

Classic CRSs & Fuzzing Infrastructure

1. **crs-libfuzzer** — A lightweight CRS wrapping libFuzzer for coverage-guided fuzzing. Starting point for any new target; produces corpus, coverage reports, and crash artifacts consumable by downstream triage modules.
 2. **crs-aflplusplus** — A CRS built on AFL++ for binary-only fuzzing with QEMU mode, allowing grey-box fuzzing without source code — crucial when working with compiled binaries where source is unavailable.
 3. **crs-honggfuzz** — A CRS wrapping Honggfuzz for hardware-assisted coverage feedback and persistent fuzzing mode, which outperforms libFuzzer on certain program structures and multithreaded targets.
 4. **crs-symbolic-execution** — A CRS using [angr](#) (or [manticore](#)) to symbolically execute targets and generate inputs that reach deep code paths fuzzing alone misses. Especially effective on parsers and protocol state machines.
 5. **crs-sanitizers** — A CRS that compiles targets with AddressSanitizer, MemorySanitizer, ThreadSanitizer, and UBSanitizer enabled and then fuzzes them, surfacing more vulnerability classes than plain binary runs.
 6. **crs-crash-dedup** — A dedicated module for deduplicating and clustering crashes found by multiple CRSs in an ensemble run, using stack trace hashing and coverage-based grouping to reduce noise before triage.
 7. **crs-triage** — A module that takes raw crashes, determines exploitability (stack control, PC control, heap overflow vs. stack overflow), and assigns initial severity scores before passing them to deeper analysis modules.
-

Static Analysis & Code Audit

8. **crs-static-analysis** — A CRS running static analyzers (Semgrep, CodeQL, Flawfinder, cppcheck) in batch mode against C/C++ source code in the dataset, producing structured vulnerability reports mapped to CWE categories.
 9. **crs-dataflow** — A CRS that builds inter-procedural dataflow graphs from source or LLVM IR and tracks tainted input propagation from entry points to sensitive sinks (memory ops, format strings, syscalls) without executing the target.
 10. **crs-binary-diff** — A module that takes two versions of the same binary (e.g., before and after a vendor patch) and identifies changed basic blocks, then directs fuzzing energy specifically at the patched regions to find patch bypasses or regressions.
-

AI/ML-Powered CRSs

11. **crs-harness-gen (AI)** — An LLM-powered CRS that auto-generates fuzz harnesses for new targets, using [attack_surface_approximation](#) output as context so the LLM focuses on realistic entry points rather than generic wrappers.
 12. **crs-patch-gen (AI)** — An LLM-based patch generation CRS that takes a crash plus root cause analysis from [vulnerability_analytics](#) and proposes C-level patches, then verifies them using the sanitizer CRS in a closed loop.
 13. **crs-root-cause-analysis (AI)** — An ML/LLM module that takes a crash trace, binary, and disassembly and produces a natural-language root cause explanation, extending [vulnerability_analytics](#) with semantic understanding beyond crash classification.
 14. **crs-exploit-refinement (AI)** — An LLM-augmented layer on top of [automatic_exploit_generation](#) that takes a partially-working exploit skeleton and iteratively refines it using binary feedback, with the LLM proposing constraint relaxations at each failed step.
 15. **crs-vuln-classifier (ML)** — A trained classifier that takes binary features (call graph metrics, CFG complexity, entropy, import tables) and predicts likelihood of vulnerability in a function, helping prioritize where fuzzing and symbolic execution should focus.
 16. **crs-cve-correlator (AI)** — A module that takes a found vulnerability's characteristics (type, affected function, crash pattern) and queries NVD/CVE databases plus embedding search over known CVEs to find related past vulnerabilities and existing PoCs.
 17. **crs-coverage-guided-llm (AI)** — A CRS that uses code coverage feedback from fuzzing runs to prompt an LLM to generate targeted inputs aimed at uncovered branches — a hybrid between coverage-guided fuzzing and LLM-based input generation.
 18. **crs-neural-repair (AI/ML)** — A module that uses a fine-tuned code model (trained on C patch datasets like Big-Vul or CVEFixes) to generate ranked patch candidates for detected vulnerabilities, complementing the LLM-based [crs-patch-gen](#).
-

Platform & Tooling Modules

19. **module-decompiler-bridge** — An integration module that passes binaries through Ghidra or Binary Ninja headless and exports pseudo-C decompilation, which then feeds into AI-powered CRSs as additional context for understanding stripped binaries.
20. **module-coverage-dashboard** — A web dashboard (extending [docs_website](#)) that ingests coverage reports from all running CRS campaigns and shows live per-function coverage heat maps for targets in the dataset.

