



(11) **EP 3 639 141 B1**

(12) **EUROPEAN PATENT SPECIFICATION**

(45) Date of publication and mention of the grant of the patent:  
**11.01.2023 Bulletin 2023/02**

(21) Application number: **18730906.7**

(22) Date of filing: **24.05.2018**

(51) International Patent Classification (IPC):  
**G06F 11/36** <sup>(2006.01)</sup> **G06F 21/57** <sup>(2013.01)</sup>  
**G06N 3/04** <sup>(2006.01)</sup> **G06N 3/08** <sup>(2006.01)</sup>  
**G06N 5/00** <sup>(2006.01)</sup> **G06N 20/00** <sup>(2019.01)</sup>

(52) Cooperative Patent Classification (CPC):  
**G06N 20/00; G06F 11/3684; G06F 21/577;**  
**G06N 3/0445; G06N 3/08; G06N 5/003;**  
**G06F 11/3676; G06F 2221/033**

(86) International application number:  
**PCT/US2018/034256**

(87) International publication number:  
**WO 2018/231479 (20.12.2018 Gazette 2018/51)**

(54) **MACHINE LEARNING FOR CONSTRAINED MUTATION-BASED FUZZ TESTING**

MASCHINENLERNEN FÜR EINGESCHRÄNKTE MUTATIONSBASIERTE FUZZING-TESTS

APPRENTISSAGE AUTOMATIQUE POUR CONTRÔLE DE DONNÉES ALÉATOIRES BASÉ SUR UNE MUTATION CONTRAINTE

(84) Designated Contracting States:  
**AL AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HR HU IE IS IT LI LT LU LV MC MK MT NL NO PL PT RO RS SE SI SK SM TR**

(30) Priority: **15.06.2017 US 201762520173 P**  
**28.06.2017 US 201715636132**

(43) Date of publication of application:  
**22.04.2020 Bulletin 2020/17**

(73) Proprietor: **Microsoft Technology Licensing, LLC**  
**Redmond, WA 98052-6399 (US)**

(72) Inventors:  
• **RAJPAL, Mohit**  
**Redmond, Washington 98052-6399 (US)**  
• **BLUM, William**  
**Redmond, Washington 98052-6399 (US)**  
• **SINGH, Rishabh**  
**Redmond, Washington 98052-6399 (US)**

(74) Representative: **CMS Cameron McKenna Nabarro Olswang LLP**  
**Cannon Place**  
**78 Cannon Street**  
**London EC4N 6AF (GB)**

(56) References cited:  
**US-A1- 2011 302 455**

- **Fabian Beterke: "Distributed Evolutionary Fuzzing with Evofuzz", Sicherheit 2016, Lecture Notes in Informatics (LNI), 1 January 2016 (2016-01-01), pages 23-32, XP055499534, Bonn Retrieved from the Internet: URL:<https://subs.emis.de/LNI/Proceedings/Proceedings256/23.pdf> [retrieved on 2018-08-14]**
- **SHERRI SPARKS ET AL: "Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting", COMPUTER SECURITY APPLICATIONS CONFERENCE, 2007. ACSAC 2007. TWENTY-THIRD ANNUAL, IEEE, PAGE(S) 1 - 10 , 14 December 2007 (2007-12-14), XP002573215, ISBN: 978-0-7695-3060-4 Retrieved from the Internet: URL:<http://www.cs.ucf.edu/~czou/research/EvolutionaryInputCrafting-ACSAC07.pdf> [retrieved on 2010-03-15]**

Note: Within nine months of the publication of the mention of the grant of the European patent in the European Patent Bulletin, any person may give notice to the European Patent Office of opposition to that patent, in accordance with the Implementing Regulations. Notice of opposition shall not be deemed to have been filed until the opposition fee has been paid. (Art. 99(1) European Patent Convention).

**EP 3 639 141 B1**

**Description**

## BACKGROUND

5 **[0001]** A fuzzing algorithm is used to test software code by executing the code with various versions (mutations) of an input file and determining whether those versions of the input file cause the code to crash or to execute properly. Traditional fuzzing algorithms are parametrized by many parameters such as number of mutations, choosing locations for mutations, types of mutations etc., which are typically assigned random values.

## 10 BRIEF DESCRIPTION OF THE DRAWINGS

**[0002]** Some embodiments of the technology are illustrated, by way of example and not limitation, in the figures of the accompanying drawings.

15 FIG. 1 is a block diagram illustrating an example computer system in which constrained mutation-based fuzz testing may be implemented, in accordance with some embodiments.

FIGS. 2A-2B are a flow chart illustrating an example method for constrained mutation-based fuzz testing, in accordance with some embodiments.

FIG. 3 is a first data flow diagram for constrained mutation-based fuzz testing, in accordance with some embodiments.

20 FIG. 4 is a second data flow diagram for constrained mutation-based fuzz testing, in accordance with some embodiments.

FIG. 5 is a block diagram illustrating components of a machine able to read instructions from a machine-readable medium and perform any of the methodologies discussed herein, in accordance with some embodiments.

25 Fabian Beterke: "Distributed Evolutionary Fuzzing with Evofuzz", Sicherheit 2016, Lecture Notes in Informatics (LN1), 1 January 2016 (2016-01-01), pages 23-32, XP055499534, describes the design of a tool (*called Evofuzz*) that implements the technique of evolutionary (or coverage-guided) fuzzing in a scalable, distributed manner. SHERRI SPARKS ET AL: "Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting", COMPUTER SECURITY APPLICATIONS CONFERENCE, 2007. ACSAC 2007. TWENTY-THIRD ANNUAL, IEEE, PAGE(S) 1 - 10, 14 DECEMBER 2007 (2007-12-14) XP002573215, ISBN: 978-0-7695-3060-4, discloses an extension of traditional "black box" fuzz testing using a genetic algorithm based upon a Dynamic Markov Model fitness heuristic. This heuristic allows the authors to "intelligently" guide input selection based upon feedback concerning the "success" of past inputs that have been tried. US2011/302455 A1 discloses technologies for performing targeted, black-box fuzzing of input data for application testing. A dataflow tracing module traces an application while it reads and processes a set of template data to produce operation mapping data that maps data locations in the template data to operations performed by the application in processing the data at the location. The tracing is performed without requiring the application source code, knowledge of the syntactical structure of the input data, or specially instrumented binaries for the application. A fuzzing module is then utilized to target a specific operation or operations in the application by fuzzing data locations within the template data according to the operation mapping data until the desired outcome is achieved.

40

## SUMMARY

**[0003]** According to aspects of the present invention there is provided a system and medium as defined in the accompanying claims. The present disclosure generally relates to machines configured for executing fuzzing algorithms to test code, including computerized variants of such special-purpose machines and improvements to such variants, and to the technologies by which such special-purpose machines become improved compared to other special-purpose machines that provide technology for executing fuzzing algorithms to test code. In particular, the present disclosure addresses systems and methods for using machine learning techniques for constrained mutation-based fuzz testing.

## 50 DETAILED DESCRIPTION

**[0004]** The present disclosure describes, among other things, methods, systems, and computer program products that individually provide various functionality. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the various aspects of different embodiments of the present disclosure. It will be evident, however, to one skilled in the art, that the present disclosure may be practiced without all of the specific details.

**[0005]** As noted above, techniques for optimizing the parameters for fuzzing algorithms based on previous executions of the algorithms may be desirable. Techniques for optimizing parameters for fuzzing algorithms may be desirable to

make fuzzing algorithms more efficient. According to some aspects, the subject technology provides techniques for optimizing fuzzing algorithms. In general, a fuzzing algorithm receives an input file for code for testing, and mutates (e.g., changes one or more bytes of) the input file to determine if the mutation causes the code to crash. The information about which versions of the input file cause the code to crash is provided to developers for fixing the code. Some problems in current fuzzing algorithms include identifying which bytes in the input file should be mutated in order to increase the likelihood that a version of the input file that causes the code to crash will be discovered.

**[0006]** Throughout this document, the subject technology is described as being implemented with a single input file. However, the subject technology may be implemented with multiple input files, or input in a form other than a file. Unless explicitly stated otherwise, the phrase "input file" includes multiple files or input in form(s) different from a file.

**[0007]** Some aspects of the subject technology are directed to constrained mutation-based fuzz testing. Fuzz testing or "fuzzing" is the process of supplying a target executable with maliciously crafted input to cause it to crash, hang, or otherwise break. A target program is "fuzzable" if it accepts arbitrary input from an untrusted source. According to the claimed invention, a machine accesses an input file of code for testing, the input file comprising a plurality of bytes. The machine performs a plurality of runs of a fuzzing algorithm using the input file and the code for testing. Each run includes performing a mutation of at least one byte of the input file and determining which parts of the code under test were executed when the code was run with the mutated input file. The machine stores, for each run of the plurality of runs of the fuzzing algorithm, an indication of whether the mutation caused execution of a portion of the code for testing which was not executed prior to the mutation. The machine generates a heatmap of the input file based on the stored indications. The heatmap maps each of the plurality of bytes in the input file to a value indicating whether the mutation of the byte caused execution of the portion of the code for testing which was not executed prior to the mutation. The machine tailors the fuzzing algorithm based on the heatmap. The machine executes the tailored fuzzing algorithm to identify one or more versions of the input file for which the code for testing crashes. The machine provides, as a digital transmission, indicia of the one or more versions of the input file for which the code for testing crashes. In some cases, the digital transmission is provided to a client device or a display unit for display thereat.

**[0008]** Fuzz testing is one widely used automated testing technique that has been successful in automatically finding security vulnerabilities in complex software. The key idea in fuzzing is to continuously generate new inputs to stress-test a software under test to discover unexpected behaviors such as crashes, buffer overflows, or exceptions. Traditionally, the fuzzing techniques start with a set of seed input files, and then continuously transform these files to generate new promising inputs either by random mutations, constraint-solving, or using a set of manual heuristics. Since the input formats can be quite complex, it typically requires millions of mutations to generate a few meaningful new inputs, and therefore it can also be seen as a huge search problem to identify a good set of mutations that would lead to higher code coverage and crashes. In some aspects, the subject technology explores a learning technique that uses neural networks to learn patterns in the inputs based on past fuzzing explorations to guide the future fuzzing explorations.

**[0009]** There are three main types of fuzzing: i) Blackbox fuzzing, ii) Whitebox fuzzing, and iii) Greybox fuzzing. The blackbox fuzzers treat the binary under test as a black box with no internal inspection inside the binary. The whitebox fuzzers on the other hand also analyze the source code of the binary to generate input mutations to specifically target certain code fragments. The greybox fuzzers are somewhere in between where they perform limited source code inspection such as computing code coverage. Although all fuzzing techniques have different pros and cons, greybox fuzzing techniques based on random mutations have been used in the real-world because of their simplicity and efficiency.

**[0010]** FIG. 1 is a block diagram illustrating an example computer system 100 in which constrained mutation-based fuzz testing may be implemented, in accordance with some embodiments. The computer system 100 may include a single computer or multiple computers. As shown, the computer system 100 includes processor(s) 105, a network interface 110, and a memory 115. The processor(s) execute instructions stored in a computer-readable medium or a machine-readable medium (such as the memory 115) to receive input(s), process data stored in the memory 115 or other computer-readable medium or machine-readable medium, and provide output(s). The processor(s) may include a central processing unit (CPU), a graphics processing unit (GPU), and the like. The network interface 110 allows the computer system 100 to communicate via a network (e.g., the Internet, an intranet, a local area network, a wide area network, a wired network, a wireless network, and the like). The network interface 110 allows the computer system 100 to, among other things, receive input(s) and provide output(s) via the network. The network interface may include one or more network interface cards (NICs). The memory 115 stores data or instructions. The memory 115 may include long-term storage or short-term storage. As shown, the memory includes a fuzzing algorithm controller 120, a fuzzing algorithm 125, code for testing 130, an input file 135, a byte mutation success map 140, and a heatmap 135.

**[0011]** The code for testing 130 may include any code which may be executed by the processor(s) 105 and tested using a fuzzing algorithm. The code for testing 130 takes as input the input file 135 and processes the input file to generate an output. The input file 135 may be of any file type, such as PDF (portable document format), JPG, PNG, and the like. The code for testing 130 may be an application for processing (e.g., displaying, editing, or transmitting) files of the type associated with the input file 135. The input file 135 includes multiple bytes.

