



SOFTWARE

Fuzzing: What are the Latest Developments?

Explore the recent advances in fuzzing, including the challenges and opportunities it presents for high-integrity software development.

Paul Butcher, Dr. José Ruiz Related To: AdaCore April 9, 2026 13 min read

What you'll learn:

- How modern fuzz testing has evolved into a core assurance technique for embedded, real-time, and safety-critical software, and why it's essential where exhaustive testing is infeasible.
- How fuzzing complements static analysis, runtime verification, and formal methods to form a layered, risk-driven assurance strategy for high-integrity systems.

- How recent innovations, such as grammar-based, hybrid, and AI-assisted fuzzing, are improving defect detection, robustness, and confidence in safety- and mission-critical software.

[Fuzz testing, or fuzzing](#), has become a cornerstone of modern software assurance. Once regarded as a niche technique primarily for security research, it's evolved into a mainstream practice for uncovering security vulnerabilities and functional defects across various systems.

Its adoption is expanding rapidly, from cloud-native stacks to deeply embedded systems, and innovations such as genetic mutation algorithms,

grammar-based fuzzing, hybrid fuzzing, and AI-assisted fuzzing continue to push the boundaries of what's achievable with a fuzz testing validation strategy. In safety- and mission-critical domains, where failures can have severe consequences, fuzzing is increasingly integrated alongside [formal methods, static analysis, and runtime verification](#) as part of a comprehensive assurance strategy.

Table of Contents

- [How important is fuzzing on embedded targets?](#)
- [Is fuzzing well supported within the wider software-development community, particularly for non-mainstream or constrained platforms?](#)
- [What's the interplay between fuzzing and other software-development tools or methodologies that aim to detect bugs or vulnerabilities, such as static analysis or runtime verification?](#)
- [Can fuzz testing still play a role in a development process that uses formal proof?](#)
- [What are some recent innovations in fuzzing techniques, and how might they improve testing for safety-critical systems?](#)
- [What are the key challenges of applying fuzzing to systems with real-time constraints or HIL testing?](#)
- [To what extent does fuzzing help uncover security vulnerabilities vs. functional bugs? Is it evolving more toward one area?](#)
- [Are there any notable success stories or case studies where fuzzing significantly improved software robustness or uncovered latent critical issues?](#)

- [How should development teams prioritize fuzzing relative to other bug-detection strategies, especially when resources are limited?](#)
- [What metrics or benchmarks are emerging to assess the effectiveness of a fuzzing campaign, particularly in safety- and mission-critical domains?](#)
- [Where are we with fuzzing?](#)
- [References](#)

How important is fuzzing on embedded targets?

[Fuzzing](#) is highly important for embedded targets. Embedded software increasingly includes complex data-processing components, communication stacks, and application programming interfaces (APIs). The issue is that, while more processor compute power allows for more interconnected systems, the expanded system input space broadens the attack surface and makes verification and validation more challenging to achieve.

With 64-bit architectures, even simple data types necessitate the compromise of verification strategies, and it's widely accepted that completeness testing isn't feasible. For example, within a simple C++ subprogram with four 64-bit Integer parameters, the total possible test case permutations equals 2^{256} . as

introduce more complex input structures, you clearly exacerbate the problem, leading to untested scenarios where vulnerabilities can linger and cause failures.

These failures aren't trivial, as they can lead to high-impact outcomes such as system crashes, degraded performance, or triggered safety hazards. Unit testing is widely regarded as an excellent methodology for proving functional requirements have been satisfied. However, it's hopelessly inadequate at arguing that a system is free of runtime errors.

Instead, we must utilize dedicated technology, such as fuzzing, which is designed to exercise multiple scenarios through all code paths within a control flow, significantly increasing assurance that our software is bug-free. Modern approaches to fuzzing embedded targets include emulation- and simulation-based fuzzing, as well as

each Integer can represent 2^{64} distinct values. If we wanted to write that as a full number, it would be 78 digits long!

Or to put it another way, if we could execute the test cases at a rate of 1 trillion times a second (which would be extremely challenging with today's technology), it would still take considerably longer than the known age of the universe to execute all of the permutations. When you

[Go to Table of Contents](#)

Is fuzzing well supported within the wider software-development community, particularly for non-mainstream or constrained platforms?

Within mainstream environments, fuzzing is well-supported. Coverage-guided fuzzers such as AFL++, libFuzzer, and honggfuzz are widely used and mature, while large-scale initiatives such as Google's OSS-Fuzz provide infrastructure and resources for continuous fuzzing of thousands of open-source projects (Google,

host-based fuzzing. These allow teams to start testing earlier in the development cycle, even before the final hardware is available.

