



Codonomicon whitepaper:

# Fuzzing Challenges: Metrics and Coverage



- 1** Introduction
- 2** Fixing costs
- 3** Fuzzing benefits
- 4** Ways of creating fuzzers
- 5** Metrics for Fuzzing Coverage
- 6** Conclusions
- 7** Links

CODENOMICON Ltd. | [info@codenomicon.com](mailto:info@codenomicon.com) | [www.codenomicon.com](http://www.codenomicon.com)

Tutkijantie 4E | FIN-90570 OULU | FINLAND | +358 424 7431  
10670 North Tantau Avenue | Cupertino, CA 95014 | UNITED STATES | +1 408 252 4000

---

PREEMPTIVE SECURITY AND ROBUSTNESS TESTING SOLUTIONS

# 1 Introduction

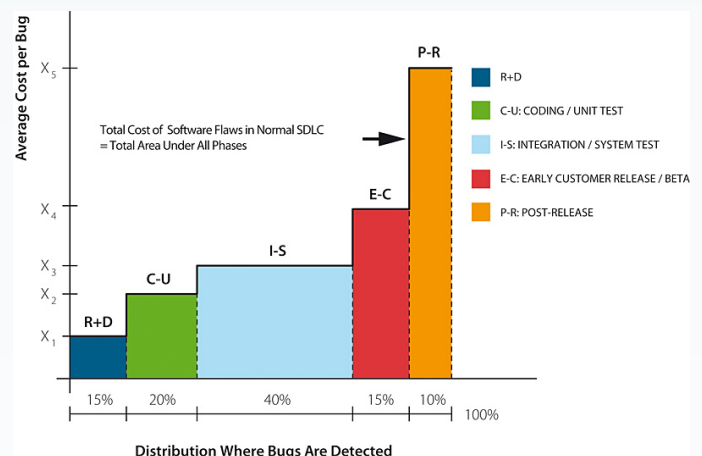
The fast adoption rate of new technologies has overwhelmed traditional security testing. Meanwhile, fuzzing has proven its ability to answer the changing testing. This adaptability stems from the very nature of fuzz testing: while most traditional security testing techniques focus on scanning for known vulnerabilities, fuzzers are based solely on the protocols they test, thus they have the ability to expose all vulnerabilities, not just the known, published ones. The idea behind fuzzing is fairly simple: it is a functional black-box testing technique, in which unexpected, abnormal inputs, called fuzzers, are generated and then fed to the system under test (SUT) and the behavior of the SUT is monitored. If the SUT crashes, then there is a bug in the software. However, different fuzzing methods differ from each other in how the fuzzing is done, and this has a significant impact on both the metrics and the coverage of the tests. In this paper, we will look at four different fuzzing techniques, namely random fuzzing, block-based fuzzing, traffic capture fuzzing and model-based fuzzing, and discuss test metrics and coverage in each case. We will also introduce the terms precision and accuracy to discuss the differences between different fuzzing techniques.

# 2 Fixing costs

One of the key metrics used for both known and unknown vulnerabilities is their estimated cost for developer organizations and customers. This metric can be used to compare different security testing techniques. All security vulnerabilities are harmful for business, however, unknown or zero-day bugs can potentially cause more damage, because, in contrast to published threats with patches, the existence of zero-day bugs is not known and therefore no ready patches exist for them. If there is no patch available to fix the vulnerability, then software developers have to go through the painstaking process of trying to develop and deploy the corrective measures, while unhappy customers are complaining and taking their business elsewhere. In addition to being damaging to company reputation, post-deployment fixes are also costly in terms of downtime and lost business. Distributing fixes is also more costly, after deployment. Informing customers of available software updates for programs is still fairly convenient, but just think about costs of repairing vulnerabilities in automotive software after deployment compared to fixing them in the development phase.

# 3 Fuzzing benefits

The main benefit of fuzzing is its unparalleled ability to find unknown bugs. Fuzzing enables testers to accurately simulate potential attacks, and to patch the found vulnerabilities, before somebody else finds them and exploits them. Instead



of, racing against the clock to get your system to work again and spending valuable resources in calming down angry customers, you can discover flaws and create patches for them proactively, before any problems occur. Furthermore, if vulnerabilities are fixed early on during the software development process, then there simply will not be any time for anyone to devise an attack. Fuzzing does what attackers do, but before them. By integrating fuzz testing into your software development process, you can discover flaws at the earliest possible moment and the earlier the bugs are discovered the cheaper and easier it is to fix them. You can actually gain time and save money through preemptive security and robustness testing.

The importance of fuzzing is heightened by the fact that finding unknown vulnerabilities is becoming increasingly challenging software testing. Software release cycles are getting

faster and new technologies are increasingly complex, creating a perfect breeding ground for bugs. Moreover, a growing share of vulnerabilities is never disclosed publicly. Instead, they are sold and distributed within underground hacker communities. Companies, which continue to rely on security vulnerabilities reported by third party security researchers, will lose their competitive advantage to competitors, who are building their internal product security practices. As software development companies can no longer rely on user communities to find bugs, they need to find new ways to protect their products and services. Many companies have started to endorse in-built security and to adopt the Secure Development Lifecycle (SDL) into their software development process. Fuzzing is generally used in the SDL to verify the security and robustness of a system before release.

## 4 Ways of creating fuzzers

There is an infinite number of ways to manipulate protocol messages to create anomalous feeds. The number of misuse-cases created for each use-case can easily be thousands or even millions, thus most fuzzing solutions contain at least some degree of automation. Based on the methodology used to create the tests and the way the test target functionality is learned a difference can be made between two popular types of fuzzers. Mutation-based (or template-based) Fuzzers read message structures from templates such as traffic captures (PCAP) or document files, like PDFs and GIFs, and then generate test cases the model either by using random mutations or by applying some predetermined algorithms to the data. Specification-based fuzzers learn the interface functionality from the protocol specification, and they can generate tests either from a full model of the protocol, or from meta-data added to protocol building blocks. Next, we will look at the four most common fuzzing techniques in more detail: Random Fuzzing, Block-based Fuzzing, Model-based Fuzzing and Traffic Capture Fuzzers.