**[0012]** The fuzzing algorithm 125 takes the input file 135 for code for testing 130, and mutates the input file 135 to

determine if the mutation causes the code 130 to crash. The fuzzing algorithm 125 mutates one or more bytes in the input file 135 to cause unexpected behavior in an execution of the code for testing 130. The information about which versions of the input file cause the code 130 to crash is provided to developers for fixing the code. In some examples, the fuzzing algorithm 125 is a greybox or blackbox fuzzer which generates input, code coverage pairs. The fuzzing algorithm 125 randomly mutates bytes in the input file 135 and identifies mutations that cause the code 130 to crash. One example of a fuzzing algorithm, which may be used as the fuzzing algorithm 125 with some aspects of the subject technology, is AFL (American Fuzzy Lop), which is an open source random fuzzer.

**[0013]** The fuzzing algorithm controller 120, when executed by the processor(s) 105, causes the processor(s) 105 to access the input file 135 of code for testing 130. The processor(s) 105 perform multiple runs of the fuzzing algorithm 125 using the input file 135 and the code for testing 130. Each run includes performing a mutation of one or more bytes of the input file 135 and determining which parts of the code for testing 130 were executed when the code was run with the mutated input file 135. The processor(s) 105 store, in the byte mutation success map 140 and for each run of the plurality of runs of the fuzzing algorithm 125, an indication of whether the mutation caused execution of a portion of the code for testing 130 which was not executed prior to the mutation. The processor(s) 105 generate, based on the byte mutation success map 140, a heatmap 145 of the input file. In some examples, the heatmap 145 corresponds to the byte mutation success map 140 or some modification thereof. The heatmap 145 maps each of the plurality of bytes in the input file 135 to a value indicating whether the mutation of the byte caused execution of the portion of the code for testing 130 which was not executed prior to the mutation. The processor(s) 105 tailor the fuzzing algorithm 125 based on the heatmap 145. The processor(s) 105 execute the tailored fuzzing algorithm 125 to identify one or more versions of the input file 135 for which the code for testing 130 crashes. The processor(s) 105 provide, as a digital transmission (e.g., via the network interface 110 or to a display unit coupled with the computer system 100), indicia of the one or more versions of the input file 135 for which the code for testing 130 crashes. In some cases, the digital transmission is provided to a display unit or another machine for display thereat. More details of some examples of the operation of the fuzzing algorithm controller 120 are provided in conjunction with FIGS. 2A-2B.

**[0014]** FIGS. 2A-2B are a flow chart illustrating an example method 200 for constrained mutation-based fuzz testing, in accordance with some embodiments. As described herein, the method 200 may be implemented in the computer system 100 shown in FIG. 1. For example, during the implementation of the method 200, the processor(s) 105 may be running the fuzzing algorithm controller 120. However, the method 200 is not limited to being implemented in the computer system 100 and may be implemented in other system(s) having different component(s).

**[0015]** As shown in FIG. 2A, the method 200 begins at operation 210 where the computer system 100 accesses an input file 135 of code for testing 130. The input file 135 includes a plurality of bytes.

**[0016]** At operation 220, the computer system 100 performs a plurality of runs of the fuzzing algorithm 125 using the input file 135 and the code for testing 130. In each run, at operation 222, the computer system 100 performs a mutation of one or more bytes of the input file 135. In each run, at operation 224, the computer system 100 determines which parts of the code for testing were executed when the code 130 was run with the mutated input file 135.

**[0017]** At operation 230, the computer system 100 stores, for each run of the plurality of runs of the fuzzing algorithm, an indication of whether the mutation caused execution of a portion of the code for testing 130 which was not executed prior to the mutation. For example, a code execution tracking application may be used to track the parts of the code for testing that are executed. In some examples, the indications are stored in the byte mutation success map 140, which maps each byte of the input file 135 to an indication of the number of times its mutation led to additional code being executed and the number of times its mutation did not lead to additional code being executed.

**[0018]** As shown in FIG. 2B, at operation 240, the computer system 100 generates a heatmap 145 of the input file 135 based on the stored indications. The heatmap 145 maps each of the plurality of bytes in the input file 135 to a value indicating whether the mutation of the byte caused execution of the portion of the code for testing 130 which was not executed prior to the mutation. In some examples, the heatmap 145 may indicate TRUE or 1 if, in at least one instance, the mutation of the byte caused execution of the portion of the code for testing which was not executed prior to the mutation, and FALSE or 0 otherwise.

**[0019]** In some examples, the value in the heatmap 145 indicating whether the mutation of the byte caused execution of the portion of the code for testing 130 which was not executed prior to the mutation corresponds to a proportion, of runs where the byte had the mutation, which caused execution of the portion of the code for testing 130 which was not executed prior to the mutation. The proportion may be determined using the byte mutation success map 140. The heatmap 145 may be generated based on the byte mutation success map 140. In accordance with some examples, the heatmap 145 is a matrix identifying each byte of the input file 135 and the value indicating whether the mutation of that byte caused execution of the portion of the code for testing 130 which was not executed prior to the mutation.

**[0020]** At operation 250, the computer system 100 tailors the fuzzing algorithm 125 based on the heatmap 145. For example, as shown at operation 252, the computer system 100 adjusts the fuzzing algorithm 125 to perform mutations only on bytes associated, in the heatmap 145, with a value exceeding a threshold. Alternatively, the computer system 100 adjusts the fuzzing algorithm 125 to perform mutations only on bytes associated, in the heatmap 145, with a non-

zero value or a TRUE value. In some cases, the computer system 100 provides, as a digital transmission, an output of the tailored fuzzing algorithm. In some cases, the digital transmission is provided to a display unit or another computer for display thereat.

**[0021]** At operation 260, the computer system 100 executes the tailored fuzzing algorithm 125 to identify one or more versions of the input file 135 for which the code for testing 130 crashes. At operation 270, the computer system 100 provides, as a digital transmission, indicia of the one or more versions of the input file 135 for which the code for testing 130 crashes. The digital transmission may be provided, for example, to a display unit coupled with the computer system 100 or to another machine via a network to which the computer system 100 connects via the network interface 110.

**[0022]** Constrained mutation-based fuzz testing limits the search space of fuzzing while optimizing code coverage. The constrained mutation-based fuzzing constrains the search space by focusing on input, code coverage pairs. The heatmap and the machine learning model are provided. Fuzzing can be thought of as a discrete optimization problem with code coverage being the variable being optimized. In some examples, one possible goal of fuzzing is to maximize code coverage (the objective function) while limiting the number of total executions (the constraints) to a suitable minimum.

**[0023]** In some aspects of the subject technology, the size of the search space of the above discrete optimization problem is reduced, while maintaining reachability of optimal objective function value. This may transform the problem into a simpler, more tractable problem. This may yield higher objective function values subject to the same constraints.

**[0024]** Some aspects reduce the size of the search space by focusing mutations on only specific bytes in each input seed file. Some aspects ignore portions of the input file that are believed not to affect code coverage. By constraining the possible mutations, some aspects are able to more thoroughly optimize within the limited search space. This approach may result in higher code coverage.

**[0025]** In some examples, fuzz testing is challenging because crafting malicious input is difficult. For a fixed input and executable, most modifications to the input will cause no discernible change in execution behavior. This is because most of the input is indeed data for which all possible values are acceptable. The "interesting" parts of the input are typically non-data sections such as headers, checksums, magic values, or length fields. Modifying some or all of these bytes is likely to yield malicious input which causes the target program to misbehave unexpectedly.

**[0026]** Some aspects of the subject technology are directed to intelligently exploring the input space to maximize the ratio of malicious inputs that is generated. Unlike previous formulations, some aspects of the subject technology approach the above problem as a discrete optimization problem that may be solved through quantitative approaches.

**[0027]** However, "malicious input" is ill defined and difficult to optimize. Some aspects of the subject technology attempt to optimize a related metric, code coverage. Code coverage is defined as the portion of the code that is executed by at least one test case. In some cases, optimizing code coverage is feasible, and may be used to discover most malicious inputs which cause the binary to misbehave.

**[0028]** There are many conventional approaches to fuzz testing. Fuzzers, programs which fuzz test binaries, come in many flavors. They can be very simple, to very intelligent. The typical fuzzing scenario involves supplying a binary executable, and a set of input seed files to the fuzzer. These input seed files are possible inputs the binary could expect to parse. For example: if the binary is a PDF viewer, the seed files will contain many PDF documents exercising the various capabilities of the PDF format. The fuzzer's job is to mutate these seed files to cause unexpected behavior such as crashes, access violations, buffer overflows, and other exceptions in the binary.

**[0029]** A very simple fuzzer will randomly flip bits and bytes in each of the input seeds to cause unexpected behavior. This "random corruption" approach yields results because it is simple and efficient, and thus can be heavily parallelized. However, fuzzers can also be very intelligent.

**[0030]** SAGE<sup>®</sup>, a fuzzer developed by Microsoft Corporation in Redmond, Washington, operates backwards towards maximizing code coverage. Instead of exploring the input space, SAGE<sup>®</sup> explores the code coverage space. SAGE<sup>®</sup> begins by choosing some previously unreachable code block to explore. Next, by inspecting the assembly instructions of the binary, SAGE<sup>®</sup> formulates a set of satisfiability constraints necessitated to reach this code block. If the code block is theoretically reachable by some possible input, and the program itself is deterministic on the input, the reachability of this code block can be formulated as a set of satisfiability constraints on the input. These satisfiability constraints can be given to a SMT (satisfiability modulo theories) solver, to output an input which satisfies all the constraints. This algorithm is best illustrated with an example, shown in Algorithm 1, below.

#### Algorithm 1

**[0031]**

```
void top(char input[4])
{
    int cnt = 0;
    if(input[0] == 'b') cnt++;
```

```

if(input[1] == 'a') cnt++;
if(input[2] == 'd') cnt++;
if(input[3] == '!') cnt++;
if(cnt >= 4) crash(); }

```

5 **[0032]** The code section of Algorithm 1 is difficult to test for simple fuzzers. This is because four separate "if" checks are required on the input prior to the crash being triggered. The four character input has  $2^{32}$  possible configurations ( $2^8$  possible symbols for each byte raised to the fourth power for four bytes), of which only one will trigger the crash. The probability of a simple fuzzer finding the crash by randomly flipping bits or bytes is incredibly low: about 1 in 4 billion.

10 **[0033]** However, SAGE<sup>®</sup> is able to find the crash very easily by examining the code itself. As an example, first, SAGE<sup>®</sup> decides to explore the code block containing the crash() code. This requires the constraint of `cnt >= 4`. This is dependent on 4 other blocks incrementing `cnt`. These blocks themselves have constraints on `input[0]`, `input[1]`, `input[2]`, and `input[3]` respectively. These constraints are formalized and given to a SMT-solver. The solver is able to successfully resolve this set of constraints with the input "bad!" which triggers the crash code.

15 **[0034]** SAGE's<sup>®</sup> technology is able to reach more complex code paths by working backwards. This comes at a cost. Code inspection, constraint formulation, and constraint solving are inherently slower than random fuzzing. As such this fuzzing mechanism has different cost-benefit tradeoffs. Fuzzing with SAGE<sup>®</sup> can be more useful on code already fuzzed by a random fuzzer. If all the "low-hanging" bugs are discovered by a random fuzzer, then a sophisticated intelligent fuzzer such as SAGE<sup>®</sup> can discover more nuanced bugs. Random fuzzers and "intelligent" fuzzers (e.g. SAGE<sup>®</sup>) are often used in conjunction due to their different tradeoffs. Another possible approach is to invest prudently in many fuzzing technologies to best cover as many scenarios as possible.

20 **[0035]** Some aspects of the subject technology improve random fuzzers by restricting which bits or bytes are randomly changed. According to some aspects, random fuzzers can be easily improved without any negative tradeoffs. One such approach is to constrain the portions of the input files which are modified. The idea behind this approach is that most of the input is data for which all possible values are valid. Some aspects of the subject technology attempt to identify which part of the input file is data. Then, some aspects can reliably ignore the portions of the input that are data while still maintaining identical code reachability. In other words, some aspects reduce the input space that must be explored, while guaranteeing that the optimal solution lies within the constrained set. This simplifies the problem exponentially.