- 21. `module-severity-scorer`** — An automated CVSS 3.1 score estimator that takes structured vulnerability metadata (type, access vector, privileges required, scope) and outputs a standardized score for found bugs, enabling consistent prioritization across campaigns.
- 22. `module-multilang-support`** — An extension of the dataset and CRS modules to support source-available targets in Python (via `atheris`), Go (`go-fuzz`), and Rust (`cargo-fuzz`), broadening OpenCRS beyond C/C++ binaries.
- 23. `module-ci-plugin`** — A reusable GitHub Actions composite action and GitLab CI template that any open-source project can drop into their repo to run OpenCRS CRS campaigns automatically on every push or release.
- 24. `module-patch-verifier`** — A module that takes a proposed patch, applies it to the target, rebuilds, re-runs the crashing input, and verifies the fix without regression — closing the loop between bug-finding and bug-fixing entirely within the framework.
-

Cryptography

- 25. `crs-crypto-misuse`** — A static + dynamic analysis CRS that detects cryptographic API misuse in source code and binaries: ECB mode usage, hardcoded IVs, weak key sizes, predictable seeds in PRNGs, broken hash functions (MD5/SHA1) used for security purposes. Works across C, Java, Python, Go.
- 26. `crs-side-channel`** — A CRS that instruments compiled programs with hardware performance counters (via `perf` or `libpfm`) and detects timing and cache-based side channels — particularly secret-dependent branching and memory access patterns in crypto implementations (RSA, AES, curve operations).
- 27. `crs-differential-crypto`** — A fuzzing-based CRS that takes two implementations of the same cryptographic primitive (e.g., two AES libraries, or a reference vs. optimized implementation) and runs differential testing to surface behavioral divergences that could indicate implementation bugs or algorithmic weaknesses.
- 28. `crs-tls-prober`** — A CRS that fuzzes TLS/DTLS handshake state machines by crafting malformed records at every protocol layer (ClientHello extensions, certificate chains, cipher suite negotiation) to find implementation bugs in OpenSSL, BoringSSL, mbedTLS, and others.
- 29. `crs-pqc-migration (AI)`** — An LLM-powered module that scans codebases for uses of classical cryptography (RSA, ECDSA, DH) and generates migration candidates to post-quantum equivalents (ML-KEM, ML-DSA), with an analysis of where key material flows through the codebase.
- 30. `crs-protocol-oracle`** — A black-box CRS that probes running cryptographic services for padding oracle, BEAST, CRIME/BREACH, LUCKY13, and Bleichenbacher

vulnerabilities through adaptive chosen-ciphertext query strategies without requiring source access.

Network Protocols

31. [crs-stateful-protocol-fuzzer](#) — A CRS that models network protocols as state machines (from RFCs or captured PCAPs) and fuzzes them state-by-state, covering protocols like SMB, RDP, MQTT, CoAP, QUIC, and SIP — not just the packet format, but the full session flow.

32. [crs-bgp-hijack-detector](#) — A monitoring and analysis module that ingests live BGP feed data (via RouteViews or RIPE RIS) and uses anomaly detection to flag potential route hijacks, prefix squatting, and AS path manipulation in near-real-time.

33. [crs-dns-chaos](#) — A CRS that fuzzes DNS resolver implementations with malformed zone files, crafted DNSSEC records, oversized EDNS0 responses, and cache poisoning payloads — targeting both authoritative and recursive resolver software.

34. [crs-wifi-protocol](#) — A CRS for 802.11 protocol stack fuzzing using software-defined radios (SDRs) or mac80211_hwsim to inject malformed management frames (Beacon, Probe, Association, Authentication) and trigger driver-level vulnerabilities in Linux and Windows Wi-Fi stacks.

35. [crs-can-bus](#) — A CRS targeting automotive CAN bus implementations — fuzzes ECU firmware images for buffer overflows triggered by malformed CAN frames, and performs replay/injection testing on virtual CAN interfaces ([vcan](#)) to detect safety-critical bugs in automotive software.

