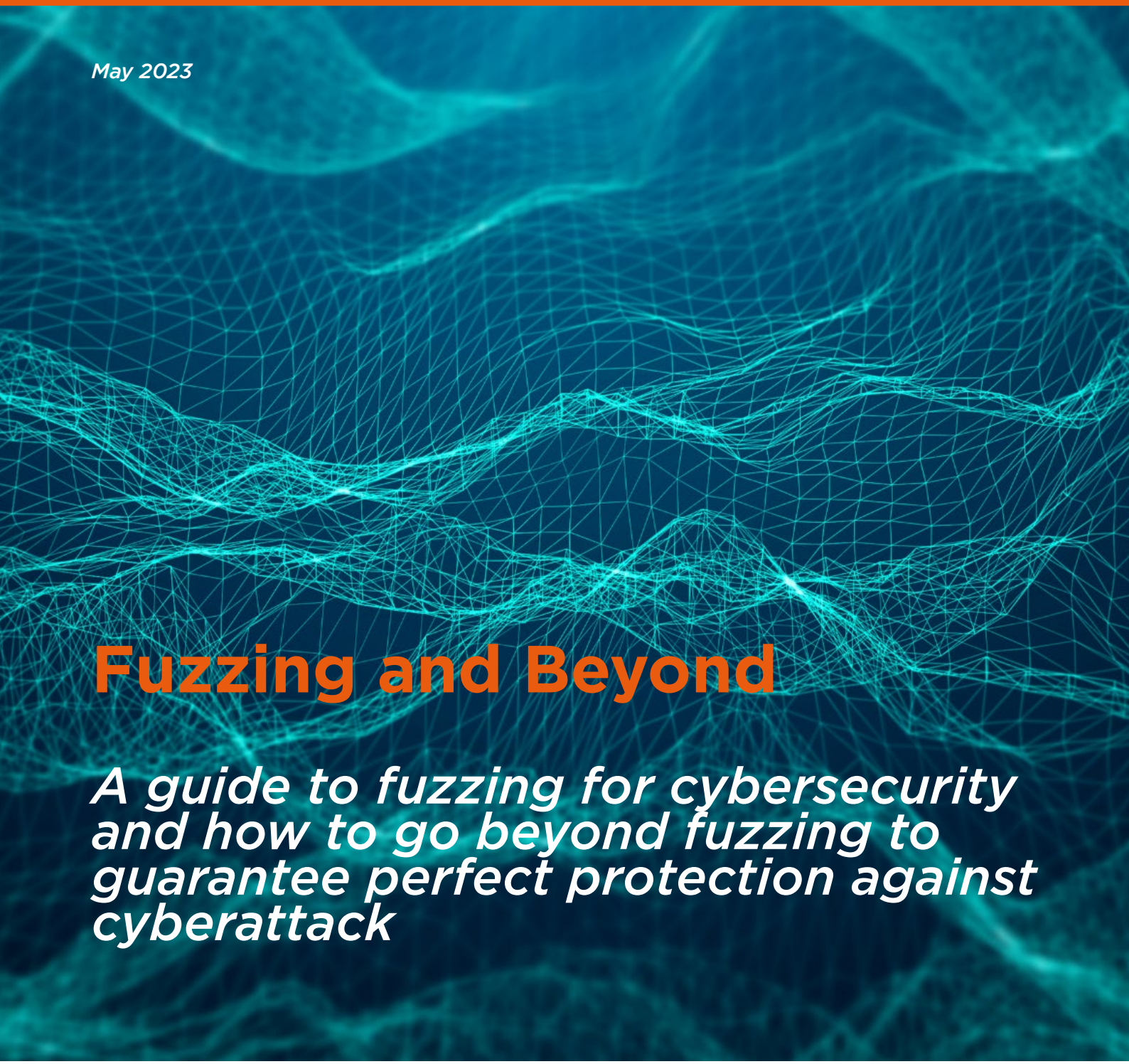


May 2023

Fuzzing and Beyond

A guide to fuzzing for cybersecurity and how to go beyond fuzzing to guarantee perfect protection against cyberattack

TRUST IN SOFT

Table of contents

FUZZING AND BEYOND

Fuzzing and Beyond

The rapid growth in the connectivity of software-driven products to Wi-Fi and cellular networks and the internet—in industries that include consumer electronics, smart home and office, IoT, industrial automation, automotive, aerospace and defense—has been forecast to continue for the foreseeable future.[1,2]

That growth has engendered a corresponding growth in attack surfaces and, subsequently, an increasing interest in exploiting those attack surfaces on the part of unscrupulous hackers and unfriendly governments. In short, the cyber threat is here to stay.

Countering that threat has become critically important to businesses and has resulted in a growing interest in the use of fuzzing on the part of both software development organizations and security specialists.

Fuzzing is a software testing technique that applies vast numbers of input combinations to a target program very rapidly, in an automated manner. By generating these inputs semi-randomly, fuzzing can test combinations the developer may not have anticipated while saving the developer the tedium of manually defining individual test cases. The goal is to reveal hard-to-find vulnerabilities that are rarely caught by conventional software testing.

Hackers frequently use fuzzing tools to find loopholes in code—untested input values that create unexpected behavior they can exploit remotely. They wait patiently while their fuzzer applies millions of random inputs until, finally, it uncovers a flaw that suits their purposes.

To unleash havoc, a hacker need only find a single vulnerability the developer failed to correct before release. That puts software vendors at a distinct disadvantage.

While testing for corner cases is a best practice, development teams need to go further. Under time-to-market pressure, they need new methodologies for efficient verification. Unlike hackers, they can't be satisfied with stumbling upon one exploitable vulnerability. They need to find and fix all the vulnerabilities they can while maintaining their release schedule.

A large proportion of issues in C/C++ code that are exploited by hackers is so-called undefined behavior. Undefined behavior is a technical term that includes all sorts of runtime errors such as buffer overflows, division by zero, null pointers, etc. Fuzzing is a great first step for uncovering undefined behavior that normal testing is not designed to catch. However, even the best fuzzers are not designed to catch every vulnerability, either because the fuzzer did not select the input that causes the problem or because the problem happened but there was no visible manifestation of it (so the fuzzer is unaware of the problem); they need to be paired with analysis tools that are.

What's more, for critical applications, it's often necessary to go beyond fuzzing to ensure airtight cybersecurity.

In this white paper, we'll examine fuzzing: what it is, who uses it, how they use it, its benefits, its limitations, and how those limitations can be mitigated. We'll also look at circumstances in which development teams need to go beyond fuzzing, along with some tools and methods that can guarantee the complete elimination of all exploitable vulnerabilities in your software or firmware.

We'll begin with the most basic question…

What is fuzzing?

FUZZING AND BEYOND

What is fuzzing?

To answer this question, it's best to start with a few definitions (located in the orange box to the left). We will use those from a recent survey of the published work on fuzzing titled, "The Art, Science, and Engineering of Fuzzing: A Survey" (Manès et al).[3] Its authors' intent was to consolidate and distill as much of the available research as possible and to resolve discrepancies between the various sources.

Fuzzing

The term "fuzz" was coined by Miller et al in 1990[4] to refer to the actions of an automated testing program that "generates a stream of random characters to be consumed by a target program."[5]

Manès et al define fuzzing as,

The execution of a PUT (Program Under Test) using inputs sampled from an input space (the "fuzz input space") that protrudes the expected input space of the PUT[6]

By "protrudes" the authors mean they consider fuzz inputs to be inputs that the PUT may not be expecting. They point out that "the sampling process is not necessarily randomized," as it had been originally envisioned, and "in practice, fuzzing almost surely runs for many iterations."[7]

In general practice, fuzzing is a software testing technique that rapidly applies large numbers of valid, nearly valid, or invalid inputs to a program, one after the other, in a search for undesired behaviors (vulnerabilities).

By "nearly valid" inputs, we mean inputs that meet the expected form of the input space but contain values that are malformed or unexpected. The idea is to automatically generate inputs for the tested program, trying to find such parameters and input data that cause the program to misbehave in some way. This may result in a safety or security flaw, such as a crash, memory leak, or arbitrary code execution.