25 **[0036]** The approach described above has several benefits. The benefits include: simplicity because the approach is simple to implement, and almost any fuzzer can be augmented with it; performance because the approach has low overhead; effective because it can reliably improve random fuzzers; and intuitivity because it is easy to understand.

30 **[0037]** Algorithm 2 provides a slight modification of Algorithm 1.

```

Algorithm 2
void top(char input[4])
{
int cnt = 0;
if(input[0] == 'b') cnt++;
if(input[1] == 'd') cnt++;
if(cnt >= 2) crash(); }

```

35 **[0038]** The example of Algorithm 2 has one modification relative to the example of Algorithm 1: only the first two bytes are inspected to trigger the crash. This is similar to most file formats, as almost all bugs can be triggered with very few byte changes. In some cases, the random fuzzer may be augmented to ascertain that only the first two bytes of the input are relevant and should be mutated. As a result, even the random fuzzer may be able to find a crash in Algorithm 2. This knowledge constrains the search space from  $2^{32}$  to  $2^{16}$ , which is well within the reach of random fuzzing.

40 **[0039]** Further aspects of the subject technology are directed to a Machine Learning (ML) based approach to identify bytes which affect code coverage. This approach of the claimed invention adaptively finds which bytes to fuzz, and which bytes to ignore in arbitrary file formats. Some aspects use a supervised machine learning model to predict a "heatmap" function which denotes which bytes are worth fuzzing and which should be ignored. This model is trained and utilized in a tight online-learning loop.

45 **[0040]** The ML Model advises the execution of the binary by restricting which portions of the input file to fuzz, and the feedback from the execution informs which portions of the file yielded interesting executions. In some cases, the ML Model learns to ignore portions of the input for which no change has yielded new or interesting code coverage. This "veto" approach restricts the input bytes which are considered suitable for fuzzing by learning from previous executions.

50 **[0041]** During execution, a fuzzer takes an input file and mutates some number of bytes of that file. The byte mutations which increase code coverage are noted, and a code coverage bitmap is generated. A code coverage bitmap is a {0, 1} bit sequence of the same size as the binary being executed. The code coverage bitmap of an execution denotes

which instructions were executed at least once (1), or never (0).

[0042] It should be noted that any random fuzzer may be augmented with some of the techniques of the subject technology.

[0043] Some aspects of the subject technology define a metric called unidirectional hamming distance over bitmaps. The unidirectional exclusive-or operator  $\{\oplus_{\mu}\}$  is a bitwise operation of the truth table shown in Table 1.

**Table 1: Unidirectional Exclusive Or Truth Table, with:  $A \oplus_{\mu} B = C$ .**

A	B	C
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	FALSE
TRUE	TRUE	FALSE

[0044] The unidirectional exclusive or function is interested in whether a bit that was previously unset became set. In the overarching picture of bitmap coverage, reaching a piece of code which was previously unreachable is interesting for fuzzing purposes. Because some aspects aim to maximize code coverage, we are interested in mutations which cause some new piece of code to be executed.

[0045] Some aspects of the subject technology are directed to the idea of whether it is possible to use machine learning to learn a strategy for guiding the input mutations based on patterns in input bytes and code coverage information. More specifically, some aspects run the traditional fuzzing techniques for a limited time to obtain data regarding which mutations lead to new code coverage, and then use this data to learn a function to guide further input modifications towards generating promising new inputs. Although this technique is applicable to any fuzzing system, it may be instantiated on a blackbox or greybox fuzzer. The fuzzer performs random mutations to a set of seed input files, and maintains an input queue of promising new input files that lead to execution of new code paths. Since it may be difficult to precisely mutate the input files using random mutations, typically millions of newly generate inputs are discarded and only a handful of them (in the input queue) are considered for future mutations. The technique of some aspects aims to learn to guide this process of input generation to minimize the time spent on generating unpromising inputs, thereby increasing the chance of the fuzzer to cover new code paths.

[0046] Some aspects use different neural network architectures to learn a function that predicts the expected code coverage map given a set of input modifications. Since input files can be of varying lengths, architectures such as LSTM (Long Short Term Memory) and sequence2sequence with attention may be used. At fuzzing time, the learned function is used to predict a coverage map for the complete input file, which corresponds to likelihood of mutations of each byte leading to new code coverage. Some aspects then use the coverage map to prioritize the mutation locations. For training these functions, the fuzzing algorithm is first run on a subset of seed files for a limited time, and the training data for mutation-coverage information is obtained.

[0047] Fuzz testing is a dynamic analysis technique to discover software faults. Fuzz testing a target binary involves crafting malicious input which causes the binary to misbehave unexpectedly. This misbehavior could be in the form of a crash, buffer overflow, or unhandled exception. Many fuzzers exist which perform fuzz testing in various ways. Fuzzers are typically invoked on parsers which take input such as document viewers and media players. Security vulnerabilities for these executables are especially severe due to their propensity to receive untrusted user input. A fuzzing session is started with a small number of seed files: files which the target binary can expect to parse during normal operation. These seed files can contain validly formatted input, as well as input which triggers error handling cases. A fuzzer then mutates seed files to trigger latent bugs in the binary. One goal of the fuzzer is to find bugs, however it is difficult to optimize for this objective function. Fuzzers instead attempt to maximize code coverage, or exercise specific pieces of code in the hope of finding bugs.

[0048] Fuzzers come in varieties such as blackbox, greybox, and whitebox. Blackbox fuzzers treat the target binary as a black box with no binary inspection. In contrast, whitebox fuzzers analyze source code to generate mutations which target specific pieces of code. Greybox fuzzers draw a compromise between the two with limited binary instrumentation, typically in the form of code coverage inspection. While mutation strategies differ across fuzzers, a simple fuzzer can be thought to have the fuzzing algorithm shown in Algorithm 3.

Algorithm 3: Pseudocode description of a typical blackbox fuzzing algorithm.

[0049]

**Algorithm: Simple Fuzzing**

**begin**

**Input:** *Seed*

**Result:** *MaliciousInputs*

**for:** *iterations* ← 0 to *limit* **do**

*input* ← *Seed*

*mutations* ← RandInt(len(*Seed*))

**for** *mut* ← 0 to *mutations* **do**

*byte* ← RandInt(len(*Seed*))

mutate(*input*,*byte*)

**end**

*result* ← ExecuteBinary(*input*)

**if** *result* is *crash* **then**

Append *input* to *MaliciousInputs*

**end**

**end**

**end**

**[0050]** There is no clear best type of fuzzer. A blackbox fuzzer may outperform whitebox fuzzer on buggy code with many "low-hanging" bugs. Improvements to simple fuzzers are useful because, in practice, simple strategies find more bugs than complex strategies.

**[0051]** A fuzzing algorithm instruments source code during compilation to gain access to code coverage during execution. The fuzzing algorithm attempts to maximize code coverage. Its mutation strategy is simple: attempt many small localized changes to the seed files, as well as some stacking changes which mutate many locations in the input simultaneously. However, a strength of some fuzzing algorithms lies in their genetic algorithms. During execution of the target binary, the fuzzing algorithm may observe the code coverage that a mutated input induces. A mutated input is considered interesting if it induces some never before seen piece of code to be executed, or if it changes the frequency of execution of a previously seen code block. This is referred to as "input gain." The fuzzing algorithm may save mutated inputs. This may induce an input gain. The fuzzing algorithm may treat the mutated inputs as further seed files to mutate. This constant evolution of the seed pool helps reach many obscure code paths which require iterated small changes to an input. This pool is also frequently culled to pick the best seeds to mutate for even further input gain.

**[0052]** For training the model, the data elements of Table 2 from an instrumented fuzzer may be used. Note that these data elements are used in a greybox fuzzers. Greybox fuzzers operate by taking some input file, and repeatedly mutating it to produce new code coverage paths. Most greybox fuzzers require no additional instrumentation to generate these data pairs. Blackbox fuzzers can be easily augmented to generate code coverage information for the target binary.

**Table 2: Data elements from an instrumented fuzzer**

<i>x</i> :	The input file being fuzzed
<i>b</i> :	Code coverage bitmap yielded by executing the target binary on <i>x</i>
<i>x'</i> :	The mutated input file
<i>b'</i> :	Code coverage bitmap yielded by executing the target binary on <i>x'</i>

**[0053]** Quantitative techniques may be used to identify useful bytes to increase code coverage. Some aspects of the subject technology are directed to simple machine learning techniques which can identify useful bytes using only input and code coverage pairs.

**[0054]** Some aspects include a fuzzer, and a model which highlights useful bytes. During runtime, the fuzzer queries the model for each input and focuses mutations on the highlighted bytes.

**[0055]** FIG. 3 is a first data flow diagram 300 for constrained mutation-based fuzz testing, in accordance with some embodiments. The first data flow diagram 300 includes a model 310 (e.g., corresponding to the fuzzing algorithm controller 120), a fuzzer 320 (e.g., corresponding to the fuzzing algorithm 125), and a target binary 330 (e.g., corresponding to the input file 135). As shown in FIG. 3, the model 310 highlights worthwhile bytes for the fuzzer 320. The fuzzer 320 queries the input from the model 310. The fuzzer 320 fuzzes the target binary 330.

**[0056]** The model given an input file in sequence-of-bits format should ideally produce a sequence of useful positions to fuzz:  $\forall k . \{0, 1\}^k \rightarrow [k]^*$ . However, in practice, this is difficult for a model to learn. An approximation of the above may be provided with a heatmap function, shown in Equation 1.

$$f': \forall k . \{0, 1\}^k \rightarrow [0,1]^k$$

*Equation 1*

**[0057]** Equation 1 associates each position with the probability of a mutation yielding an input gain. Moreover the magnitude at each position divulges the relative efficacy of mutation yielding input gain.

**[0058]** As shown in FIG. 3, the fuzzer 320 queries this model 310 during augmented execution. Before mutating a given input for the first time, the model 310 is queried for this input. The resultant heatmap is used to guide mutation towards useful positions. In some examples, the phrase "useful positions" in an input file refers to heatmap positions with a non-zero quantity: positions which were involved in at least one useful mutation. Useful positions are defined in Equation 2

$$\sum (cutoff_{\beta}(f'(x)) \wedge (x \text{ XOR } x')) > \alpha$$

*Equation 2*

**[0059]** In Equation 2,  $x$  = input file to fuzz;  $x'$  = mutated input; XOR = bitwise exclusive-or;  $\alpha$  denotes some fixed cutoff parameter in  $[0,1]$ ;  $cutoff_{\beta}$  denotes the element-wise cutoff function on  $[0,1]^k$  returning 1 for each element greater than  $\beta$  and 0 otherwise, for some fixed parameter  $\beta \in [0,1]$ . It should be noted that some of the equations are example instantiations of some aspects of the subject technology, and not the subject technology itself. Some of the equations describe the engineering that makes some aspects of the subject technology work in practice.

**[0060]** Potential mutations which target no useful bytes are vetoed during augmented execution. Some aspects do not waste time executing the binary and analyzing results on mutated inputs which are unlikely to give input gain. One possible advantage of augmented execution comes from avoiding useless mutations.

**[0061]** In some cases, the proposed approach gives improvements on some target binaries. Furthermore the model requires only input, code coverage tuples for training. This approach may be easy to integrate with some blackbox or greybox fuzzers, which provide input and code coverage tuples.

**[0062]** Some aspects are directed to deciding how to train  $f'$  with limited data. It is clear that a lack of change in code coverage indicates mutations applied to useless bytes. In some cases, there is no straightforward method to determine useful bytes through input, code coverage tuples. A general framework for training based on some scoring of  $b, b'$  is shown in Equation 3.

$$f(x) = x \oplus x' \text{ iff } s(b, b') > \alpha$$

*Equation 3*

**[0063]** One result of the above approach is that useless mutations are scored lower than the useful mutations.  $f$  points to the useful bytes with scores over  $\alpha$ . A model learning  $f$  will receive multiple  $x, x'$  pairs in a supervised setting. To optimize some form of classification loss,  $f$  will learn to hedge its' bets as a singular  $x$  may have many mutations  $x'$  which yield good scores. To minimize the aggregate loss over this "one-to-many" relationship,  $f$  learns the mean of the binomial

variable  $x' = x'_0, x'_1, \dots, x'_k$ . This demonstrates the relative usefulness of flipping bytes at given positions. The bytes which are seldom involved in an input gaining mutation will be lower than some threshold  $\alpha$ . This may result in some binary categorization over useful bytes.