Such techniques have already demonstrated their effectiveness in surfacing security and robustness issues frequently missed by traditional unit testing or code reviews. Recent reviews confirm the relevance of embedded fuzzing and the growing ecosystem of tool support (Nguyen et al., 2022).

vulnerabilities, such as static analysis or runtime verification?

Fuzzing is best understood as a complement to, rather than a replacement for, other bug-detection methodologies. Static analysis effectively identifies broad classes of potential defects, such as null-pointer dereferences or data taint issues, across an entire codebase without executing the program. However, it can't always capture real execution context or account for complex environmental inputs.

Fuzzing addresses this gap by

2025).

By contrast, support for constrained or non-mainstream platforms, such as microcontrollers with limited resources, is less consistent. Nevertheless, steady progress has been made in this area, with frameworks and methodologies emerging that adapt fuzzing to memory-constrained and real-time embedded contexts. Therefore, support for constrained platforms, although uneven today, is improving over time.

What's the interplay between fuzzing and other software-development tools or methodologies that aim to detect bugs or

[Go to Table of Contents](#)

Can fuzz testing still play a role in a development process that uses formal proof?

Yes. While formal proof provides mathematical guarantees about specific properties of the verified components, it doesn't cover the

dynamically exercising the software with malformed or unexpected inputs, revealing concrete issues such as crashes, hangs, assertion failures, or subtle logic errors. Symbolic or concolic execution adds further depth by solving constraints to generate inputs that explore hard-to-reach code paths (Godefroid et al., 2008).

Runtime verification strengthens this ecosystem by embedding oracles, such as assertions, invariant monitors, and other differential fuzzing techniques, which detect and flag behaviors that might otherwise pass unnoticed. Together, these techniques form a layered assurance stack, providing broader coverage and higher confidence in the robustness of the software.

developers in understanding the results.

Fuzzing plays a vital role in stress-testing these boundaries, validating assumptions, checking that specifications hold under various real-world conditions, and providing concrete reproducers to aid in patching. It's also valuable for

entire system. Practical software invariably includes unproven modules, third-party libraries, integration code, and assumptions about the environment, all of which may introduce vulnerabilities or errors. Furthermore, while formal proof is very effective at finding issues, it can also struggle to generate reproducer scenarios to aid

[Go to Table of Contents](#)

What are some recent innovations in fuzzing techniques, and how might they improve testing for safety-critical systems?

Recent years have seen significant innovation in fuzzing techniques, particularly in safety-critical domains.

Grammar-based fuzzing, for example, combines input grammars with coverage feedback to generate well-structured inputs that exercise deep semantic states in protocols or data formats such as ASN.1 and JSON (Aschermann et al., 2019). This approach increases the likelihood of valid test cases and improves exploration of novel execution paths,

detecting regressions, integration issues, and misuse scenarios such as incorrect API sequencing.

In practice, many teams adopt a hybrid strategy: They apply formal methods to the most critical components and complement these proofs with fuzzing for interfaces, parsers, and supporting code.

fuzzer mutation cycles aren't wasted by generating repeating or invalid test cases.

Hybrid fuzzing integrates symbolic or concolic execution with traditional fuzzing, enabling the bypass of checksums, magic values, complex branch conditions, and other defensive barriers that block naive mutation-based approaches (Godefroid et al., 2008). AI- and machine-learning-assisted fuzzing, exemplified by systems such as NEUZZ, leverages neural networks to model input features and guide fuzzing more effectively when coverage feedback is sparse (She et al., 2019).

Another notable development is API or stateful fuzzing, as seen in tools

making it well-suited to avionics and automotive communications protocols.

Other approaches involve the use of specialist data representations that are considered highly fuzz-test efficient, including tightly packed binary representations of mutable test case components. They ensure that

[Go to Table of Contents](#)

What are the key challenges of applying fuzzing to systems with real-time constraints or HIL testing?

Applying fuzzing to real-time or hardware-in-the-loop (HIL) systems presents unique challenges. Real-time systems rely on strict timing determinism, meaning input mutations that alter execution time can produce misleading results or mask genuine faults.