Ultimately, the goal of fuzzing is to automate the process of finding vulnerabilities by generating a large number of test inputs that exercise a program or system in ways that are unexpected or that stress its functionality.

Motivations and principles of fuzzing

Programmers often make many assumptions concerning the structure and contents of the data their programs handle internally.

For example, an application may store in memory an array of a certain size and use a variable to indicate this size. If at some point the actual size of the array does not match what is stored in the size variable, then the assumption is broken, and the internal state of the application is invalid. This may, of course, cause severe problems. In our example, an out-of-bounds write—which could result in a crash or an arbitrary code execution if exploited by an attacker—is quite possible.

Furthermore, even if the program's internal data manipulation logic is flawless, its internal state still may become corrupted when it reads data from the outside. All external information entering the application, whether it be a command line parameter or a collection of bytes received through a network socket, must be properly verified and either accepted as valid or rejected as invalid.

If the program is completely correct, safe, and secure, then it should recognize and gracefully reject any invalid inputs. However, due to programming oversights or other anomalies, invalid inputs are not always caught at the frontier. Some may be accepted inside the program, invalidating its internal state. This is especially prone to happen when external data validation is not a trivial task.

Applications that handle complex structured inputs—communications using specific protocols or employing specific file formats for information storage, for example—are especially vulnerable.

This is where fuzz testing can be deployed with great effect. Fuzzers attempt to generate invalid, unexpected, or completely random data to feed a given program in the hope of discovering any holes in its input verification. Basically, their aim is to detect the situations when the program accepts an invalid input as valid.

Although there are different approaches to generating such inputs, many fuzzers skim along the valid/invalid input border. They attempt to generate inputs that are almost valid but contain some subtle invalidity or expose an obscure corner case.

FUZZING AND BEYOND

What is fuzzing? Continued

Fuzz testing

Whereas fuzzing has been defined as merely exercising a PUT with fuzz inputs, fuzz testing has the goal of verifying the fitness of the PUT against a specification. In general, fuzz testing can be defined as *a form of software testing that uses fuzzing*.

Manès, et al, consider fuzz testing to have the specific goal of finding security-related vulnerabilities, as this is its predominant application. Hence, their definition of fuzz testing is: *The use of fuzzing to test if a PUT violates a security policy*.[8]

Fuzzer

A fuzzer, also called a fuzzing engine, is: *A program that performs fuzz testing on a PUT*.[9]

Fuzz campaign

According to Manès et al, a fuzz campaign is: *A specific execution of a fuzzer on a PUT with a specific security policy*.[10]

Though, as they point out, "*fuzz testing can actually be used to test any security policy observable from an execution*," not just one specific security policy.[11]

Bug oracle

The term oracle may call to mind figures from Greek mythology like the priestess to Apollo at Delphi or the Sibyls, the oracles through which the gods were believed to speak. In fuzzing, the *bug oracle* (or simply, the oracle) is the device that determines what the PUT's response should be to a given fuzz input.

To reflect common fuzzing practice, Manès, et al, define a bug oracle as: *A program, perhaps as part of a fuzzer, that determines whether a given execution of the PUT violates a specific security policy*.[12]

Testing corner cases

A corner case is a test case that tests the program or system at the extreme limits of its intended inputs and conditions. This can include inputs that are outside the normal range of values, inputs that are specifically designed to stress the program or system, inputs that violate assumptions or constraints of the program or system, and inputs that would not be encountered in normal use.

Corner cases are important to test because they can reveal security vulnerabilities that may not be detected through normal testing methods. By testing these cases, software developers can ensure that their programs or systems behave correctly and securely even under unusual or unexpected conditions.

Typical uses of fuzzing

In a nutshell, fuzzing is used to expose flaws in a software program. Historically, it has been found extremely efficient in detecting safety and security issues, both in applications and operating systems.

While fuzzing can be used as a part of any general-purpose software testing program, it is most useful (and most used) in a cybersecurity context. It helps improve robustness against malicious penetration via unanticipated inputs.

In short, fuzzing can be used to detect all kinds of vulnerabilities, but it is most often used to uncover security vulnerabilities.

FUZZING AND BEYOND

What is fuzzing? Continued

This definition implies that a single oracle need not and often does not cover ALL security policies. We'll see later that this partial coverage of security policies (like CWEs) is a disadvantage. And while it is common for a bug oracle to be partial, that is not necessarily the case for all oracles.

Fuzz algorithm

Very simply, a fuzz algorithm is:

The algorithm implemented by a fuzzer[13]

These vary significantly from fuzzer to fuzzer, depending on the area within the fuzzing space a specific fuzzer addresses.

Fuzz configuration

A fuzz configuration of a fuzz algorithm comprises the parameter value(s) that control(s) the fuzz algorithm.[14]

The types of values in a fuzz configuration depend upon the type of the fuzz algorithm. Fuzzers typically maintain a collection of "seeds," and some fuzzers evolve the collection as the fuzz campaign progresses.[15]

A seed is a (commonly well-structured) input to the PUT used to generate test cases by modifying it. The collection of seeds maintained by a fuzzer is called a *seed pool*.[16]

Fuzzers are normally able to store some data within each configuration. A coverage-guided fuzzer, for example, may store the attained coverage in each configuration.[17]

Typical users

Who uses fuzzing? In general, potential users include anyone interested in detecting security vulnerabilities.

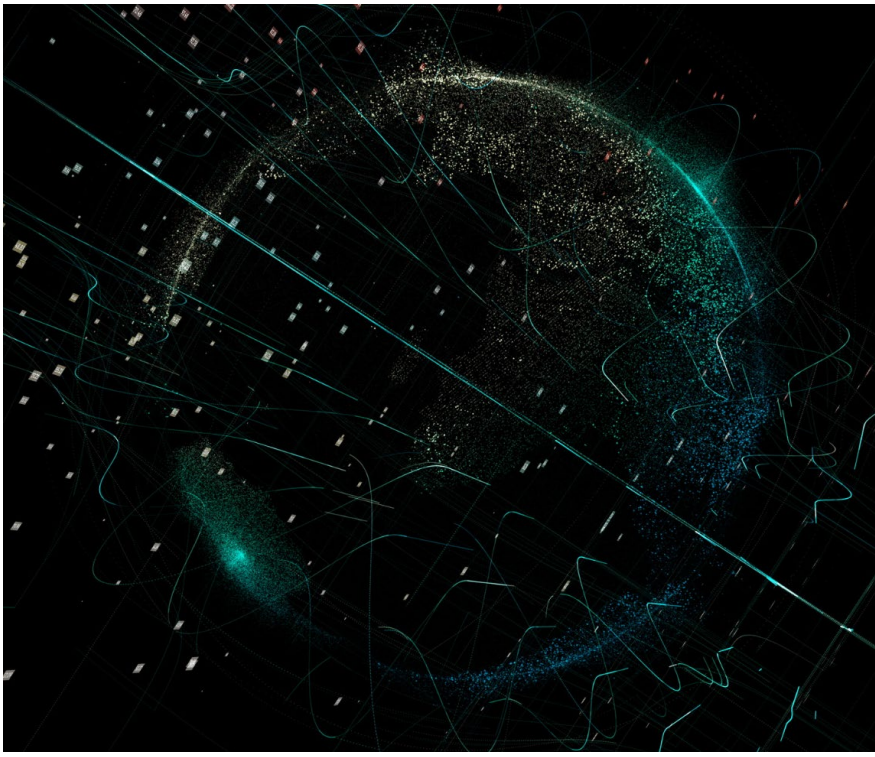
More specifically, typical users fall into three general categories.

Hackers (black hats) use fuzzers and fuzzing with malicious intentions. Their aim is to detect security vulnerabilities they can exploit, so they can take control of the software for financial or espionage motives.

The second group, ***software security researchers*** (white hats) frequently employ the same methods as black hats. They typically use fuzzing to discover security flaws in new software. They then report their findings to the software vendor, so the latter can correct the defect found before black hats can cause them any damage.