[0064] In principle, an effective incarnation of  $s(b, b')$  is difficult.  $s(b, b')$  must distinguish mutations which cause input gain. However, input gain depends on the presence of "never-before-seen" execution behavior. The sequential dependency on fuzzing history requires a function,  $s^*$  also conditioned on the same. A function of this type may be difficult for machine learning models to learn. Some aspects provide an approximation of  $s^*$ , as shown in Equation 4.

$$s(b, b') = b \oplus_{\mu} b'$$

Equation 4

[0065] In Equation 4,  $\oplus_{\mu}$  represents the Unidirectional Hamming Function or unidirectional exclusive-or function. The truth table for this function is shown in Table 1, above.

[0066] It should be noted that the Unidirectional Hamming Function is similar, but not identical, to a strict exclusive-or function. The Unidirectional Hamming Function used as a scoring function, which highlights code sections which are not executed in  $b$ , but are executed in  $b'$ . A small mutation in the header or magic bytes section of  $x$  induces a very sparse  $b'$ . It is unlikely that the Hamming distance metric is suitable due to this reason. The Unidirectional Hamming Function attempts to apply a "low-pass filter" to the noisy Hamming distance metric to highlight locations which are key for some code block.

[0067] An implementation of some aspects of the subject technology may include a model and augmentations to a fuzzing algorithm.

[0068] The fuzzing algorithm may be augmented for use with some aspects. The augmented fuzzing algorithm may query a neural network model with each input prior to fuzzing. The neural model categorize the input into useful and useless section with byte granularity. This query strategy is illustrated in FIG. 4.

[0069] FIG. 4 is a second data flow diagram 400 for constrained mutation-based fuzz testing, in accordance with some embodiments. As shown, the second data flow diagram 400 includes the augmented fuzzing algorithm 410 (corresponding to the fuzzer 320 or the fuzzing algorithm 125), the model 430 (corresponding to the model 310 or the fuzzing algorithm controller 120), the input 420 (e.g., corresponding to the target binary 330 or the input file 135), and the annotated input 440. The annotated input 440 highlights bytes which yield input gain (increases in code coverage), and may correspond to the input file 135 coupled with the heatmap 145. As shown in FIG. 4, the augmented fuzzing algorithm 410 queries the model 430 with the input 420. The model 430 provides to the augmented fuzzing algorithm 410, as a response to the query, the annotated input 440.

[0070] The categorization is used during fuzzing. The fuzzing strategy applies the localized changes shown in Table 3. It should be noted that that the mutations of Table 3 may be performed on sequential sections.

Table 3: Example Mutations

Bit flips: Mutate the input by flipping [1/2/4] bit(s) at a time.
Byte flips: Mutate the input by xor'ing [1/2/4] byte(s) with 0xFF.
Arithmetic mutations: Mutate the input by adding/subtracting interesting quantities at [1/2/4] byte granularities.
Interesting substitutions: Mutate the input by splicing an "interesting" value at [1/2/4] byte granularities.
Dictionary substitutions: Mutate the input by replacing bytes with user supplied "interesting" values. These may be longer than 4 bytes in length.

[0071] All the mutations of Table 3 are small and localized changes of which there are finitely many for a give input. After the conclusion of this phase, the fuzzing algorithm begins stacking many of these small localized changes which are non-local and of significant Hamming distance with respect to the input. The fuzzing algorithm may apply somewhere between two and 128 stacking changes chosen uniformly. The previously mentioned small localized changes form some of the base changes. In addition the mutations shown in Table 4 may also be applied.

Table 4: Example Mutations

Random byte assignment: Assign a random value to a random byte.
Delete bytes: Delete a section of the input file.

(continued)

Clone bytes: Append bytes to a section of the input file.
---

Overwrite bytes: Overwrite a section of the input file
--

5

**[0072]** Due to the location and context insensitive nature of fuzzing, most mutations yield no input gain. One goal of augmented fuzzing is to improve the hit-rate of mutations. Consulting the annotated input given by the model, some aspects can avoid mutations which are unlikely to give input gain. Some aspects implement a highly permissive veto approach to reject mutations which target no useful bytes.

10

**[0073]** In some aspects, a function from the family of Equation 1 (above) is to be learned. Due to the arbitrary length and sequential nature of the input, Recurrent Neural Network (RNN) may be used. Each input file is sequential data which is most likely parsed sequentially by the target binary. RNNs are able to count. This is useful in annotating file formats which contain header information at fixed offsets. RNNs have been used in Statistical Machine Translation, and this task is similar since binary file formats can be considered a form of language.

15

**[0074]** Some aspects implement Long Short-Term Memory (LSTM) as the base recurrent unit. LSTMs extend the memory capability of a recurrent unit to longer sequences. This is accomplished by a separate pathway for the flow of memory. LSTMs can also "forget" memories that have outlived their usefulness which allows more expressiveness for longer sequences. Recall that a recurrent unit is defined as shown in Equation 5.

20

$$h_t = f(x_t, h_{t-1})$$

**Equation 5**

25

**[0075]** An LSTM decomposes the overall framework of Equation 5 into the subcomponents of Equation 6-9.

$$f_t = \sigma(W_f \cdot [x_t, h_{t-1}] + b_f)$$

30

**Equation 6**

$$i_t = \sigma(W_i \cdot [x_t, h_{t-1}] + b_i)$$

35

**Equation 7**

$$C_t = f_t * C_{t-1} + i_t * \tanh(W_C \cdot [x_t, h_{t-1}] + b_C)$$

40

**Equation 8**

$$o_t = \tanh C_t * \sigma(W_o \cdot [x_t, h_{t-1}] + b_o)$$

45

**Equation 9**

**[0076]** The forget gate  $f_t$  and the input gate  $i_t$  control whether old memory is forgotten, and whether the current input is worth remembering. This interplay allows the memory information of LSTMs to persist through longer sequences.

50

**[0077]** The architectures of Table 5 may be used with the subject technology.

**Table 5. Architectures**

LSTM: A simple unidirectional LSTM.
-------------------------------------

Bidirectional LSTM: A bidirectional LSTM which sees the input backwards and forwards.
---

Seq2Seq: Sequence-to-sequence architecture.
---

Seq2Seq+Attn: Sequence to sequence with attention.
--

55

**[0078]** Bidirectional LSTMs see the input in backward and forward order, as shown in Equation 10.

$$h_t = \text{Merge} \left( \vec{f}(x_t, h_{t-1}), \vec{f}(x_t, h_{t-1}) \right)$$

5

*Equation 10*

**[0079]** The recurrence relation of Equation 10 is satisfied by two unidirectional LSTMs, one in each forward and backward direction. Given a length n sequence, to compute the values for timestep t, the forward LSTM's  $h_{t-1}$  and the backward LSTM's  $h_{n-t-1}$  are used in conjunction. The Merge function can be one of many functions combining two like sized vectors such as sum, multiply, or concatenate. Some aspects use the sum function for one layer Bidirectional LSTMs, and the concatenate function for two layer or more LSTMs.

**[0080]** In some aspects, one layer bidirectional LSTMs uses a sum merge function, while two layer bidirectional LSTM use a concatenate function. The second layer of the two layer bidirectional LSTM is a unidirectional LSTM. Seq2Seq and Seq2Seq+Attn are comprised of one encoding and one decoding layer. The encoding layer is a bidirectional LSTM which was merged using the concatenate function.

## COMPONENTS AND LOGIC

**[0081]** Certain embodiments are described herein as including logic or a number of components or mechanisms. Components may constitute either software components (e.g., code embodied on a machine-readable medium) or hardware components. A "hardware component" is a tangible unit capable of performing certain operations and may be configured or arranged in a certain physical manner. In various example embodiments, one or more computer systems (e.g., a standalone computer system, a client computer system, or a server computer system) or one or more hardware components of a computer system (e.g., a processor or a group of processors) may be configured by software (e.g., an application or application portion) as a hardware component that operates to perform certain operations as described herein.

**[0082]** In some embodiments, a hardware component may be implemented mechanically, electronically, or any suitable combination thereof. For example, a hardware component may include dedicated circuitry or logic that is permanently configured to perform certain operations. For example, a hardware component may be a special-purpose processor, such as a Field-Programmable Gate Array (FPGA) or an Application Specific Integrated Circuit (ASIC). A hardware component may also include programmable logic or circuitry that is temporarily configured by software to perform certain operations. For example, a hardware component may include software executed by a general-purpose processor or other programmable processor. Once configured by such software, hardware components become specific machines (or specific components of a machine) uniquely tailored to perform the configured functions and are no longer general-purpose processors. It will be appreciated that the decision to implement a hardware component mechanically, in dedicated and permanently configured circuitry, or in temporarily configured circuitry (e.g., configured by software) may be driven by cost and time considerations.

**[0083]** Accordingly, the phrase "hardware component" should be understood to encompass a tangible record, be that a record that is physically constructed, permanently configured (e.g., hardwired), or temporarily configured (e.g., programmed) to operate in a certain manner or to perform certain operations described herein. As used herein, "hardware-implemented component" refers to a hardware component. Considering embodiments in which hardware components are temporarily configured (e.g., programmed), each of the hardware components need not be configured or instantiated at any one instance in time. For example, where a hardware component comprises a general-purpose processor configured by software to become a special-purpose processor, the general-purpose processor may be configured as respectively different special-purpose processors (e.g., comprising different hardware components) at different times. Software accordingly configures a particular processor or processors, for example, to constitute a particular hardware component at one instance of time and to constitute a different hardware component at a different instance of time.

**[0084]** Hardware components can provide information to, and receive information from, other hardware components. Accordingly, the described hardware components may be regarded as being communicatively coupled. Where multiple hardware components exist contemporaneously, communications may be achieved through signal transmission (e.g., over appropriate circuits and buses) between or among two or more of the hardware components. In embodiments in which multiple hardware components are configured or instantiated at different times, communications between such hardware components may be achieved, for example, through the storage and retrieval of information in memory structures to which the multiple hardware components have access. For example, one hardware component may perform an operation and store the output of that operation in a memory device to which it is communicatively coupled. A further hardware component may then, at a later time, access the memory device to retrieve and process the stored output. Hardware components may also initiate communications with input or output devices, and can operate on a resource

(e.g., a collection of information).

**[0085]** The various operations of example methods described herein may be performed, at least partially, by one or more processors that are temporarily configured (e.g., by software) or permanently configured to perform the relevant operations. Whether temporarily or permanently configured, such processors may constitute processor-implemented components that operate to perform one or more operations or functions described herein. As used herein, "processor-implemented component" refers to a hardware component implemented using one or more processors.

**[0086]** Similarly, the methods described herein may be at least partially processor-implemented, with a particular processor or processors being an example of hardware. For example, at least some of the operations of a method may be performed by one or more processors or processor-implemented components. Moreover, the one or more processors may also operate to support performance of the relevant operations in a "cloud computing" environment or as a "software as a service" (SaaS). For example, at least some of the operations may be performed by a group of computers (as examples of machines including processors), with these operations being accessible via a network (e.g., the Internet) and via one or more appropriate interfaces (e.g., an API).

**[0087]** The performance of certain of the operations may be distributed among the processors, not only residing within a single machine, but deployed across a number of machines. In some example embodiments, the processors or processor-implemented components may be located in a single geographic location (e.g., within a home environment, an office environment, or a server farm). In other example embodiments, the processors or processor-implemented components may be distributed across a number of geographic locations.

## EXAMPLE MACHINE AND SOFTWARE ARCHITECTURE

**[0088]** The components, methods, applications, and so forth described in conjunction with FIGS. 1-4 are implemented in some embodiments in the context of a machine and an associated software architecture. The sections below describe representative software architecture(s) and machine (e.g., hardware) architecture(s) that are suitable for use with the disclosed embodiments.

**[0089]** Software architectures are used in conjunction with hardware architectures to create devices and machines tailored to particular purposes. For example, a particular hardware architecture coupled with a particular software architecture will create a mobile device, such as a mobile phone, tablet device, or so forth. A slightly different hardware and software architecture may yield a smart device for use in the "internet of things," while yet another combination produces a server computer for use within a cloud computing architecture. Not all combinations of such software and hardware architectures are presented here, as those of skill in the art can readily understand how to implement the disclosed subject matter in different contexts from the disclosure contained herein.