36. [crs-industrial-protocols](#) — A CRS that fuzzes industrial control system protocols: Modbus, DNP3, IEC 61850, OPC-UA, Profinet. Targets PLCs and SCADA software with malformed packet sequences to find memory corruption and authentication bypass bugs in critical infrastructure software.

Firmware & Hardware

37. [crs-firmware-unpack](#) — A pipeline module that takes raw firmware blobs (from IoT devices, routers, embedded systems), runs unpacking (binwalk-based), identifies RTOS type and architecture, extracts file systems, and emits a structured target spec for downstream CRSs to consume.

38. [crs-uefi-analysis](#) — A CRS for UEFI/BIOS firmware security analysis: scans DXE drivers and SMM modules for known vulnerability patterns (SMM callout, TOCTOU in SMM handlers, SPI flash write protection bypass), decompiles PE32+ binaries, and runs fuzzing via QEMU/OVMF emulation.

39. `crs-hardware-side-channel (AI)` — An AI-assisted module that analyzes power traces or EM emissions (from real or simulated targets via ChipWhisperer) and uses ML models (CNNs, template attacks) to recover secret key material from embedded cryptographic hardware.

40. `crs-fpga-rtl-audit` — A CRS that ingests hardware description language source (Verilog/SystemVerilog/VHDL) and applies formal verification + fuzzing over simulation (Verilator, GHDL) to find security-critical bugs: unintended information flows, timing side channels baked into RTL, and privilege escalation in hardware SoC designs.

Web & Application Layer

41. `crs-graphql-fuzzer` — A CRS that introspects GraphQL schemas and generates adversarial query inputs targeting deeply nested queries (DoS via query depth), batch injection, broken field-level authorization, and schema injection — fully automated from a single endpoint URL.

42. `crs-oauth-flow-analysis` — A CRS that models OAuth 2.0 and OIDC flows as formal state machines and tests for known weaknesses: authorization code interception, open redirect abuse, PKCE downgrade, token leakage via referrer, and incorrect `state` parameter handling.

43. `crs-web-cache-poisoning` — A CRS that systematically probes web caches (Varnish, Nginx, Cloudflare) for cache poisoning gadgets by fuzzing unkeyed headers (`X-Forwarded-Host`, `X-Original-URL`, `X-Rewrite-URL`) and observing cache behavior differences.

44. `crs-deserialization` — A CRS targeting insecure deserialization bugs across multiple runtimes: Java (ysoserial gadget chains), Python (pickle), PHP (unserialize), .NET (BinaryFormatter) — with an LLM module that proposes new gadget chain candidates based on classpath analysis.

Smart Contracts & Blockchain

45. `crs-solidity-fuzzer` — A CRS that fuzzes Ethereum smart contracts (Solidity/Vyper) using Foundry/echidna for property-based testing, targeting reentrancy, integer overflow/underflow, flash loan attacks, and access control bugs — runs against both source and EVM bytecode.

46. `crs-evm-symbolic` — A symbolic execution CRS for EVM bytecode (using Mythril or a custom angr-EVM backend) that exhaustively explores contract state transitions to find unreachable kill-switch conditions, unauthorized selfdestruct calls, and storage collision bugs in proxy patterns.

47. `crs-defi-invariant` (AI) — An LLM-assisted CRS that reads DeFi protocol documentation and smart contract source, infers economic invariants (TVL conservation, price oracle integrity, collateralization ratio), then generates fuzz campaigns that try to violate those invariants through crafted transaction sequences.

Kernel & Hypervisor

48. `crs-syscall-fuzzer` — A CRS that fuzzes the Linux kernel syscall interface using coverage-guided syscall sequence generation (inspired by Syzkaller but pluggable into the OSS-CRS orchestration layer), with OpenCRS-specific harnesses for Android kernels and embedded Linux variants.

49. `crs-hypervisor-escape` — A CRS targeting hypervisor attack surfaces: KVM, QEMU device emulation, VMware VMCI, Hyper-V hypercalls. Fuzzes the guest-to-host boundary by generating malformed I/O port accesses, MMIO writes, and virtio queue manipulations from inside a controlled guest VM.