The third group includes ***software developers, penetration testers, and other software testers***. This group generally uses somewhat different tools and methods than black hats and white hats, because they have access to the source code. They need to do a far more thorough job than hackers, who only need to find one vulnerability they can exploit.

Next, we'll look at the differences between the various fuzzing tools and methods these groups use.



FUZZING AND BEYOND

Commonly Used Fuzzing Tools

Tools commonly used for fuzzing include fuzzing engines, fuzzing infrastructures and frameworks, and libraries.

Fuzzing engines

Fuzzing engines, commonly referred to as fuzzers, are not all created equal. They can be characterized along a number of lines, as the space is highly multi-dimensional. It is important to choose a fuzzer that is well-suited to your application. We will look at a number of ways in which the fuzzer space is segmented.

Need for source code

Fuzzing engines can be either compiler-based or binary-only.

Compiler-based fuzzers require access to the source code. They include a special compiler for the target programming language that adds lightweight instrumentation to the PUT when compiling it. That instrumentation typically collects coverage data during the fuzz campaign or provides data to the oracle function.

State-of-the-art fuzzers also use compilers to apply fuzzing-enhancing transformations that improve the execution speed of the PUT, enable easier penetration, and track interesting behaviors.[18]

Binary-only fuzzers are designed for situations where source code is unavailable. In practice, many fuzzing use cases are binary only, especially for security researchers working on closed-source, proprietary, or commercial software. Such fuzzers are restricted to binary instrumentation.[19]

Until very recently, available options for binary-only fuzzing have been unable to match the speed and transformation of their compiler counterparts, thus limiting their effectiveness.[20,21]

Awareness of program structure

Black-box fuzzers are unaware of the internal structure of the PUT. They observe only the target program's input/output behavior, treating it as a "black box" they can't see inside. Most early fuzzers were of this type. Some modern black-box fuzzers like Funfuzz[22] and Peach[23] take the structure of the PUT's inputs into account to generate more meaningful test cases without inspecting the source code.[24]

Black-box fuzzers are commonly used by hackers due to their ease of use and versatility. They are also used by white-hat security professionals who do not have access to the source code or who are assessing the likelihood of exploitation by hackers under such conditions.

White-box fuzzers generate test cases by analyzing the code structure of the PUT and the information they gather during execution. With this information, they are able to explore the target program's execution paths systematically.

The term "white-box fuzzing" was introduced by Patrice Godefroid to refer to fuzz testing that employs dynamic symbolic execution (DSE), a variant of symbolic execution.[25] The term is also used to describe fuzzers that employ taint analysis.[26] White-box fuzzing typically incurs much higher overhead than black-box fuzzing, partly because DSE implementations tend to employ SMT (Satisfiability Modulo Theories) solving and dynamic instrumentation. They require more work to set up and their processing is much slower.[27]

Grey-box fuzzers occupy a middle ground between the two extremes. Unlike black-box fuzzers, they can gather some information from inside the PUT to assess its structure and/or its executions. Unlike the white-box variety, grey-box fuzzers do not reason about the full semantics of the PUT. Instead, they tend to limit their investigation to performing some lightweight static analysis and/or gathering some dynamic execution data, like code coverage. Grey-box fuzzers aim to strike an effective balance between execution speed, ease of use, and ensuring broad test coverage.[28]

Coverage-guided grey-box fuzzing is probably the most successful fuzzing approach. This method adds a feedback loop to keep and mutate only the few test cases reaching new code coverage. The rationale behind it is that exhaustively exploring the target code will likely reveal more vulnerabilities. Coverage is collected via instrumentation inserted into the target program at compilation.[29] Widely successful coverage-guided grey-box fuzzers include AFL,[30] AFL++,[31] libFuzzer,[32] and honggFuzz[33].

How inputs are generated

Mutation-based fuzzers take the seeds (valid inputs) in their seed pool and generate collections of fuzz inputs by altering (mutating) them, mostly by bit manipulation, into forms that may be valid or invalid.

Generation-based fuzzers take the valid input structure provided to them, analyze it, and generate entirely new inputs that match the valid input structure.

Awareness of input structure

Dumb (unstructured) fuzzers produce completely random inputs that do not necessarily match the prescribed format of the expected input. Most early fuzzers were of this type. Due to their simplicity, dumb fuzzers can produce results with little work, but their coverage will be extremely limited. Such primitive fuzzers are unlikely to produce sufficient results to help ensure cybersecurity.

Through awareness of input structure, ***smart (structured) fuzzers*** can generate randomized inputs that are valid enough to pass program parser checks and penetrate deep into the program logic. These require more work to set up compared to dumb fuzzers since the user must define for the fuzzer the target program's input format, but they are far more likely to trigger edge cases and find vulnerabilities thanks to greater code coverage.

Types of inputs generated

Many fuzzers are optimized for fuzzing specific types of input formats, including:

- File
- Network
- Kernel I/O
- UI
- Web
- Thread (concurrency)

These specializations crosscut the other categories listed earlier.

Manès et al produced a fuzzer genealogy (Figure 1) that illustrates how the fuzzer space is subdivided along the lines just described.

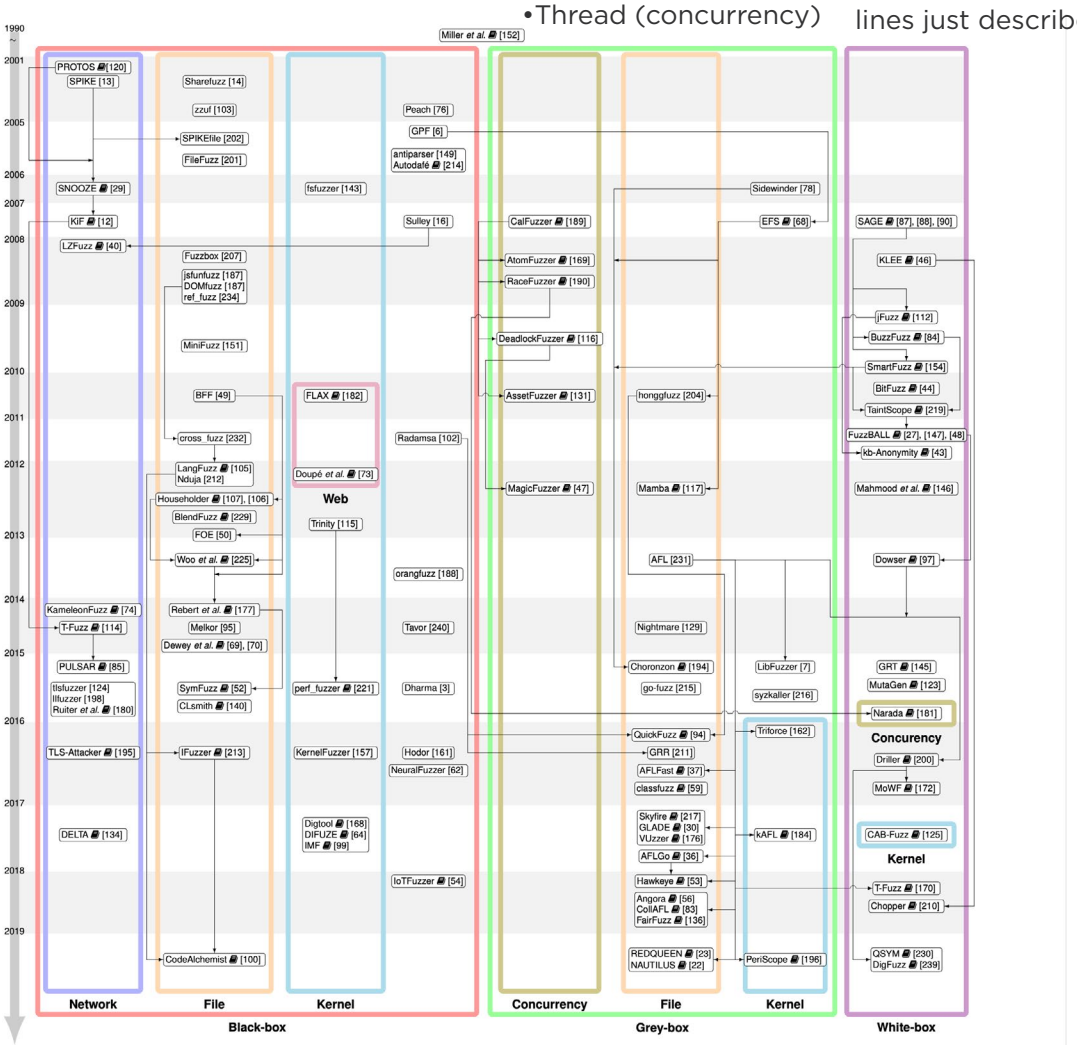


Figure 1: Genealogy tracing significant fuzzers' lineage back to Miller et al.'s seminal work. Each node in the same row represents a set of fuzzers appeared in the same year. A solid arrow from X to Y indicates that Y cites, references, or otherwise uses techniques from X. denotes that a paper describing the work was published.