**[0090]** FIG. 5 is a block diagram illustrating components of a machine 500, according to some example embodiments, able to read instructions from a machine-readable medium (e.g., a machine-readable storage medium) and perform any one or more of the methodologies discussed herein. Specifically, FIG. 5 shows a diagrammatic representation of the machine 500 in the example form of a computer system, within which instructions 516 (e.g., software, a program, an application, an applet, an app, or other executable code) for causing the machine 500 to perform any one or more of the methodologies discussed herein may be executed. The instructions 516 transform the general, non-programmed machine into a particular machine programmed to carry out the described and illustrated functions in the manner described. In alternative embodiments, the machine 500 operates as a standalone device or may be coupled (e.g., networked) to other machines. In a networked deployment, the machine 500 may operate in the capacity of a server machine or a client machine in a server-client network environment, or as a peer machine in a peer-to-peer (or distributed) network environment. The machine 500 may comprise, but not be limited to, a server computer, a client computer, PC, a tablet computer, a laptop computer, a netbook, a personal digital assistant (PDA), an entertainment media system, a cellular telephone, a smart phone, a mobile device, a wearable device (e.g., a smart watch), a smart home device (e.g., a smart appliance), other smart devices, a web appliance, a network router, a network switch, a network bridge, or any machine capable of executing the instructions 516, sequentially or otherwise, that specify actions to be taken by the machine 500. Further, while only a single machine 500 is illustrated, the term "machine" shall also be taken to include a collection of machines 500 that individually or jointly execute the instructions 516 to perform any one or more of the methodologies discussed herein.

**[0091]** The machine 500 may include processors 510, memory/storage 530, and I/O components 550, which may be configured to communicate with each other such as via a bus 502. In an example embodiment, the processors 510 (e.g., a Central Processing Unit (CPU), a Reduced Instruction Set Computing (RISC) processor, a Complex Instruction Set Computing (CISC) processor, a Graphics Processing Unit (GPU), a Digital Signal Processor (DSP), an ASIC, a Radio-Frequency Integrated Circuit (RFIC), another processor, or any suitable combination thereof) may include, for example, a processor 512 and a processor 514 that may execute the instructions 516. The term "processor" is intended to include multi-core processors that may comprise two or more independent processors (sometimes referred to as "cores") that may execute instructions contemporaneously. Although FIG. 5 shows multiple processors 510, the machine 500 may

include a single processor with a single core, a single processor with multiple cores (e.g., a multi-core processor), multiple processors with a single core, multiple processors with multiples cores, or any combination thereof.

**[0092]** The memory/storage 530 may include a memory 532, such as a main memory, or other memory storage, and a storage unit 536, both accessible to the processors 510 such as via the bus 502. The storage unit 536 and memory 532 store the instructions 516 embodying any one or more of the methodologies or functions described herein. The instructions 516 may also reside, completely or partially, within the memory 532, within the storage unit 536, within at least one of the processors 510 (e.g., within the processor's cache memory), or any suitable combination thereof, during execution thereof by the machine 500. Accordingly, the memory 532, the storage unit 536, and the memory of the processors 510 are examples of machine-readable media.

**[0093]** As used herein, "machine-readable medium" means a device able to store instructions (e.g., instructions 516) and data temporarily or permanently and may include, but is not limited to, random-access memory (RAM), read-only memory (ROM), buffer memory, flash memory, optical media, magnetic media, cache memory, other types of storage (e.g., Erasable Programmable Read-Only Memory (EEPROM)), and/or any suitable combination thereof. The term "machine-readable medium" should be taken to include a single medium or multiple media (e.g., a centralized or distributed database, or associated caches and servers) able to store the instructions 516. The term "machine-readable medium" shall also be taken to include any medium, or combination of multiple media, that is capable of storing instructions (e.g., instructions 516) for execution by a machine (e.g., machine 500), such that the instructions, when executed by one or more processors of the machine (e.g., processors 510), cause the machine to perform any one or more of the methodologies described herein. Accordingly, a "machine-readable medium" refers to a single storage apparatus or device, as well as "cloud-based" storage systems or storage networks that include multiple storage apparatus or devices. The term "machine-readable medium" excludes signals per se.

**[0094]** The I/O components 550 may include a wide variety of components to receive input, provide output, produce output, transmit information, exchange information, capture measurements, and so on. The specific I/O components 550 that are included in a particular machine will depend on the type of machine. For example, portable machines such as mobile phones will likely include a touch input device or other such input mechanisms, while a headless server machine will likely not include such a touch input device. It will be appreciated that the I/O components 550 may include many other components that are not shown in FIG. 5. The I/O components 550 are grouped according to functionality merely for simplifying the following discussion and the grouping is in no way limiting. In various example embodiments, the I/O components 550 may include output components 552 and input components 554. The output components 552 may include visual components (e.g., a display such as a plasma display panel (PDP), a light emitting diode (LED) display, a liquid crystal display (LCD), a projector, or a cathode ray tube (CRT)), acoustic components (e.g., speakers), haptic components (e.g., a vibratory motor, resistance mechanisms), other signal generators, and so forth. The input components 554 may include alphanumeric input components (e.g., a keyboard, a touch screen configured to receive alphanumeric input, a photo-optical keyboard, or other alphanumeric input components), point based input components (e.g., a mouse, a touchpad, a trackball, a joystick, a motion sensor, or another pointing instrument), tactile input components (e.g., a physical button, a touch screen that provides location and/or force of touches or touch gestures, or other tactile input components), audio input components (e.g., a microphone), and the like.

**[0095]** In further example embodiments, the I/O components 550 may include biometric components 556, motion components 558, environmental components 560, or position components 562, among a wide array of other components. For example, the biometric components 556 may include components to detect expressions (e.g., hand expressions, facial expressions, vocal expressions, body gestures, or eye tracking), measure biosignals (e.g., blood pressure, heart rate, body temperature, perspiration, or brain waves), measure exercise-related metrics (e.g., distance moved, speed of movement, or time spent exercising) identify a person (e.g., voice identification, retinal identification, facial identification, fingerprint identification, or electroencephalogram based identification), and the like. The motion components 558 may include acceleration sensor components (e.g., accelerometer), gravitation sensor components, rotation sensor components (e.g., gyroscope), and so forth. The environmental components 560 may include, for example, illumination sensor components (e.g., photometer), temperature sensor components (e.g., one or more thermometers that detect ambient temperature), humidity sensor components, pressure sensor components (e.g., barometer), acoustic sensor components (e.g., one or more microphones that detect background noise), proximity sensor components (e.g., infrared sensors that detect nearby objects), gas sensors (e.g., gas detection sensors to detect concentrations of hazardous gases for safety or to measure pollutants in the atmosphere), or other components that may provide indications, measurements, or signals corresponding to a surrounding physical environment. The position components 562 may include location sensor components (e.g., a Global Position System (GPS) receiver component), altitude sensor components (e.g., altimeters or barometers that detect air pressure from which altitude may be derived), orientation sensor components (e.g., magnetometers), and the like.

**[0096]** Communication may be implemented using a wide variety of technologies. The I/O components 550 may include communication components 564 operable to couple the machine 500 to a network 580 or devices 570 via a coupling 582 and a coupling 572, respectively. For example, the communication components 564 may include a network interface

component or other suitable device to interface with the network 580. In further examples, the communication components 564 may include wired communication components, wireless communication components, cellular communication components, Near Field Communication (NFC) components, Bluetooth® components (e.g., Bluetooth® Low Energy), Wi-Fi® components, and other communication components to provide communication via other modalities. The devices 570

5 may be another machine or any of a wide variety of peripheral devices (e.g., a peripheral device coupled via a USB).  
**[0097]** Moreover, the communication components 564 may detect identifiers or include components operable to detect identifiers. For example, the communication components 564 may include Radio Frequency Identification (RFID) tag reader components, NFC smart tag detection components, optical reader components, or acoustic detection components (e.g., microphones to identify tagged audio signals). In addition, a variety of information may be derived via the communication components 564, such as location via Internet Protocol (IP) geolocation, location via Wi-Fi® signal triangulation, location via detecting an NFC beacon signal that may indicate a particular location, and so forth.

10  
**[0098]** In various example embodiments, one or more portions of the network 580 may be an ad hoc network, an intranet, an extranet, a virtual private network (VPN), a local area network (LAN), a wireless LAN (WLAN), a WAN, a wireless WAN (WWAN), a metropolitan area network (MAN), the Internet, a portion of the Internet, a portion of the Public Switched Telephone Network (PSTN), a plain old telephone service (POTS) network, a cellular telephone network, a wireless network, a Wi-Fi® network, another type of network, or a combination of two or more such networks. For example, the network 580 or a portion of the network 580 may include a wireless or cellular network and the coupling 582 may be a Code Division Multiple Access (CDMA) connection, a Global System for Mobile communications (GSM) connection, or another type of cellular or wireless coupling. In this example, the coupling 582 may implement any of a variety of types of data transfer technology, such as Single Carrier Radio Transmission Technology (1xRTT), Evolution-Data Optimized (EVDO) technology, General Packet Radio Service (GPRS) technology, Enhanced Data rates for GSM Evolution (EDGE) technology, third Generation Partnership Project (3GPP) including 5G, fourth generation wireless (4G) networks, Universal Mobile Telecommunications System (UMTS), High Speed Packet Access (HSPA), Worldwide Interoperability for Microwave Access (WiMAX), Long Term Evolution (LTE) standard, others defined by various standard-setting organizations, other long range protocols, or other data transfer technology.

20  
**[0099]** The instructions 516 may be transmitted or received over the network 580 using a transmission medium via a network interface device (e.g., a network interface component included in the communication components 564) and utilizing any one of a number of well-known transfer protocols (e.g., HTTP). Similarly, the instructions 516 may be transmitted or received using a transmission medium via the coupling 572 (e.g., a peer-to-peer coupling) to the devices 570. The term "transmission medium" shall be taken to include any intangible medium that is capable of storing, encoding, or carrying the instructions 516 for execution by the machine 500, and includes digital or analog communications signals or other intangible media to facilitate communication of such software.

## 35 Claims

### 1. A system comprising:

one or more processors; and

40 a memory comprising instructions which, when executed by the one or more processors, cause the one or more processors to perform operations comprising:

accessing an input file (135) and code for testing (130), the input file comprising a plurality of bytes;

45 performing a plurality of runs of a fuzzing algorithm (320) using the input file and the code for testing, each run including: performing a mutation of one or more bytes of the input file and determining which parts of the code for testing were executed when the code was run with the mutated input file;

storing, for each run of the plurality of runs of the fuzzing algorithm, an indication of whether the mutation caused execution of a portion of the code for testing which was not executed prior to the mutation;

tailoring the fuzzing algorithm based on the stored indications;

50 executing the tailored fuzzing algorithm to identify one or more versions of the input file for which the code for testing crashes;

providing, as a digital transmission, indicia of the one or more versions of the input file for which the code for testing crashes; and

55 generating a heatmap (145) of the input file based on the stored indications, the heatmap mapping each of the plurality of bytes in the input file to a value indicating whether the mutation of the byte caused execution of the portion of the code for testing which was not executed prior to the mutation, wherein tailoring the fuzzing algorithm based on the stored indications comprises tailoring the fuzzing algorithm

based on the heatmap;  
 wherein generating a heatmap of the input file based on the stored indications comprises:

5            learning a machine-learned model based on the stored indications, the machine-learned model being for predicting the value indicating whether the mutation of each byte causes execution of the portion of the code for testing which was not executed prior to the mutation, said machine-learned model trained in a supervised setting; and  
              generating the heatmap of the input file using the machine-learned model.

10    **2.** The system of claim 1, wherein tailoring the fuzzing algorithm based on the heatmap comprises:  
 adjusting the fuzzing algorithm to perform mutations only on bytes associated, in the heatmap, with either a value exceeding a threshold, a non-zero value, or a TRUE value.

15    **3.** The system of claim 1, wherein the heatmap comprises a matrix identifying each byte of the input file and the value indicating whether the mutation of that byte caused execution of the portion of the code for testing which was not executed prior to the mutation.

20    **4.** The system of claim 1, wherein the fuzzing algorithm mutates one or more bytes in the input file to cause unexpected behavior in an execution of the code for testing.

25    **5.** The system of claim 1, wherein the code for testing is an application for processing files of a type associated with the input file.