### Random fuzzers

Random Fuzzers are the least effective technique for fuzzing, but the easiest to develop and deploy. A random fuzzer is the simplest of mutation based fuzzers. A random fuzzer typically reads in a seed such as a file or a message in a protocol exchange. The test generation technique is most often based on randomly adding, removing or modifying bits and bytes in those templates with almost zero knowledge on the internal structures in the messages or files. Random Fuzzers are almost always limited to testing only the first messages in complex protocols as they do not have any application logic built into them.

### Block-based fuzzers

Most open source frameworks use block-based fuzzing as a test generation technique. Some of these methods can read templates or PCAP files, while some of them require you to build the entire protocol specification using XML-like structures. The idea in block-based Fuzzers is to break down protocol structures used in message exchanges, and to manually tag these building-blocks with meta-data, which then enables the automation of the test generation process. Meta-data could, for example, be used to define that one data block is a string, and another an integer.

### Model-based fuzzers

In model-based Fuzzing, the process of data element identification is automated by using the actual protocol specifications to define the protocol interface functionality. The protocol model is fully executable, i.e. the model acts like a conformance test generator. Model-based tests know everything about the tested protocol, and therefore, they can function as the peer of the tested SUT and also test deeper protocol layers. In addition, they can target tests and thereby significantly reduce the number of test cases needed for good test coverage, resulting in shorter test run times.

### Traffic-capture fuzzers

Traffic Capture Fuzzing takes a real-life capture of network traffic and fuzzes that to find vulnerabilities. A traffic capture does not represent the whole protocol implementation; it is merely a sample of network traffic. You might miss important rare messages, and thus fail to test the entire software implementation. A Model-Based approach covers the standard specifications and contains an optimized set of tests to cover the specifications, thus providing wider test coverage of the actual implementation. However, specifications needed for model-based fuzzers might not always be available. In these cases it is advisable to use traffic capture based fuzzer, which require no information on the tested system.

## 5 Metrics for Fuzzing Coverage

Coverage can be seen as the sum of two features, precision and accuracy. Precision is concerned with protocol coverage. The precision of testing is determined by how well the tests cover the different protocol messages, message structures, tags and data definitions. Accuracy, on the other hand, measures how accurately the tests can find bugs within different protocol areas. Therefore, accuracy can be regarded

as a form of anomaly coverage. However, precision and accuracy are fairly abstract terms, thus, we will now look at more specific metrics for evaluating coverage.

### Attack surface

The first coverage analysis aspect is related to the attack surface. Test requirement analysis always starts off by identifying the interfaces that need testing. The number of different interfaces and the protocols they implement in various layers set the requirements for the fuzzers. Each protocol, file format, or API might require its own type of fuzzer, depending on the security requirements.

### Specifications

Second coverage metric is related to the specification that a fuzzer supports. This type of metric is easy to use with model-based fuzzers, as the basis of the tool is formed by the specifications used to create the fuzzer, and therefore they are easy to list. A model-based fuzzer should cover the entire specification. Whereas, mutation-based fuzzers do not necessarily fully cover the specification, as implementing or including one message exchange sample from a specification does not guarantee that the entire specification is covered. Typically when a mutation-based fuzzer claims specification support, it means it is interoperable with test targets implementing the specification.

### Statefulness

Especially regarding protocol fuzzing, the third-most critical metric is the level of statefulness of the selected Fuzzing approach. An entirely random fuzzer will typically only test the first messages in complex stateful protocols. The more state-aware the fuzzing approach you are using is, the deeper the fuzzer can go in complex protocols exchanges. The statefulness is a difficult requirement to define for Fuzzing tools, as it is more a metric for defining the quality of the used protocol model, and can, thus, only be verified by running the tests.

## 6 Conclusions

The most important metric for comparing fuzzing approaches is the amount of flaws the method finds. A simple approach, like random fuzzer, requires virtually no investments, but they can only find 10% of the vulnerabilities hiding in the software. A mutation-based fuzzer can find around 50% of the flaws. Once again the tool investments are minimal, but cost of using the tools and integrating them into the development processes are a lot greater. A fully model-based fuzzer can find as much as 80-90% of the flaws, but it can be the most expensive method to build and maintain. The choice of the tool is often based on integration capabilities and challenges, not coverage. If the protocol is standards-based, a model based fuzzer is often the right choice. But especially with emerging technologies and agile development processes, the specifications needed to create model-based tests are not always available. There might be no consensus on the protocol specification or the specification changes rapidly, or in some special cases the specifications are proprietary and not available to testers. In such cases traffic captures can be used to create mutation-based fuzzers. When the amount of interfaces is vast, random fuzzing might be a budgetary choice to get at least some fuzz test coverage before more advanced capabilities are introduced to the development process.

## 7 Links

For more information on fuzzer categorizations, see: Buzz on Fuzzing whitepaper at: <http://www.codenomicon.com/products/buzz-on-fuzzing.shtml>

For general information of fuzzing, see Fuzzing book by Ari Takanen et al.: <http://www.fuzz-test.com/>



CODENOMICON Ltd. | [info@codenomicon.com](mailto:info@codenomicon.com) | [www.codenomicon.com](http://www.codenomicon.com)

Tutkijantie 4E | FIN-90570 OULU | FINLAND | +358 424 7431  
10670 North Tantau Avenue | Cupertino, CA 95014 | UNITED STATES | +1 408 252 4000