Source: Manès, V., et al, The Art, Science, and Engineering of Fuzzing: A Survey, IEEE, October 2019.

FUZZING AND BEYOND

Fuzzing infrastructures and frameworks

A large-scale user of fuzzing to test its own software, Google has developed a number of scalable fuzzing tools. Among these, the most notable are ClusterFuzz and OSS-Fuzz.

ClusterFuzz is a distributed fuzzer execution environment and reporting tool. Essentially, it is a scalable fuzzing infrastructure. ClusterFuzz forms the fuzzing backend for OSS-Fuzz.[34]

OSS-Fuzz is a framework that combines fuzzers and provides scalable execution for the fuzzing of open-source software (OSS).[35]It combines modern fuzzing techniques with scalable, distributed execution. Along with ClusterFuzz, OSS-Fuzz supports the libFuzzer, AFL++, and Honggfuzz fuzzing engines in combination with a number of Sanitizers.[36]

Fuzzing libraries

To make "brute force testing less brutish," the ***FuzzDB*** Project has developed "*the first and most comprehensive open dictionary of fault injection patterns, predictable resource locations, and regex for matching server responses.*"[37]

FuzzDB contains lists of attack payload primitives (fuzz inputs) for fault injection testing that increase the likelihood of finding application security vulnerabilities. These are categorized by attack and, where appropriate, platform type.

Benefits of fuzzing with state-of-the-art fuzzing tools

Fuzz testing with state-of-the-art fuzzing tools offers software development organizations a number of significant benefits.

First, most fuzzing tools are relatively easy to use. This is especially true of black-box and grey-box fuzzers, which cover the vast majority of use cases.

Second, fuzzing rapidly expands your testing campaigns. It allows you to quickly and easily extend the scope of your unit tests, and it can be used in both unit testing and integration testing.

Next, fuzzing rapidly expands the code coverage of your testing. White-box and grey-box fuzzers typically include compilers that add code instrumentation that collects coverage data. In addition, the fuzzing algorithms of sophisticated fuzzers like AFL contain logic for directing coverage while limiting redundant cases and economizing campaigns. These facilities can quickly increase code coverage at the beginning of your test campaign by 60% to 80% compared to normal unit testing.

Finally, fuzzing can be easily scaled, parallelized, and combined with other techniques like static analysis and dynamic analysis.

While fuzzing offers several advantages, it is not without its limitations. It would be a mistake to view fuzzing as an exhaustive approach to ensuring code quality or security. It's important to know what those limitations are to specify the role fuzzing should play in the verification of a given application. We will examine several of those limitations next.

FUZZING AND BEYOND

Limitations of Fuzzing

Finding meaningful inputs can take significant time

Fuzzing generally requires a preliminary phase of running the fuzzer to "find" meaningful inputs. This can take significant time. In addition, you need to re-run fuzzers regularly as your code changes.

Of course, one's definition of "meaningful" will depend upon one's use case and objective. Many fuzzers do not solve the difficult problem of generating meaningful inputs, but simply generate a lot of them hoping that some will be interesting. The idea is that—since modern computers are so fast that it takes only a very small fraction of a second to run the code on the inputs—you might as well generate plenty of inputs and see which ones cause something interesting to happen.

As mentioned earlier, more sophisticated grey-box fuzzers will refine and improve their input seeds to explore promising areas and try to find more vulnerabilities. Nonetheless, fuzzing is clearly an iterative process. Any instance of the PUT can only be tested for one fuzz input at a time, and complex input profiles can present billions upon billions of possible test cases.

For example, a 256-bit input file offers 2^{256} (>1.15x 10^{77}) possible permutations. Even with fast computers and parallel processing, fuzzing an entire input space of that size would be prohibitive in terms of both time and cost.

Fuzzing engines generate fuzz inputs in a semi-random fashion. That means even the best fuzzing algorithms will produce a significant number of redundant inputs and accordingly run redundant tests.

What you can do—thanks to the instrumentation inserted by the fuzzer's compiler—is measure code coverage and the relevance of individual inputs based on the coverage they produce.

Coverage-based grey-box fuzzers can adjust their seed pools to reduce redundant tests. In general, however, it is very difficult to generate inputs that reach all the parts of the source code. Coverage tends to remain well below 100%.

Fuzz testing may be insufficient for testing embedded code

Embedded software is typically designed to run on specific embedded hardware, like an ARM RISC processor, for instance.

Good fuzz testing requires that you run as many fuzz inputs as possible. To make that cost-effective, those inputs need to be applied as quickly as possible. The power of fuzzing is measured in thousands of executions per second. 8000 executions per second is clearly better than 6000 ex/s. Due to this need for speed, fuzz testing is typically performed during unit testing in a host environment.

While finding and eliminating vulnerabilities in a host environment is good, if you only test in that environment, you have no chance of detecting vulnerabilities that occur only in the target architecture.

In other words, there are some vulnerabilities that will only occur when your code is running on your big-endian target architecture. If you fuzz test only on your little-endian desktop environment, you'll have no chance of finding them.

No amount of fuzzing will catch all undefined behavior

Since fuzzers generate fuzz inputs semi-randomly with lots of redundancy, they can't possibly generate all the possible values that make up your code's input space. As mentioned, for a non-trivial program, there are too many possible input combinations. Billions upon billions of them.

Also, fuzzers are not intended to detect all undefined behavior resulting from their inputs. The main purpose of a fuzzer is to generate sets of semi-random inputs. Secondary purposes of grey-box and white-box fuzzers include the measurement of code coverage, and the optimization of input sets to maximize code coverage while minimizing the number of inputs/executions required to achieve that coverage.

Fuzzers will indicate interesting behavior like program crashes caused by specific inputs, but they are not designed to detect every undefined behavior that might be lurking in your code.

In short, while it can help prove a program is *incorrect*, fuzzing cannot prove a program is *correct.*

Assumes the employment of an all-knowing bug oracle

At the opening of this white paper, we defined a ***bug oracle*** as a program, perhaps as part of a fuzzer, that *determines whether a given execution of the PUT violates a specific security policy.*[38]

Why the bug oracle is important

Since fuzzing generates arbitrary inputs in an automated manner, we do not know *a priori* how the target software is supposed to behave for those inputs. Is it expected to reject them as invalid? Is it expected to accept them as valid and produce a particular response to them? Either way, the fuzzer doesn't know.

The fuzzer only knows if a given input allows the program to execute or causes an execution failure (a crash). If the program executes, the fuzzer provides no indication of whether the program's response was correct or not. If the user is to recognize whether a given input results in behavior worthy of further human investigation, another component must be present. That component is the oracle.

The oracle tells the user if the target software appears to be behaving incorrectly for a given input.

Typical oracle implementations

One way to realize an oracle would be to write internal self-checks, called assertions, in the target software. Any input that causes one of these internal checks to fail makes the input interesting and worthy of investigation. Unfortunately, writing assertions for all possible security policies would be extremely time-consuming.

A second option is to rely on another implementation of the exact same functionality as a reference. One could then compare the results computed by the target software with those computed by the reference implementation. This practice is known as differential testing.