30    **6.** The system of claim 1, wherein the value indicating whether the mutation of the byte caused execution of the portion of the code for testing which was not executed prior to the mutation is:

             one or TRUE if, in at least one instance, the mutation of the byte caused execution of the portion of the code for testing which was not executed prior to the mutation, and  
              zero or FALSE otherwise.

35    **7.** The system of claim 1, wherein the value indicating whether the mutation of the byte caused execution of the portion of the code for testing which was not executed prior to the mutation corresponds to a proportion, of runs where the byte had the mutation, which caused execution of the portion of the code for testing which was not executed prior to the mutation.

40    **8.** A machine-readable medium comprising instructions which, when executed by one or more processors of a machine, cause the one or more processors to perform operations comprising:

             performing a plurality of runs of a fuzzing algorithm (320) using an input file (135) and code for testing, the input file comprising a plurality of bytes, and each run including: performing a mutation of one or more bytes of the input file and determining which parts of the code for testing were executed when the code was run with the mutated input file;

             storing, for each run of the plurality of runs of the fuzzing algorithm, an indication of whether the mutation caused execution of a portion of the code for testing which was not executed prior to the mutation;

45               generating a heatmap (145) of the input file based on the stored indications, the heatmap mapping each of the plurality of bytes in the input file to a value indicating whether the mutation of the byte caused execution of the portion of the code for testing which was not executed prior to the mutation;

             tailoring the fuzzing algorithm based on the heatmap;

50               executing the tailored fuzzing algorithm to identify one or more versions of the input file for which the code for testing crashes; and

             providing, as a digital transmission, indicia of the one or more versions of the input file for which the code for testing crashes;

             wherein generating a heatmap of the input file based on the stored indications comprises:

55            learning a machine-learned model based on the stored indications, the machine-learned model being for predicting the value indicating whether the mutation of each byte causes execution of the portion of the code for testing which was not executed prior to the mutation, said machine-learned model trained in a supervised setting; and

generating the heatmap of the input file using the machine-learned model.

9. The machine-readable medium of claim 8, wherein the fuzzing algorithm mutates one or more bytes in the input file to cause unexpected behavior in an execution of the code for testing.

10. The machine-readable medium of claim 8, wherein the code for testing is an application for processing files of a type associated with the input file.

11. The machine-readable medium of claim 8, wherein tailoring the fuzzing algorithm based on the heatmap comprises: adjusting the fuzzing algorithm to perform mutations only on bytes associated, in the heatmap, with either a value exceeding a threshold, a non-zero value, or a TRUE value.

12. The machine-readable medium of claim 8, wherein the value indicating whether the mutation of the byte caused execution of the portion of the code for testing which was not executed prior to the mutation is:

one or TRUE if, in at least one instance, the mutation of the byte caused execution of the portion of the code for testing which was not executed prior to the mutation, and zero or FALSE otherwise.

13. The machine-readable medium of claim 8, wherein the value indicating whether the mutation of the byte caused execution of the portion of the code for testing which was not executed prior to the mutation corresponds to a proportion, of runs where the byte had the mutation, which caused execution of the portion of the code for testing which was not executed prior to the mutation.

## Patentansprüche

1. System, umfassend:

einen oder mehrere Prozessoren und einen Speicher, umfassend Anweisungen, die, wenn sie durch den einen oder die mehreren Prozessoren ausgeführt werden, den einen oder die mehreren Prozessoren dazu veranlassen, Operationen durchzuführen, umfassend:

Zugreifen auf eine Eingabedatei (135) und Code zum Testen (130), wobei die Eingabedatei eine Vielzahl von Bytes umfasst;

Durchführen einer Vielzahl von Ausführungen eines Fuzzing-Algorithmus (320) unter Verwendung der Eingabedatei und des Codes zum Testen, wobei jede Ausführung einschließt: Durchführen einer Mutation von einem oder mehreren Bytes der Eingabedatei und Bestimmen, welche Teile des Codes zum Testen ausgeführt wurden, als der Code mit der mutierten Eingabedatei ausgeführt wurde;

Speichern, für jede Ausführung der Vielzahl von Ausführungen des Fuzzing-Algorithmus, einer Angabe, ob die Mutation die Ausführung eines Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat;

individuelles Anpassen des Fuzzing-Algorithmus basierend auf den gespeicherten Angaben;

Ausführen des individuell angepassten Fuzzing-Algorithmus zum Identifizieren von einer oder mehreren Versionen der Eingabedatei, für die der Code zum Testen abstürzt;

Bereitstellen, als eine digitale Übertragung, von Zeichen der einen oder der mehreren Versionen der Eingabedatei, für die der Code zum Testen abstürzt; und

Erzeugen eines Wärmebilds (145) der Eingabedatei basierend auf den gespeicherten Angaben, wobei das Wärmebild jedes der Vielzahl von Bytes in der Eingabedatei einem Wert zuordnet, der angibt, ob die Mutation des Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat,

wobei das individuelle Anpassen des Fuzzing-Algorithmus basierend auf den gespeicherten Angaben ein individuelles Anpassen des Fuzzing-Algorithmus basierend auf dem Wärmebild umfasst;

wobei das Erzeugen eines Wärmebilds der Eingabedatei basierend auf den gespeicherten Angaben umfasst:

Lernen eines Maschinenlernmodells basierend auf den gespeicherten Angaben, wobei das Maschinen-

## EP 3 639 141 B1

lernmodell für das Vorhersagen des Werts ist, der angibt, ob die Mutation jedes Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt, wobei das Maschinenlernmodell eine überwachte Einstellung trainiert hat; und Erzeugen des Wärmebilds der Eingabedatei unter Verwendung des Maschinenlernmodells.

5

2. System nach Anspruch 1, wobei das individuelle Anpassen des Fuzzing-Algorithmus basierend auf dem Wärmebild umfasst:

Einstellen des Fuzzing-Algorithmus zum Durchführen von Mutationen nur bei Bytes, die im Wärmebild entweder einem Wert, der über eine Schwelle hinausgeht, einem Nicht-Null-Wert oder einem WAHR-Wert zugeordnet sind.

10

3. System nach Anspruch 1, wobei das Wärmebild eine Matrix umfasst, die jedes Byte der Eingabedatei und den Wert identifiziert, der angibt, ob die Mutation dieses Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat.

15

4. System nach Anspruch 1, wobei der Fuzzing-Algorithmus ein oder mehrere Bytes in der Eingabedatei mutiert, um ein unerwartetes Verhalten in einer Ausführung des Codes zum Testen zu bewirken.

5. System nach Anspruch 1, wobei der Code zum Testen eine Anwendung zum Verarbeiten von Dateien eines der Eingabedatei zugeordneten Typs ist.

20

6. System nach Anspruch 1, wobei der Wert, der angibt, ob die Mutation des Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat, ist:

eins oder WAHR, wenn in mindestens einer Instanz die Mutation des Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat, und andernfalls null oder FALSCH.

25

7. System nach Anspruch 1, wobei der Wert, der angibt, ob die Mutation des Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat, einem Anteil von Ausführungen entspricht, bei denen das Byte die Mutation aufwies, die die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat.

30

8. Maschinenlesbares Medium, umfassend Anweisungen, die wenn sie durch einen oder mehrere Prozessoren einer Maschine ausgeführt werden, den einen oder die mehreren Prozessoren dazu veranlassen, Operationen durchzuführen, umfassend:

35

Durchführen einer Vielzahl von Ausführungen eines Fuzzing-Algorithmus (320) unter Verwendung einer Eingabedatei (135) und eines Code zum Testen, wobei die Eingabedatei eine Vielzahl von Bytes umfasst, und wobei jede Ausführung einschließt: Durchführen einer Mutation von einem oder mehreren Bytes der Eingabedatei und Bestimmen, welche Teile des Codes zum Testen ausgeführt wurden, als der Code mit der mutierten Eingabedatei ausgeführt wurde;

40

Speichern, für jede Ausführung der Vielzahl von Ausführungen des Fuzzing-Algorithmus, einer Angabe, ob die Mutation die Ausführung eines Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat;

45

Erzeugen eines Wärmebilds (145) der Eingabedatei basierend auf den gespeicherten Angaben,

wobei das Wärmebild jedes der Vielzahl von Bytes in der Eingabedatei einem Wert zuordnet, der angibt, ob die Mutation des Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat;

50

individuelles Anpassen des Fuzzing-Algorithmus basierend auf dem Wärmebild;

Ausführen des individuell angepassten Fuzzing-Algorithmus zum Identifizieren von einer oder mehreren Versionen der Eingabedatei, für die der Code zum Testen abstürzt; und

Bereitstellen, als eine digitale Übertragung, von Zeichen der einen oder der mehreren Versionen der Eingabedatei, für die der Code zum Testen abstürzt;

55

wobei das Erzeugen eines Wärmebilds der Eingabedatei basierend auf den gespeicherten Angaben umfasst:

Lernen eines Maschinenlernmodells basierend auf den gespeicherten Angaben, wobei das Maschinen-

lernmodell für das Vorhersagen des Werts ist, der angibt, ob die Mutation jedes Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt, wobei das Maschinenlernmodell in einer überwachten Einstellung trainiert wird; und Erzeugen des Wärmebilds der Eingabedatei unter Verwendung des Maschinenlernmodells.

5

9. Maschinenlesbares Medium nach Anspruch 8, wobei der Fuzzing-Algorithmus ein oder mehrere Bytes in der Eingabedatei mutiert, um ein unerwartetes Verhalten in einer Ausführung des Codes zum Testen zu bewirken.

10

10. Maschinenlesbares Medium nach Anspruch 8, wobei der Code zum Testen eine Anwendung zum Verarbeiten von Dateien eines der Eingabedatei zugeordneten Typs ist.

15

11. Maschinenlesbares Medium nach Anspruch 8, wobei das individuelle Anpassen des Fuzzing-Algorithmus basierend auf dem Wärmebild umfasst:

Einstellen des Fuzzing-Algorithmus zum Durchführen von Mutationen nur bei Bytes, die im Wärmebild entweder einem Wert, der über eine Schwelle hinausgeht, einem Nicht-Null-Wert oder einem WAHR-Wert zugeordnet sind.

20

12. Maschinenlesbares Medium nach Anspruch 8, wobei der Wert, der angibt, ob die Mutation des Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat, ist:

eins oder WAHR, wenn in mindestens einer Instanz die Mutation des Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat, und andernfalls null oder FALSCH.

25

13. Maschinenlesbares Medium nach Anspruch 8, wobei der Wert, der angibt, ob die Mutation des Byte die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat, einem Anteil von Ausführungen entspricht, bei denen das Byte die Mutation aufwies, die die Ausführung des Abschnitts des Codes zum Testen, der vor der Mutation nicht ausgeführt wurde, bewirkt hat.

30

## Revendications

### 1. Système comprenant :

35

un ou plusieurs processeurs ; et  
une mémoire comprenant des instructions qui, quand elles sont exécutées par les un ou plusieurs processeurs, amènent les un ou plusieurs processeurs à réaliser des opérations comprenant :

40

l'accès à un fichier d'entrée (135) et à un code de contrôle (130), le fichier d'entrée comprenant une pluralité d'octets ;

la réalisation d'une pluralité d'exécutions d'un algorithme de contrôle de données aléatoires (320) en utilisant le fichier d'entrée et le code de contrôle, chaque exécution comprenant : la réalisation d'une mutation d'un ou plusieurs octets du fichier d'entrée et la détermination de quelles parties du code de contrôle ont été exécutées quand le code a été exécuté avec le fichier d'entrée muté ;

45

le stockage, pour chaque exécution de la pluralité d'exécutions de l'algorithme de contrôle de données aléatoires, d'une indication précisant si la mutation a provoqué une exécution d'une partie du code de contrôle qui n'avait pas été exécutée avant la mutation ;

l'adaptation de l'algorithme de contrôle de données aléatoires sur la base des indications stockées ;

l'exécution de l'algorithme de contrôle de données aléatoires adapté pour identifier une ou plusieurs versions du fichier d'entrée pour lesquelles le code de contrôle plante ;

50

la fourniture, sous forme d'une transmission numérique, d'indices des une ou plusieurs versions du fichier d'entrée pour lesquelles le code de contrôle plante ; et

55

la génération d'une carte thermique (145) du fichier d'entrée sur la base des indications stockées, la carte thermique mappant chacun de la pluralité d'octets dans le fichier d'entrée à une valeur indiquant si la mutation de l'octet a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation, dans lequel l'adaptation de l'algorithme de contrôle de données aléatoires sur la base des indications stockées comprend l'adaptation de l'algorithme de contrôle de données aléatoires sur la base de la carte thermique ; dans lequel la génération d'une carte thermique du fichier d'entrée sur la base des indications stockées

comprend :

5 l'apprentissage d'un modèle appris par machine sur la base des indications stockées, le modèle appris par machine étant destiné à prédire la valeur indiquant si la mutation de chaque octet provoque une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation, ledit modèle appris par machine étant formé dans un cadre supervisé ; et  
la génération de la carte thermique du fichier d'entrée en utilisant le modèle appris par machine.