Observability is also complex. Failures can manifest not as crashes, but as subtler phenomena, such as missed deadlines, degraded control behavior, or undetected buffer overflows, which require domain-

like RESTler, which generate valid request sequences from specifications such as OpenAPI and explore dependencies between states (Godefroid et al., 2017). These innovations collectively increase the depth, precision, and efficiency of fuzzing, making it more applicable to safety- and mission-critical environments.

specific monitors to detect. Throughput presents another limitation: Physical hardware rigs can be slow compared to virtual environments, or require complex configurations and/or startup sequences to load tests into memory before execution, limiting the number of inputs that can be tested.

To overcome this, teams frequently use host-based software-in-the-loop (SIL) or hardware emulation to generate and triage inputs at scale, before selectively replaying high-value cases on real hardware. Finally, achieving sufficient fidelity in the test environment is a persistent challenge, as models, sensors, and actuators must accurately reflect real-world conditions to ensure meaningful results.

[Go to Table of Contents](#)

To what extent does fuzzing help uncover security vulnerabilities vs. functional bugs? Is it evolving more toward one area?

Historically, fuzzing was closely associated with discovering security vulnerabilities, particularly memory-safety issues and parser flaws. However, its scope has broadened significantly. Large-scale efforts such as OSS-Fuzz show that fuzzing now contributes equally to uncovering correctness and robustness issues, including functional bugs unrelated to security ([Google, 2025](#)).

For embedded systems, fuzzing is increasingly applied to protocol state handling, error recovery, and resource-exhaustion scenarios. While security remains a key driver, fuzzing is evolving into a general-purpose technique for improving software reliability.

Furthermore, systems developed using programming languages that support concepts such as design-by-contract or other mechanisms to

Several well-documented success stories demonstrate the tangible benefits of fuzzing. Microsoft's SAGE whitebox fuzzer uncovered approximately one-third of all file-format bugs during the development of Windows 7, many of which had gone undetected by other methods. It saved the company millions of dollars in potential post-release costs (Godefroid, 2011).

Similarly, the syzkaller framework has continuously identified thousands of issues in the Linux kernel through its syzbot infrastructure, contributing substantially to the kernel's long-term robustness (Google, 2025). In the cloud-native ecosystem, targeted fuzzing campaigns led by the [Cloud Native Computing Foundation \(CNCF\)](#) have yielded actionable results, uncovering vulnerabilities and robustness issues across widely used projects (CNCF, 2023).

The pace at which software needs to be developed in the modern world to ensure products are brought to market in a timely fashion, as well as meeting the stringent high-integrity industry standards for quality and

contract or other mechanisms to dynamically assert functional correctness are particularly well-suited to fuzzing. Here, the fuzzer can be used to gain further assurance that functional requirements are satisfied, while also providing the normal security verification benefits.

Are there any notable success stories or case studies where fuzzing significantly improved software robustness or uncovered latent critical issues?

[Go to Table of Contents](#)

How should development teams prioritize fuzzing relative to other bug-detection strategies, especially when resources are limited?

Multiple strategies exist for prioritizing which components of a system should be fuzz tested next, but it's best that development teams always adopt a risk-driven approach when prioritizing fuzzing. One approach is

robustness, is evident in multiple places, particularly the rapidly emerging drone industry.

Within the UK, [Volant](#) is developing complex autonomous flight guidance systems and uses fuzz testing, via AdaCore's GNATfuzz technology, to uncover latent defects that traditional testing methods may miss. By integrating it into their continuous integration workflow, they have identified issues such as unit tests that weren't updated after changes to the underlying base types and incorrect usage of assumptions within formal proof arguments.

Integrating coverage-guided fuzzing into continuous integration (CI) pipelines ensures that test coverage improves steadily, regressions are caught quickly, and coverage analytics can be used to help prioritize future campaigns. Merge approvals should ideally be conditional on a "no new crashes" policy, preventing the introduction of defects into the codebase.

For structured inputs such as ASN.1, Protobuf, or CAN bus messages, structure-aware or fuzz-efficient

to focus on components with the highest exposure to external inputs, such as parsers, protocol stacks, and command interfaces.

However, fuzzing is a highly configurable and flexible strategy, and fuzz testing doesn't always have to happen at the system level.

Therefore, it's also invaluable to specifically target highly complex algorithmic components, particularly where gaps remain in unit testing strategies, e.g., boundary value analysis, or aspects of the code that can't be formally proven.

[Go to Table of Contents](#)

What metrics or benchmarks are emerging to assess the effectiveness of a fuzzing campaign, particularly in safety- and mission-critical domains?