For differential testing to be effective, however, either the reference implementation must be of very high quality, or one must be prepared to find and fix vulnerabilities in the reference implementation as well as in the target software. An often-rediscovered fact when using differential testing is that vulnerabilities are found in the reference implementation as well as in the target software. This, too, is very time-consuming; in most cases, it doubles the amount of effort, as both the target and the reference must be developed and debugged.

Alternatively, one could hope that any incorrect behavior of the target software in response to a particular input will result in a recognizable failure like a crash or—in a memory-safe language like Ada—an uncaught exception. The latter is better than an undefined behavior but still not ideal, especially in a memory-unsafe language like C/C++ where the possibilities for undefined behavior abound.

Limitations of typical oracle implementations

In the case of C/C++ and other memory-unsafe languages, the frequently-used default oracles just described are only partial. They do not check for all of the possible types of undefined behavior that hackers might exploit; there is a vast array of these and some are more subtle than others. Furthermore, the results computed for a given input may vary depending on the memory layout defined at compilation.

In memory-unsafe languages, a given memory layout may cause a defect to result in an undefined behavior or not, or cause an undefined behavior to result in a crash or not. Consequently, one can't be sure that the result obtained during testing will be the same as the result for the same input obtained after deployment.

One way to palliate the problem would be to make sure the tested binary is exactly that which is intended for deployment and that all memory is allocated statically.

However, to detect more undefined behavior, what many users of fuzzing also like to do is allow a sanitizer to instrument the code generated during compilation with automatically-inserted additional checks. In this case, the memory layout of the binary executed during fuzzing is different from the memory layout of the uninstrumented binary intended for deployment.

So, not finding anything interesting during the execution of the (instrumented) binary during fuzzing does not mean that nothing interesting will happen during the execution of the (uninstrumented) binary after deployment.

Classic fuzzing is always a partial solution

Classic fuzzing will greatly expand your exploration of the total input space of your target program. You will likely find a lot of vulnerabilities that you would not normally find during a normal testing campaign.

Due to the limitations just described, however, classic fuzzing will always remain a partial solution, for three reasons. First, you cannot explore the entire input space due to its size. Second, you will only find vulnerabilities that occur in the specific memory allocation as dictated by the compilation performed by the fuzzer. Finally, you will only find vulnerabilities your instrumentation and analysis tools are designed to detect.

Because the compilation and memory allocation of your binary is likely to differ from what you explored in your fuzzing environment and may be affected by the order in which applications were loaded on the target hardware, your code will remain vulnerable. Hackers may be using black-box fuzzing to penetrate it on specific hardware platforms.

To overcome these limitations, you need a *complete* oracle rather than a partial one.

The perfect bug oracle

To ensure your code is safe from the exploits of malicious hackers, it is essential that you detect and eliminate undefined behavior (e.g. buffer overflows, non-initialized variables, invalid pointer usage, signed overflows, division by zero, etc.) in every execution path of your program.

To be clear, an undefined behavior is not the same as either an *implementation-defined behavior* or an *unspecified behavior*.[39]

Undefined behavior is a C/C++ language concept, defined in Annex J2 of the language: It consists in code constructs for which there are no requirements how the compiler will implement them, i.e. there is zero guarantee that the code will behave the same in different contexts (in particular the toolchain (and toolchain settings) used to generate the executable and the environment in which the executable will run). They therefore frequently cause random crashes or random program behavior. These types of problems are often very difficult to detect under standard laboratory testing conditions.

Undefined behavior is also very dangerous because it is a major angle of attacks on C/C++ code. Hackers exploit undefined behavior to remotely gain control of the software and achieve arbitrary code execution.

A great first step toward eliminating all undefined behavior in your code is to fuzz your code with the help of TrustInSoft Analyzer. Running fuzz inputs through TrustInSoft Analyzer allows you to formally verify the elimination of all undefined behavior that TrustInSoft Analyzer finds.

FUZZING AND BEYOND

TrustInSoft Analyzer

TrustInSoft Analyzer is a hybrid code Analyzer combining advanced static and dynamic analysis techniques together with Formal Methods to mathematically guarantee C/C++ code quality, reliability, security and safety. It has been designed to detect ALL undefined behavior in any execution path, and in combination with a fuzzer or any test drivers, with no false alarms.

TrustInSoft Analyzer can guarantee your fuzz testing results are valid for any compiler, any chosen set of compiler options, and any memory layout. In short, TrustInSoft Analyzer is a complete and perfect bug oracle for C/C++ code. It optimizes the fuzzing process.

Why formal verification is superior to classic fuzzing

While they will generate many more tests than normal testing, fuzzers typically do not detect vulnerabilities that don't cause the PUT to crash. What's more, a fuzzer will only explore the execution paths of one specific code compilation and memory layout.

In contrast, TrustInSoft Analyzer's mathematical analysis of the PUT using formal methods *guarantees* that every undefined behavior on every execution path explored by the provided fuzz inputs will be detected for every possible compilation and every possible memory layout.

Fuzzing with TrustInSoft Analyzer and AFL

Fuzzing with TrustInSoft Analyzer is fast and efficient because the tool integrates easily with AFL, one of the most popular coverage-based grey-box fuzzers. We like fuzzing with AFL for a number of reasons. Of these, three stand above the rest.

First, AFL is chainable to other tools. As the creator's documentation states, "The fuzzer generates superior, compact test corpora that can serve as a seed for more specialized, slower, or labor-intensive processes and testing frameworks. It is also capable of on-the-fly corpus synchronization with any other software."[40]

Second, AFL employs an efficient source code instrumentation to record the edge coverage of each execution of the program being tested and the coarse hit counts for each edge. It uses this information not only to generate seed files for new fuzz inputs but also in the implementation of a unique deduplication scheme that optimizes code coverage using a minimum set of inputs.[41]

Third, AFL uses a heuristic Evolutionary Algorithm (EA) to refine its seed pool based on branch coverage. This helps improve the odds that newly generated fuzz inputs help to increase coverage.[42]

All of these features help shorten the fuzzing cycle and save time.

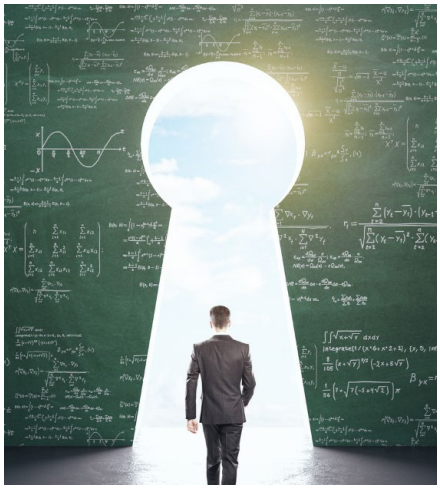
After running AFL to generate a set of test cases (input files), you just load those test cases along with your code into TrustInSoft Analyzer and run an analysis in the Analyzer's ***interpreter mode***. This is a simple operation—as easy as ordering a compilation of your code.

For each test case, TrustInSoft Analyzer will detect whether or not the execution depends upon the memory layout. If for some memory layout that input will cause an undefined behavior, the tool will detect that as well and generate a warning to that effect.

FUZZING AND BEYOND

Benefits of fuzzing with TrustInSoft Analyzer

Fuzzing with TrustInSoft Analyzer produces a number of important benefits.



1. Find more vulnerabilities.

TrustInSoft Analyzer can detect undefined behavior that typically remains unnoticed throughout standard unit and integration tests. You will be 100% sure you have no undefined behavior for the specific entry points defined by your test (fuzz) inputs.

2. No false alarms.

When running analysis on discrete inputs, TrustInSoft Analyzer generates no false alarms. All alarms raised correspond to real vulnerabilities, thanks to the use of formal methods. You'll waste no time investigating false positives.

3. Better validation of embedded code.

TrustInSoft Analyzer provides target emulation for embedded hardware platforms. Target emulation allows you to test your embedded code in an environment that closely resembles your target architecture. It helps you find vulnerabilities in embedded code that unit testing in a host environment cannot possibly reveal.