10 2. Système selon la revendication 1, dans lequel l'adaptation de l'algorithme de contrôle de données aléatoires sur la base de la carte thermique comprend :  
l'ajustement de l'algorithme de contrôle de données aléatoires pour réaliser des mutations seulement sur des octets associés, dans la carte thermique, soit à une valeur dépassant un seuil, soit à une valeur non nulle, soit à une valeur VRAI.

15 3. Système selon la revendication 1, dans lequel la carte thermique comprend une matrice identifiant chaque octet du fichier d'entrée et la valeur indiquant si la mutation de cet octet a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation.

20 4. Système selon la revendication 1, dans lequel l'algorithme de contrôle de données aléatoires fait muter un ou plusieurs octets dans le fichier d'entrée pour provoquer un comportement imprévu dans une exécution du code de contrôle.

25 5. Système selon la revendication 1, dans lequel le code de contrôle est une application pour le traitement de fichiers d'un type associé au fichier d'entrée.

6. Système selon la revendication 1, dans lequel la valeur indiquant si la mutation de l'octet a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation est :

30 un ou VRAI si, dans au moins un cas, la mutation de l'octet a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation, et  
zéro ou FAUX autrement.

35 7. Système selon la revendication 1, dans lequel la valeur indiquant si la mutation de l'octet a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation correspond à une proportion d'exécutions où l'octet a subi la mutation qui a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation.

40 8. Support lisible par machine comprenant des instructions qui, quand elles sont exécutées par un ou plusieurs processeurs d'une machine, amènent les un ou plusieurs processeurs à réaliser des opérations comprenant :

45 la réalisation d'une pluralité d'exécutions d'un algorithme de contrôle de données aléatoires (320) en utilisant un fichier d'entrée (135) et un code de contrôle, le fichier d'entrée comprenant une pluralité d'octets, et chaque exécution comprenant : la réalisation d'une mutation d'un ou plusieurs octets du fichier d'entrée et la détermination de quelles parties du code de contrôle ont été exécutées quand le code a été exécuté avec le fichier d'entrée muté ;

le stockage, pour chaque exécution de la pluralité d'exécutions de l'algorithme de contrôle de données aléatoires, d'une indication précisant si la mutation a provoqué une exécution d'une partie du code de contrôle qui n'avait pas été exécutée avant la mutation ;

50 la génération d'une carte thermique (145) du fichier d'entrée sur la base des indications stockées, la

carte thermique mappant chacun de la pluralité d'octets dans le fichier d'entrée à une valeur indiquant si la mutation de l'octet a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation ;

55 l'adaptation de l'algorithme de contrôle de données aléatoires sur la base de la carte thermique ;  
l'exécution de l'algorithme de contrôle de données aléatoires adapté pour identifier une ou plusieurs versions du fichier d'entrée pour lesquelles le code de contrôle plante ; et  
la fourniture, sous forme d'une transmission numérique, d'indices des une ou plusieurs versions du fichier

## EP 3 639 141 B1

d'entrée pour lesquelles le code de contrôle plante ;  
dans lequel la génération d'une carte thermique du fichier d'entrée sur la base des indications stockées comprend :

- 5 l'apprentissage d'un modèle appris par machine sur la base des indications stockées, le modèle appris par machine étant destiné à prédire la valeur indiquant si la mutation de chaque octet provoque une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation, ledit modèle appris par machine étant formé dans un cadre supervisé ; et
- 10 la génération de la carte thermique du fichier d'entrée en utilisant le modèle appris par machine.
9. Support lisible par machine selon la revendication 8, dans lequel l'algorithme de contrôle de données aléatoires fait muter un ou plusieurs octets dans le fichier d'entrée pour provoquer un comportement imprévu dans une exécution du code de contrôle.
- 15 10. Support lisible par machine selon la revendication 8, dans lequel le code de contrôle est une application pour le traitement de fichiers d'un type associé au fichier d'entrée.
11. Support lisible par machine selon la revendication 8, dans lequel l'adaptation de l'algorithme de contrôle de données aléatoires sur la base de la carte thermique comprend :
- 20 l'ajustement de l'algorithme de contrôle de données aléatoires pour réaliser des mutations seulement sur des octets associés, dans la carte thermique, soit à une valeur dépassant un seuil, soit à une valeur non nulle, soit à une valeur VRAI.
12. Support lisible par machine selon la revendication 8, dans lequel la valeur indiquant si la mutation de l'octet a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation est :
- 25 un ou VRAI si, dans au moins un cas, la mutation de l'octet a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation, et zéro ou FAUX autrement.
- 30 13. Support lisible par machine selon la revendication 8, dans lequel la valeur indiquant si la mutation de l'octet a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation correspond à une proportion d'exécutions dans lesquelles l'octet a subi la mutation qui a provoqué une exécution de la partie du code de contrôle qui n'avait pas été exécutée avant la mutation.
- 35
- 40
- 45
- 50
- 55