Several metrics and benchmarks are gaining traction as ways to measure the effectiveness of fuzzing campaigns. Coverage progression remains fundamental, with edge, line,

binary representation fuzzing maximizes effectiveness by ensuring that generated inputs remain valid while maximizing edge case exploration.

Scaling should occur incrementally, beginning with SIL harnesses and extending to HIL for high-value scenarios. Finally, fuzzing should be combined with complementary methods, such as static analysis, runtime verification, and advanced branch-solving techniques like symbolic and concolic execution, to ensure comprehensive coverage and efficiency.

detection strength is also key; executing fuzz-testing campaigns with a poor ability to detect issues will lower assurance.

In comparison, incorporating memory sanitizer technology (which can be either hardware or software-based) will ensure that more unsafe memory instruction vulnerabilities can be detected, thereby elevating assurance. The strength of test oracles also matters. Campaigns that detect subtle failures through assertions or invariant monitors are

and function coverage compared against unit and integration tests. The quality of findings is another crucial dimension, assessed by the number of unique crashes, the triaged root causes, the proportion of bugs fixed, and the rate of vulnerability disclosures (CVEs, where applicable).

Campaign health can be measured by throughput (executions per second), uptime, corpus growth, and reproducibility rates. Anomaly

[Go to Table of Contents](#)

Where are we with fuzzing?

Fuzzing has matured from a specialist technique into a widely adopted practice for strengthening software reliability, robustness, and security. Its role in embedded systems, real-time environments, and safety-critical domains is becoming increasingly clear. Innovations such as grammar-based and hybrid fuzzing are opening

[Go to Table of Contents](#)

References

Aschermann, C., Schumilo, S.,
Abbas, A., Gawlik, D. & Holz, T.

more valuable than those relying solely on crashes.

Finally, longitudinal evidence, such as continuous participation in initiatives like OSS-Fuzz or the presence of dashboards similar to syzbot, helps teams evaluate the sustainability and impact of fuzzing over time (Google, 2025). Tracking the quality and freshness of fuzzing harnesses is critical to avoid degradation and maintain meaningful results.

new opportunities for deeper and more effective testing. When combined with static analysis, runtime verification, and formal methods, fuzzing delivers complementary assurance that no single approach can provide in isolation.

While challenges remain, success stories across commercial and open-source ecosystems show that fuzzing delivers tangible value.

Systems Security Symposium (NDSS).

Godefroid, P., Peleg, H. & Singh, R. (2017). *DEFSTAR: Stateful DEFSTAR ADI*

Abbas, A., Gawlik, R. & Holz, T. (2019). *NAUTILUS: Fishing for Deep Bugs with Grammars*. Network and Distributed Systems Security Symposium (NDSS).

CNCF (2023). *Fuzzing Audits and Reports*. Cloud Native Computing Foundation. Available at: <https://www.cncf.io> [Accessed 29 Sept 2025].

Godefroid, P. (2011). *SAGE: Whitebox Fuzzing for Security and Reliability*. Communications of the ACM, 55(3), pp. 40–44.

Godefroid, P., Levin, M.Y. & Molnar, D. (2008). *Automated Whitebox Fuzz Testing*. Network and Distributed

(2017). [NEUTER: Systemic NEUTER API Fuzzing](#). Microsoft Research Technical Report.

Google (2025). *OSS-Fuzz: Continuous Fuzzing for Open Source Software*. Google Security Blog. Available at: <https://github.com/google/oss-fuzz> [Accessed 29 Sept 2025].

Nguyen, C., Phung, Q., Dang, T. & Tran, L. (2022). *Embedded Fuzzing: A Review*. *Cybersecurity*, 5(9).

She, D., Pei, K., Epstein, D., Yang, J., Jana, S. & Ray, B. (2019). *NEUZZ: Efficient Fuzzing with Neural Program Smoothing*. IEEE Symposium on Security and Privacy.

About the Author

Paul Butcher

UK PROGRAMME MANAGER AND UNIT DIRECTOR OF DYNAMIC ANALYSIS. ADACORE

Paul Butcher is the UK Programme Manager and Unit Director of Dynamic Analysis for

most iconic aerospace and defense platforms.

[Show more](#)



Dr. José Ruiz

PRODUCT MANAGER, ADACORE



Dr. José Ruiz is a Product Manager at AdaCore. He has more than 20 years of experience in developing real-time systems, participating in many different industrial and

[Show more](#)

Source URL: <https://www.electronicdesign.com/technologies/embedded/software/article/55369732/adacore-fuzzing-what-are-the-latest-developments>