Out of the box, TrustInSoft Analyzer supports a number of common target platforms, including 32-bit ARM, 64-bit ARM, Power PC, RISC-V, and X86. If your hardware is more exotic, the emulator can easily be configured by adjusting a series of parameters. Everything that can change from one target platform to another is configurable in the Analyzer.

FUZZING AND BEYOND

Case Study

Fuzzing Wireshark in interpreter mode

A few years ago, as an experiment, we used TrustInSoft Analyzer to search for hidden vulnerabilities in ***Wireshark***, a popular network protocol analyzer and IP packet sniffer.

Target program

An open-source C application initially released in 1998, Wireshark had already been tested extensively over the course of two decades of use. It's also a huge application (5 million lines of code with plugins) that accepts a wide variety of input formats. Even if we had wanted to write new tests, the task would have been overwhelmingly large and extremely time-consuming. We wouldn't have known where to start.

Procedure

Instead, we downloaded existing Wireshark test scripts from GitHub, many of which were likely generated through fuzzing. There were 44 in total. We then fuzzed Wireshark with AFL for good measure, since we weren't sure how current our downloaded scripts were. Finally, we ran those test cases through TrustInSoft Analyzer in interpreter mode.

The overall process took in total 4 person days of effort, in order to understand the project structure and add a fuzzer on top of the existing test driver to generate 10k random data sets input values. Running the analysis itself took a few hours.

Results

In the end, TrustInSoft Analyzer found thirteen previously undiscovered vulnerabilities, including one undefined behavior deemed an exploitable vulnerability. Most of these defects had likely been latent within the application for years. Figure 2 summarizes our results.

The novelty of our approach was not in finding a way to generate better test cases. We used methods and inputs that had already been used by others. Instead, we leveraged a better way of taking advantage of those inputs. TrustInSoft Analyzer found subtle vulnerabilities that would be missed (and were missed) when simply executing the program with those same inputs.

TrustInSoft Analyzer in interpreter mode is the most complete oracle for detecting undefined behavior caused by inputs generated through fuzzing. It does not rely on one particular execution of the target code. Instead, it provides guarantees that apply for all possible executions of the target code, for any optimization level used during compilation, and for any memory configuration.

The only thing fuzzing in interpreter mode can't guarantee is the detection of undefined behavior along execution paths your fuzz inputs failed to explore. As explained earlier, the input space, in most cases, is simply too large for this to be practical. This is a limitation of fuzzing. It is, however, a limitation that can be overcome by going *beyond* fuzzing... with TrustInSoft Analyzer.

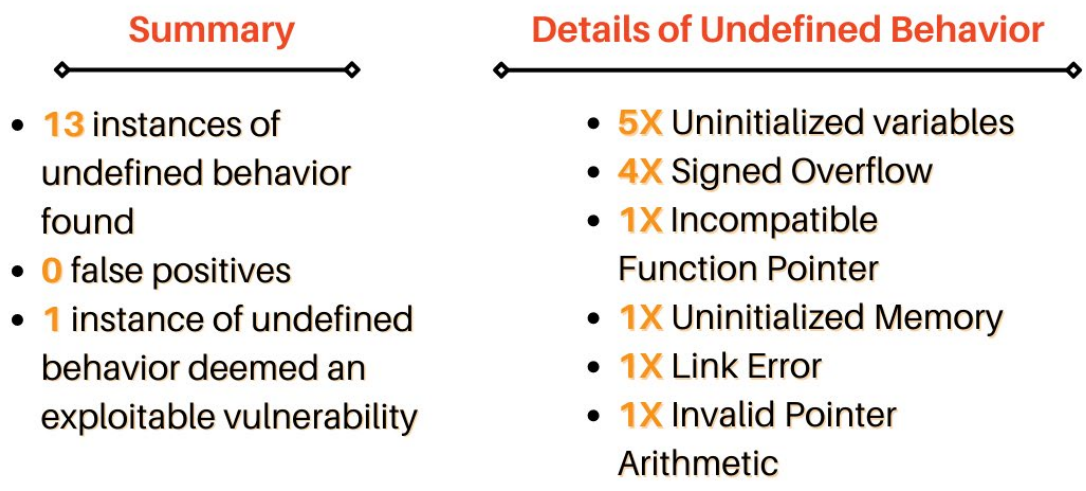


Figure 2: Results of Wireshark analysis using TrustInSoft Analyzer in interpreter mode.TrustInSoft Analyzer found subtle vulnerabilities that would be missed (and were missed) when simply executing the program with those same inputs.

FUZZING AND BEYOND

Beyond fuzzing: How to guarantee air-tight security

For applications where security is key and you want to be absolutely sure your code is free from vulnerabilities that can be exploited by hackers, you'll need to go beyond fuzzing.

While the first step we presented earlier (i.e. fuzzing with TrustInSoft Analyzer) is a great way to detect and eliminate more vulnerabilities than conventional testing or classic fuzzing can reveal, it cannot guarantee perfect coverage of all possible input vectors.

In interpreter mode, the Analyzer makes iterative test runs on the discrete input sets it has been given. But, as we've already seen, input space for most software applications—consisting of billions upon billions of possible combinations—is too immense to be covered completely through iterative testing. If you try, you'll never finish.

So, while a coverage-guided grey-box fuzzer like AFL can explore your program's input space efficiently, it can't explore it completely. Fuzzing alone cannot guarantee you've found every undefined behavior that may be lurking in your code.

For applications where assurance of a high level of cybersecurity is required, TrustInSoft Analyzer offers a more advanced solution. As we'll see shortly, this solution is a complement to fuzz testing that can guarantee perfect cybersecurity. We call this solution *exhaustive static analysis.*

Exhaustive static analysis

Exhaustive static analysis goes beyond fuzzing.

Instead of performing individual executions on individual inputs in an iterative fashion, it relies on a formal method called *abstract interpretation* to fully explore a program's input and execution space.

Abstract interpretation allows TrustInSoft Analyzer to perform abstract executions for the entire range of values defined by your input variables. It turns your existing tests with discrete inputs into a generalized test covering your code's entire input space.

For example, let's say you were testing a function of an integer `I` using values of `I` = –10 and `I` = +10. Thanks to the power of formal methods, you would be able to test this function over the full interval of values for the integer `I`, from -2^{31} to $2^{31}-1$, in a single test.

This input generalization works for all variable types in C/C++: integer, float, pointer, function pointer, etc. You can easily generate a generalized test from one of your existing tests with discrete inputs or from your API interface.

Thanks to the power of mathematics, exhaustive static analysis allows you to run the equivalent of billions upon billions of test cases simultaneously in a just a few seconds, in a single test run. It is guaranteed to detect all undefined behavior in your code, regardless of compilation optimization level or memory layout. Plus, once all the undefined behavior it detects have been eliminated, it provides formal proof that your code is totally free of exploitable vulnerabilities due to undefined behavior.

FUZZING AND BEYOND

Why fuzz with TrustInSoft Analyzer?

At this point, you may be asking yourself, "So, if you can eliminate every last vulnerability with exhaustive static analysis, why use fuzzing at all in the first step?"

The first thing to mention before we address this question is that there are some types of code that cannot be analyzed with discrete test inputs at all. Hence analyzing this type of software with exhaustive analysis and input generalization is the best approach. When a device starts up, the value of the registers can be any value, and the tester of a bootloader must take this into account. There is no point in testing this state of the device with discrete values as the test will not be representative enough. An exhaustive analysis is required to test the vulnerability of a bootloader.

Having said that, when it comes to the type of software where both fuzzing in interpreter mode or exhaustive analysis techniques are suitable, sometimes it can still be more efficient over the entire test campaign to start initially with Fuzzing in interpreter mode and then move on to the second stage, exhaustive static analysis.

Even though exhaustive static analysis generates far fewer false positives than other classic static analysis tools, it can generate some false positives due to the approximations made. In exhaustive static analysis, there are only two ways to determine which warnings are true vulnerabilities and which are false positives. One way is to manually investigate each warning, one by one. This method can be time-consuming. This is the same method that is being used with classic static analysis tools.

The second way, which we prefer, is to re-tune and repeat your analysis and then compare results.