50. `crs-ebpf-verifier` — A CRS that generates adversarial eBPF programs designed to probe the Linux kernel eBPF verifier for logic bugs — value range analysis bypasses, type confusion in pointer arithmetic, speculative execution leaks — without triggering the verifier's safety rejections.

AI/ML System Security

51. `crs-adversarial-ml` — A CRS that generates adversarial examples against ML models deployed in security tools (malware classifiers, IDS models, spam filters) using gradient-based and black-box attacks (FGSM, PGD, square attack), measuring how much perturbation is needed to flip a classification.

52. `crs-model-extraction` — A black-box CRS that attempts to reconstruct a proprietary ML model's decision boundary through adaptive query strategies, then uses the extracted surrogate model to generate adversarial inputs or infer training data properties.

53. `crs-llm-prompt-injection` (AI) — A CRS that systematically generates prompt injection payloads against LLM-powered applications (agents, RAG pipelines, code assistants) — testing for data exfiltration via tool calls, system prompt leakage, SSRF via injected URLs in tool parameters, and privilege escalation via role confusion.

Supply Chain & Code Integrity

54. `crs-dependency-confusion` — A CRS that scans package manifests across ecosystems (npm, PyPI, Maven, RubyGems, crates.io) for dependency confusion

vulnerability patterns: internal package names that could be squatted on public registries, version pinning gaps, and typosquatting candidates.

55. `crs-binary-provenance` — A module that takes a compiled binary and attempts to reconstruct its build provenance: matches against known-good compiler toolchains, detects injected backdoors via diffing against source-compiled reference builds (reproducible builds analysis), and flags suspicious compiler artifacts.

56. `crs-ci-pipeline-audit (AI)` — An LLM-powered CRS that ingests CI/CD pipeline definitions (GitHub Actions, GitLab CI, CircleCI, Jenkins) and identifies security weaknesses: script injection via untrusted PR data, overly broad OIDC token scopes, secrets exposed in logs, and artifact tampering opportunities.

Radio & Physical Layer

57. `crs-rf-protocol-fuzzer` — A CRS that uses SDR hardware (HackRF, USRP, RTL-SDR) or software simulation to fuzz RF protocols: Zigbee, Z-Wave, LoRaWAN, Bluetooth LE advertisement parsing, and 4G/5G NAS layer messages — targeting both the physical demodulation stack and upper-layer protocol parsers.

58. `crs-gnss-spoofing-analysis` — A module that analyzes GNSS receiver firmware and software for susceptibility to signal spoofing and jamming, using replay attacks with pre-recorded or synthesized GNSS signals to test receiver integrity checks and fallback behavior.

Boundary-Pushing

59. `crs-jit-compiler-fuzzer` — A CRS that generates adversarial JavaScript/WebAssembly or JVM bytecode specifically designed to trigger JIT compiler bugs (incorrect deoptimization, type confusion in speculative execution, bounds check elimination bypasses) in V8, SpiderMonkey, HotSpot, and Graal.

60. `crs-file-format-polyglot` — A CRS that generates polyglot files — valid in multiple formats simultaneously (PDF+ZIP, HTML+SVG, JPEG+ZIP) — and feeds them to parsers across different applications, exploiting format ambiguity to trigger parser confusion bugs and bypass content-type validation.

61. `crs-memory-forensics` — A module that takes live memory dumps or VM snapshots and applies automated analysis: extracts running processes, decrypts in-memory secrets, identifies injected code via VAD tree anomalies, and correlates findings against threat intelligence feeds — all as structured CRS output.

62. `crs-language-runtime (AI)` — A CRS that fuzzes language runtimes themselves (CPython, Ruby MRI, PHP-FPM, Node.js) by generating semantically valid but edge-case

programs using an LLM to construct inputs that stress garbage collectors, reference counting, and JIT internals rather than just random byte sequences.

63. `crs-hardware-trojan-detector` (ML) — An ML-based CRS that analyzes IC netlists or synthesized gate-level designs to detect hardware trojans: rare activation condition patterns, anomalous combinational paths, and suspicious side-channel covert channels baked into hardware at design time.

64. `crs-quantum-circuit-audit` — A forward-looking CRS that analyzes quantum circuit implementations (Qiskit, Cirq, PennyLane) for security-relevant bugs: qubit measurement ordering leaking secret state, incorrect unitary decompositions in cryptographic protocols, and noise model assumptions that break security proofs.