour toute crypto destinée à un usage réel.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `constant-time-testing`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/constant-time-testing/SKILL.md`

harness-writing

harness-writing

> *Catalogue généré le 2026-05-11*

En une phrase

Le harness (ou "harnais") c'est la petite fonction d'entrée qu'on écrit pour brancher un fuzzer à son code : un mauvais harness = aucun bug trouvé, donc c'est l'élément le plus critique d'une campagne de fuzzing.

Quand l'utiliser

- Démarrer une nouvelle campagne de fuzzing (avant tout, écrire le bon harness).
- Améliorer un harness existant qui ne trouve plus de bugs.
- Convertir des bytes aléatoires en arguments typés pour ta fonction (FuzzedDataProvider).
- Garantir que ton harness est déterministe (même entrée = même comportement).
- Couvrir plusieurs APIs avec un seul harness (interleaved fuzzing).

Comment l'invoquer

- **Slash command** : `/harness-writing`
- **Phrases déclencheurs (texte)** : "write fuzz harness", "fuzz target", "LLVMFuzzerTestOneInput"
- **Auto-invocation** : Sur demande explicite

Description détaillée

Un harness de fuzzing, c'est le "point d'entrée" écrit par toi qui relie le fuzzer (qui génère des octets aléatoires) à ton vrai code (qui a une vraie API typée). En libFuzzer/AFL++, c'est typiquement une fonction `LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)` : elle reçoit un paquet d'octets bruts et c'est à toi de les transformer en quelque chose que ta fonction comprend.

La qualité du harness détermine TOUT. Si tu fais juste `parser(data, size)` alors que ta vraie API attend une URL valide suivie d'un dict de headers, ton fuzzer va bombarder le parseur d'URL et jamais le code en aval. Inversement, un bon harness extrait intelligemment plusieurs valeurs typées des bytes (en utilisant `FuzzedDataProvider`), appelle plusieurs APIs dans un ordre cohérent, et fait du nettoyage entre chaque itération.

Les règles d'or : (1) reset l'état global entre chaque exécution (déterministe), (2) ne jamais quitter brutalement (`exit()` interdit), (3) libérer la mémoire allouée à chaque tour, (4) valider la taille d'entrée si nécessaire. Cette skill couvre les patterns pour C/C++, Rust (cargo-fuzz), Python (Atheris), Ruby (Ruzzy) et même les smart contracts. Tu peux faire un harness "single-API" (tester une fonction) ou "interleaved" (utiliser les premiers bytes pour choisir une opération parmi N).

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/testing-handbook-skills`
- **Nom interne** : `harness-writing`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/testing-handbook-skills/1.0.1/skills/harness-writing/SKILL.md`

property-based-testing

property-based-testing

> *Catalogue généré le 2026-05-11*

En une phrase

Le property-based testing teste des **propriétés universelles** ("encoder puis décoder doit redonner la valeur d'origine") au lieu de cas spécifiques — la machine génère elle-même des milliers d'exemples pour traquer un contre-exemple.

Quand l'utiliser

- Tester une paire encode/decode, serialize/deserialize, toJSON/fromJSON (propriété "roundtrip").
- Tester un parseur (URL, config, protocole, format custom).
- Tester une fonction de normalisation (idempotence : appliquer 2 fois doit donner le même résultat).
- Tester une fonction pure (mathématique, tri, comparaison).
- Tester un smart contract Solidity/Vyper (invariants sur les balances, supply, ACL).

Comment l'invoquer

- **Slash command** : `/property-based-testing`
- **Phrases déclencheurs (texte)** : "property-based testing", "PBT", "Hypothesis", "QuickCheck", "invariant"
- **Auto-invocation** : Détection automatique sur patterns encode/decode, parsers, validators

Description détaillée

Le testing classique, c'est : "je liste 5 cas spécifiques (string vide, string longue, caractère bizarre…) et je vérifie chacun à la main". Le problème : tu testes ce à quoi tu penses, donc tu rates les cas auxquels tu ne penses pas. Le property-based testing renverse l'approche : tu décris une **propriété** que ton code doit respecter pour **toute** entrée valide, et la bibliothèque génère pour toi des centaines d'entrées aléatoires (et tordues : valeurs limites, Unicode bizarre, listes vides, négatifs…) pour vérifier la propriété.

Exemples concrets de propriétés :

- **Roundtrip** : `decode(encode(x)) == x` pour tout `x`.
- **Idempotence** : `normalize(normalize(x)) == normalize(x)`.
- **Commutativité** : `add(a, b) == add(b, a)`.
- **Invariant** : "Après transfert, la somme des balances reste constante" (smart contract).

Quand la lib trouve un contre-exemple qui casse ta propriété, elle fait du "shrinking" : elle simplifie automatiquement l'entrée pour te donner le contre-exemple minimal (par exemple `[1, 0]` au lieu de `[42, 17, 99, 0, -5]`). C'est ultra utile pour comprendre le bug. Outils populaires : Hypothesis (Python), QuickCheck (Haskell), fast-check (JS/TS), proptest (Rust), Echidna/Medusa (Solidity). Cette skill couvre les patterns pour tous ces langages.

Pour aller plus loin

Pour les exemples concrets, options de configuration et patterns avancés, voir le SKILL.md original.

Source

- **Plugin** : `trailofbits/property-based-testing`
- **Nom interne** : `property-based-testing`
- **Fichier** : `/home/thymon/.claude/plugins/cache/trailofbits/property-based-testing/1.1.0/skills/property-based-testing/SKILL.md`