By tuning, we mean adjusting the approximations of the acceptable and forbidden zones with slightly different parameters—to change their "shape," if you will in order to eliminate false positives.

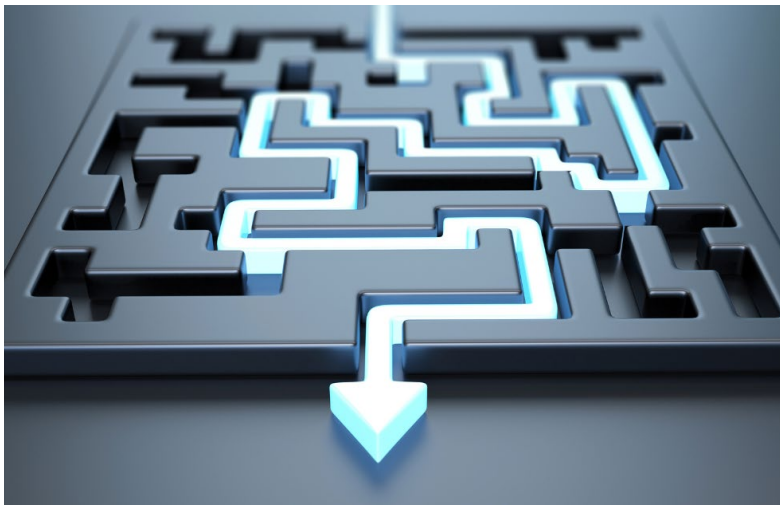
This is a far more efficient way to eliminate false positives.

Unfortunately, the more vulnerabilities you have in your target program, the more laborious this re-tune-and-compare process becomes. It becomes much more efficient after you've eliminated the more obvious vulnerabilities in your code.

Now, you'll remember that TrustInSoft Analyzer's interpreter mode will automatically run any set of inputs automatically and generate no false positives. That's why we called fuzzing with interpreter mode a great first step. By first fuzzing your program with AFL and running the resulting fuzz inputs through TrustInSoft Analyzer in interpreter mode, you can quickly detect and eliminate many true vulnerabilities before running the Analyzer in exhaustive static analysis mode.

Interpreter mode thins your vulnerability herd considerably. This greatly simplifies the task of re-tuning your analysis. It makes the elimination of the hard-to-find undefined behavior that exhaustive static analysis reveals much easier and quicker. Ultimately, it saves you a lot of time over the course of your debugging campaign.

You'll achieve formal proof that your code is 100% vulnerability-free much, much sooner.



The advantages of proceeding to exhaustive static analysis

In today's world, software providers need assurance of a high level of cybersecurity in their source code. To do this, there are several significant advantages to proceeding to exhaustive static analysis after fuzzing in interpreter mode.

First, it's exhaustive. You'll have peace of mind knowing you have found and removed every undefined behavior—every single vulnerability from your code.

Second, once you've removed all undefined behavior, TrustInSoft Analyzer provides a mathematical guarantee that you've removed every vulnerability from your code. This is formal proof you can use as evidence in reviews with security specialists, customers, and regulators.

Finally, having accomplished exhaustive static analysis once for a given program, you'll find it is much less work than fuzzing when you modify your code. You're now working from a much cleaner baseline. You simply re-run the analyses you've already set up.

To illustrate the power of exhaustive static analysis, we'll look at two more examples.

Case Studies

Seeing is believing: going beyond fuzzing or coupling fuzzing and analysis with TrustInSoft Analyzer allows for more powerful and robust results. Find out how TrustInSoft Analyzer helped secure the Goodix GT915 capactivie touchscreen driver and the Mbed TLS library in the following case studies:

Case study 1: Goodix GT915 capacitive touchscreen driver

The Goodix GT915 capacitive touchscreen driver is a hardware driver for a multi-touch screen sensor, used for mobile phones but also other touchscreens in cars, tablets etc..... It is an open-source driver that can sense and process multiple simultaneous touches on a touchscreen. Since the code is open source, hackers have access to it, and it makes it easier for them to find attack vectors.

The difficulty in designing and testing such a firmware device is that the hardware has no limitations. It can detect and provide input from up to 256 simultaneous touches.

How can one be sure the driver handles many simultaneous inputs flawlessly? Usually, people would use 1 or 2 fingers on a touch screen, and a resolute tester may try 5 or 10 fingers. Though what happens if there is a material defect, or if a hacker is able to simulate 256 simultaneous touches? Is the driver robust enough to cope?

It is extremely difficult to adequately test such a driver with hardware in the loop. There are just too many possible input combinations, and it is extremely difficult to exert a large number of touches simultaneously.

Conversely, in a hosted environment, you can't be certain of your test results due to the memory layout issues discussed earlier.

Thanks to the power of mathematics, with TrustInSoft Analyzer it was possible to determine, simulate and cascade the superset of all possible inputs, code values and behaviors.

Results

TrustInSoft Analyzer detected an undefined behavior that occurs when the hardware sends 256 simultaneous touches. This high number of inputs was too large for the software to handle and caused a buffer overflow.

TrustInSoft Analyzer was also able to confirm and guarantee the absence of undefined behavior due to a large array of common weakness enumeration (CWE)[43]: CWEs 119 to 127, 369, 415 416, 457 476, 562, 690 and 787.

Case study 2: Mbed TLS library

Mbed TLS (previously PolarSSL), a collaborative project managed by TrustedFirmware (formerly by Arm), is a library that implements the TLS protocol for encrypted secure communication over the internet.[44] The purpose of TLS is to provide strong guarantees of security. It lets you be certain that you know to whom you are talking and that other parties cannot eavesdrop on your conversation (by intercepting your packets as they are being transmitted).

Initially released in 2009, Mbed TLS is a widely used library that had already been tested extensively. Because it is such an important case from a cybersecurity perspective, we decided to analyze it.

The objective was to demonstrate absence of undefined behavior on several modules of the TLS library, namely: SSL Server and its submodules MD5 and SHA1 hashing, AES and RSA cryptography and MPI (Multi Precision Integer). The entire library is about 100K LoC, so each module was analyzed separately to simplify and speed up the work.

Results

During the analysis, TrustInSoft Analyzer detected a total of 9 issues, most notably instances of signed integer overflow and invalid pointer arithmetic—vulnerabilities that could result in the loss of the security properties one would expect from such a protocol. The code coverage resulting from the analysis was above 98%. The residual uncovered code was verified to be dead code.

After patches, TrustInSoft Analyzer was able to formally guarantee the absence of vulnerabilities covering 17 different CWEs. For this achievement, TrustInSoft was cited in a National Institute of Standards and Technology (NIST) report to the White House for having demonstrated that it can formally guarantee that no undefined behavior is present in a system.[45]

Conclusion

In today's hyper-connected, software-dependent world, exhaustive detection and elimination of undefined behavior is a must. Typically, these vulnerabilities are very difficult to detect under standard testing conditions and are the primary targets for exploitation by software hackers.

Hackers use fuzzing to find these weaknesses they can breach in software products. The developers of those products should be doing the same, so they can eliminate those weaknesses before hackers can exploit them.

Fuzzing is a great first step in eliminating security vulnerabilities in your code, but it won't find them all. Fuzzing is not exhaustive. It tests one input at a time. There is simply not enough time to test every possible input combination. The best you can hope for is that your pool of fuzz inputs is fairly representative of your program's possible execution paths.

What's more, hackers can still outsmart you with their own use of fuzzers. They only need to find one exploitable flaw in your application to breach it. You need to find every single one.

Fortunately, fuzzing results can be greatly improved by running fuzz inputs through TrustInSoft Analyzer. The Analyzer's interpreter mode, thanks to its use of a variety of mathematical formal methods, is designed to find more vulnerabilities than fuzzers or other static analyzers—indeed, to find every type of undefined behavior—while not generating any false positives. Fuzzing with TrustInSoft Analyzer gives you far greater confidence in the depth of your testing.

For applications where security is key and assurance of a high level of cybersecurity is required, it is necessary to go beyond fuzzing with TrustInSoft Analyzer.