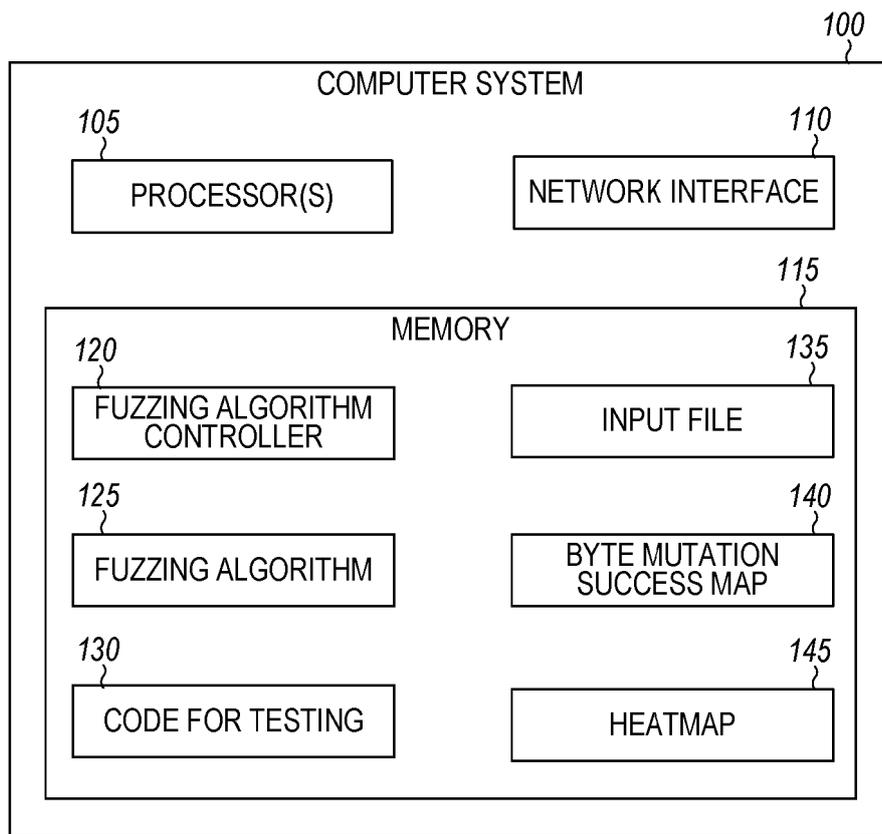


FIG. 1

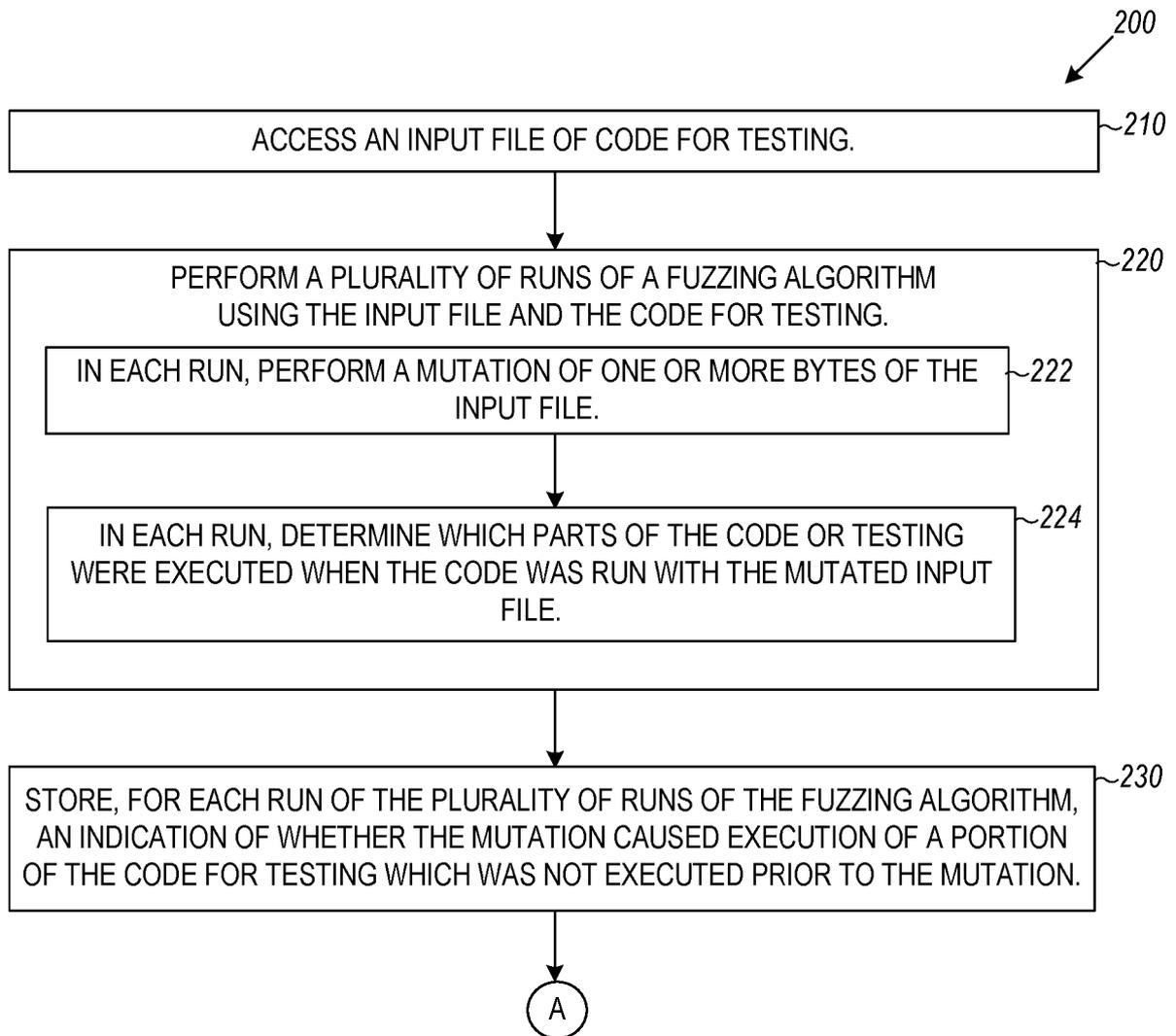


FIG. 2A

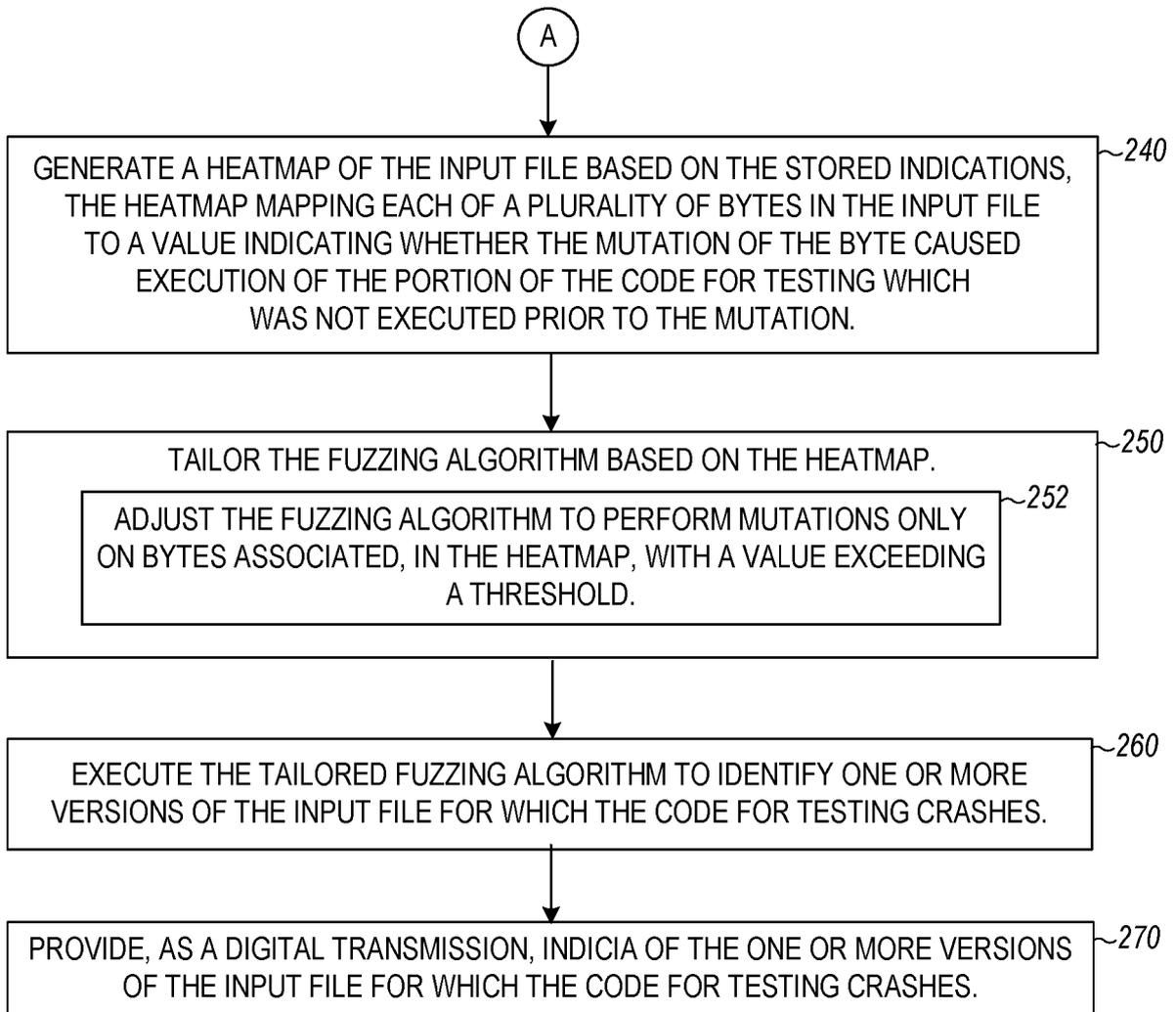


FIG. 2B

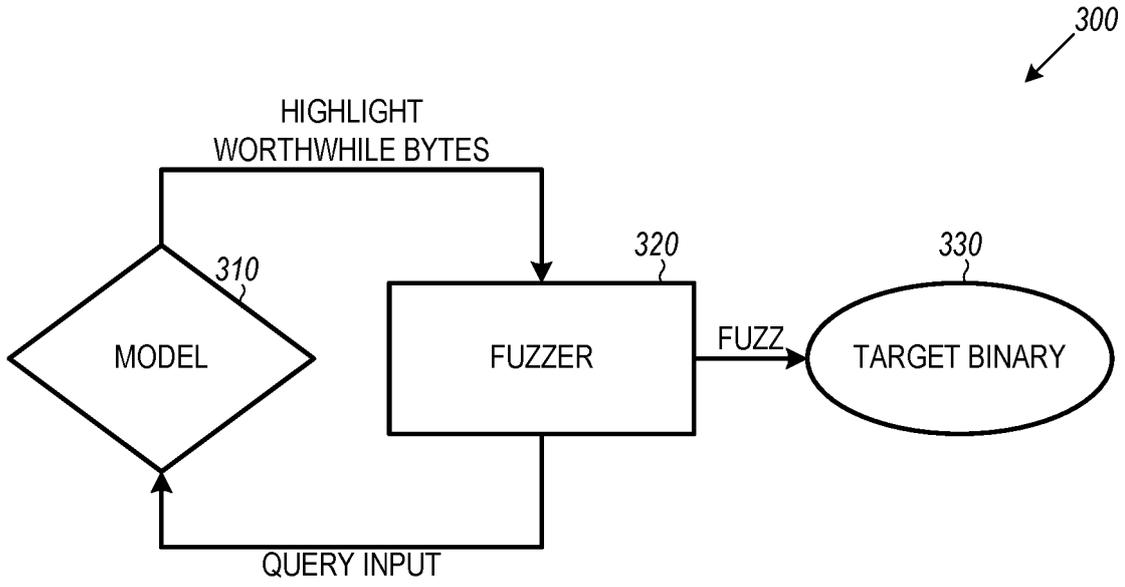


FIG. 3

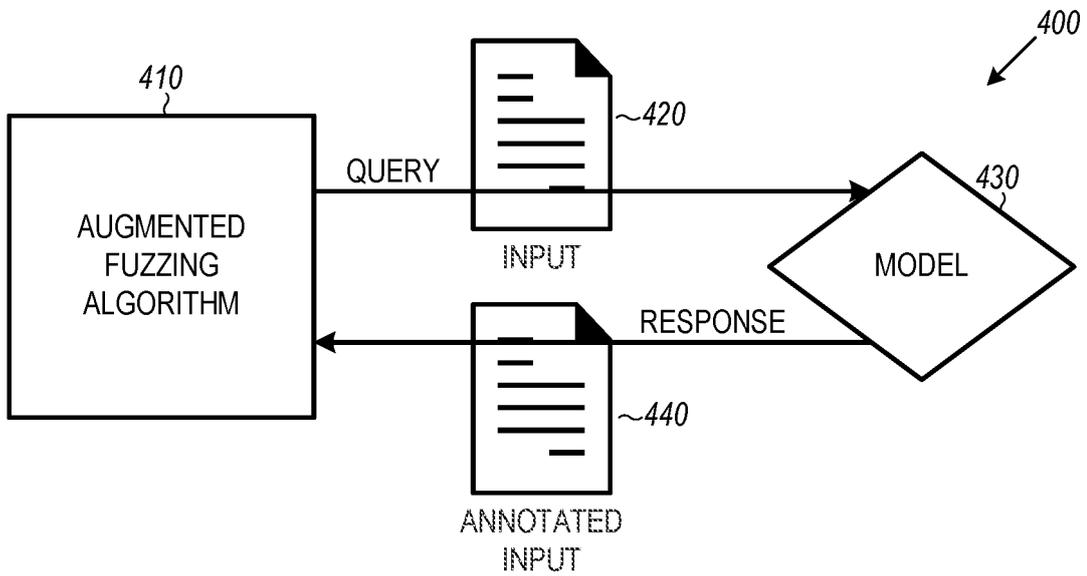


FIG. 4

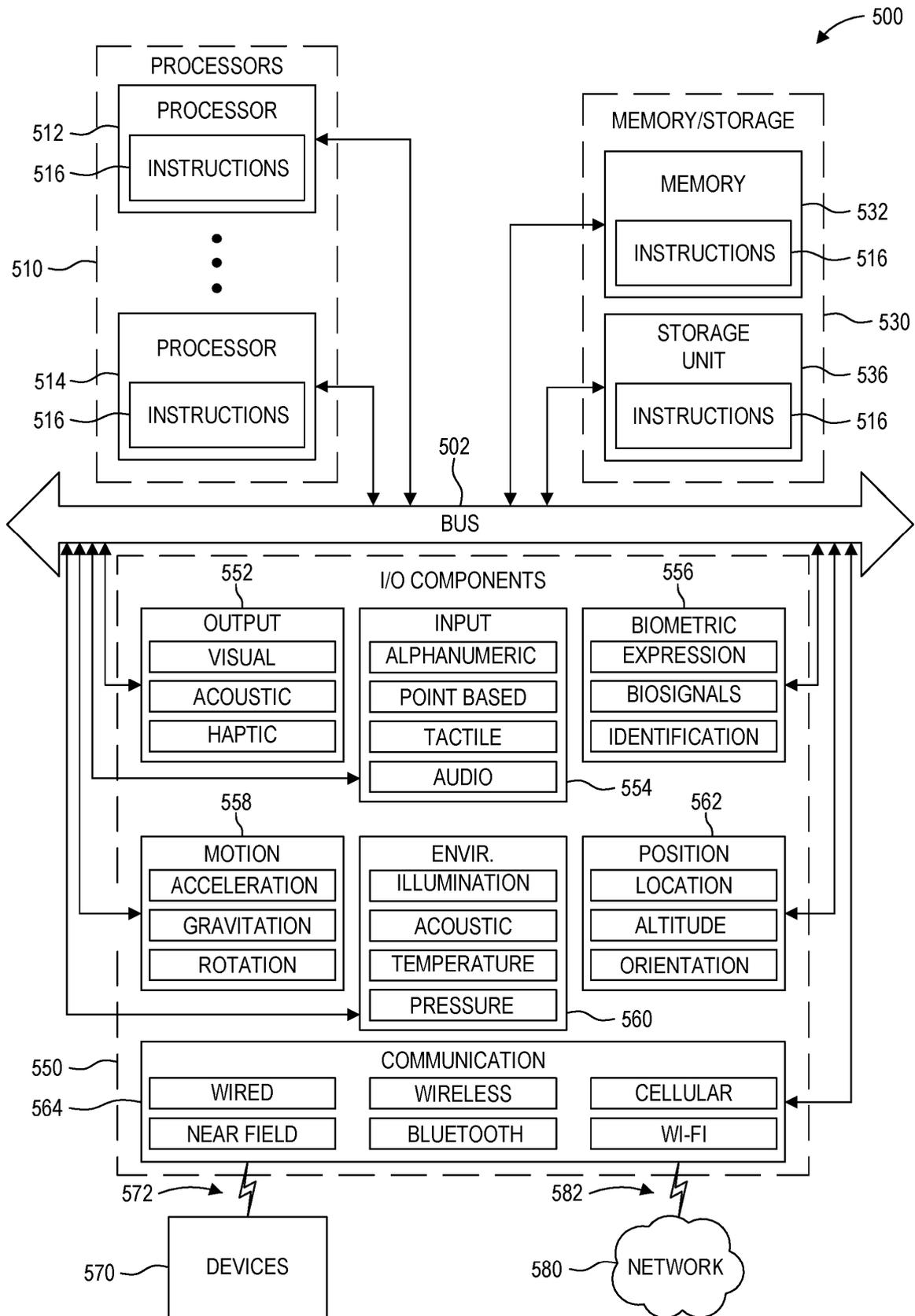


FIG. 5

**REFERENCES CITED IN THE DESCRIPTION**

*This list of references cited by the applicant is for the reader's convenience only. It does not form part of the European patent document. Even though great care has been taken in compiling the references, errors or omissions cannot be excluded and the EPO disclaims all liability in this regard.*

**Patent documents cited in the description**

- US 2011302455 A1 [0002]

**Non-patent literature cited in the description**

- **FABIAN BETERKE**. Distributed Evolutionary Fuzzing with Evofuzz. *Sicherheit 2016, Lecture Notes in Informatics (LN1)*, 01 January 2016, 23-32 [0002]
- **SHERRI SPARKS et al.** Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. *COMPUTER SECURITY APPLICATIONS CONFERENCE, 2007. ACSAC 2007. TWENTY-THIRD ANNUAL, IEEE*, 14 December 2007, ISBN 978-0-7695-3060-4, 1-10 [0002]