Exhaustive static analysis—TrustInSoft Analyzer's generalization of inputs through the use of abstract interpretation—goes beyond fuzzing. Using advanced formal methods, exhaustive static analysis tests your program for all possible input combinations—not just a (hopefully) representative selection of them—across all possible compilations and memory layouts. It does so by solving your code as though it were a mathematical equation. It enables to exhaustively detect all undefined behavior and vulnerabilities which hackers may otherwise exploit. Once they are corrected, it provides a mathematical guarantee of their absence.

FUZZING AND BEYOND

References

1. ***Hyper Connectivity Market Forecast 2022-2030***, Precedence Research, September 2022.
2. ***Internet of Things Connectivity Market***, Emergen Research, June 2022.
3. Manès, V., et al, ***The Art, Science, and Engineering of Fuzzing: A Survey***, IEEE, October 2019.
4. Miller, B. P., Fredriksen, L., and So, B., ***An empirical study of the reliability of UNIX utilities***, Communications of the ACM, vol. 33, no. 12, pp. 32–44, 1990.
5. Manès, V., et al, ***The Art, Science, and Engineering of Fuzzing: A Survey***, IEEE, October 2019.
6. Ibid.
7. Ibid.
8. Ibid.
9. Ibid.
10. Ibid.
11. Schneider, F. B., ***Enforceable security policies***, ACM Transactions on Information System Security, vol. 3, no. 1, pp. 30–50, 2000.
12. Manès, V., et al, ***The Art, Science, and Engineering of Fuzzing: A Survey***, IEEE, October 2019.
13. Ibid.
14. Ibid.
15. Ibid.
16. Ibid.
17. Ibid.
18. Nagy, S., et al, ***Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing***, 30th Usenix Security Symposium, August 2021.
19. Ibid.
20. Ibid.
21. Pauley, E., et al, ***Performant Binary Fuzzing without Source Code using Static Instrumentation***, IEEE, October 2022.
22. Mozilla Security, Funfuzz, ***https://github.com/MozillaSecurity/funfuzz***.
23. GitLab, Peach Fuzzer, ***https://peachtech.gitlab.io/peach-fuzzer-community/***.
24. Manès, V., et al, ***The Art, Science, and Engineering of Fuzzing: A Survey,*** IEEE, October 2019.
25. Godefroid, P., ***Random testing for security: Blackbox vs. whitebox fuzzing, Proceedings of the International Workshop on Random Testing***, 2007.
26. Ganesh, V., Leek, T., Rinard, M., ***Taint-based directed whitebox fuzzing***, IEEE, May 2009.

FUZZING AND BEYOND

References

27. Manès, V., et al, ***The Art, Science, and Engineering of Fuzzing: A Survey***, IEEE, October 2019.

28. Ibid.

29. Nagy, S., et al, ***Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing***, 30th Usenix Security Symposium, August 2021.

30. Zalewski, M., ***American fuzzy lop***.

31. Advanced Fuzzing League ++, ***AFLPlusPlus***.

32. Serebryany, K., ***Continuous Fuzzing with libFuzzer and AddressSanitizer***, IEEE, November 2016.

33. Swiecki, R., ***honggfuzz***.

34. Google, ***ClusterFuzz***.

35. Google, ***OSS-Fuzz***.

36. Google, ***Sanitizers***.

37. FuzzDB Project, ***FuzzDB***.

38. Manès, V., et al, ***The Art, Science, and Engineering of Fuzzing: A Survey***, IEEE, October 2019.

39. Gruevski, P., ***Falsehoods programmers believe about undefined behavior***, predr.ag/blog, November 2022.

40. Zalewski, M., ***American fuzzy lop.***

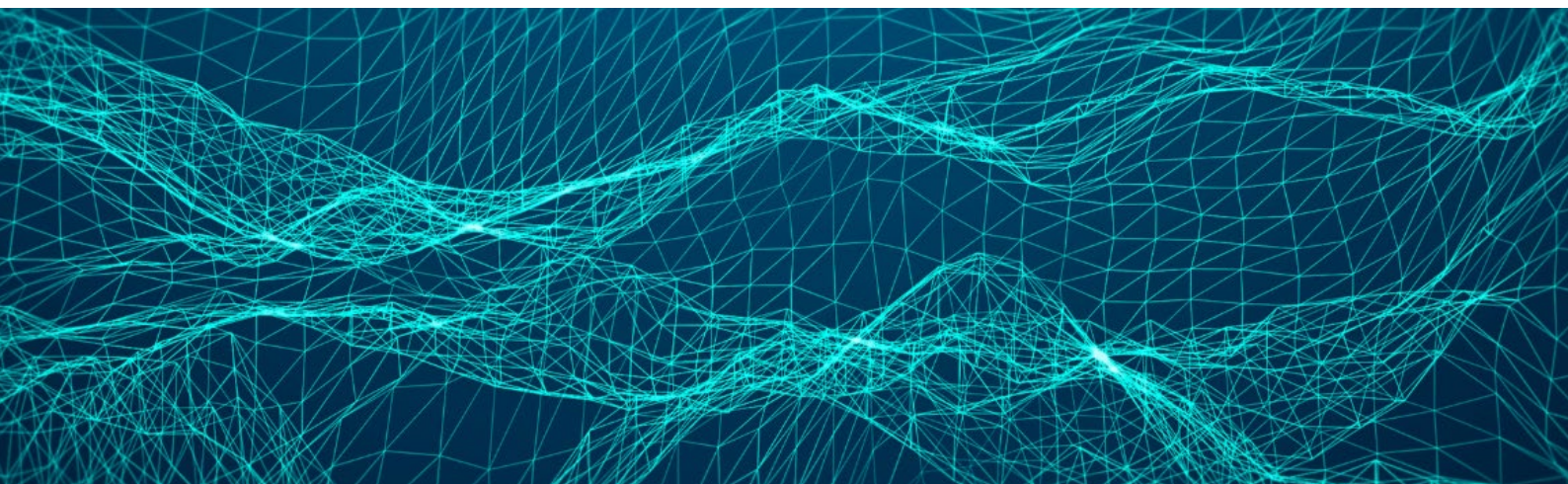
41. Manès, V., et al, ***The Art, Science, and Engineering of Fuzzing: A Survey***, IEEE, October 2019.

42. Ibid.

43. Common Weakness Enumeration, ***CWE List Version 4.10***, MITRE, October 2021.

44. TrustedFirmware, ***Mbed-TLS***.

45. Black, P.; Badger, L.; Guttman, B.; Fong, E.; ***Dramatically Reducing Software Vulnerabilities: Report to the White House Office of Science and Technology Policy; National Institute of Science and Technology (NIST)***, November 2016.



TrustInSoft Analyzer also provides the following business benefits:

1. Reduces overall software testing efforts and costs
2. Provides mathematical guarantees on software quality, robustness, cybersecurity and safety
    - Reduces risks linked to Time to market and meeting software deliverable milestones
    - Legal liability
    - Brand reputation
3. Enables differentiation through higher level of software quality, robustness, cybersecurity and safety.

Since the beginning, TrustInSoft Analyzer has been adopted by industry-leading companies around the world to ensure sound cybersecurity in their low-level code.

Founded in 2013, TrustInSoft developed a game-changing product for software code analysis. TrustInSoft Analyzer is a hybrid code analyzer that combines advanced static and dynamic analysis techniques together with Formal Methods to mathematically guarantee C/C++ code quality, reliability, security and safety. TrustInSoft has customers worldwide in the automotive, IoT, telecom, semiconductor, aeronautics and defense industries. TrustInSoft has received awards and recognition from the NIST, RSA and Linux Foundation.

To learn more about TrustInSoft Analyzer, visit ***trust-in-soft.com/product/trustinsoft-analyzer***.

If you'd like to speak with a TrustInSoft technical representative about how TrustInSoft Analyzer can meet your organization's specific needs, contact us by email at ***contact@trust-in-soft.com***.

contact@trust-in-soft.com
Phone : +33 1 84 06 43 91 or +1 (408) 829-5882

TRUST IN SOFT