Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3713081.3731722

RESEARCH-ARTICLE

Personalized Fuzzing: A Case Study with the FANDANGO Fuzzer on a GNSS Module (Short Paper)

STEPHAN NEUHAUS, ZHAW Zurich University of Applied Sciences, Winterthur, ZH, Switzerland

JOSE ANTONIO ZAMUDIO AMAYA, CISPA - Helmholtz Center for Information Security, Saarbrucken, Saarland, Germany

ANDREAS ZELLER, CISPA - Helmholtz Center for Information Security, Saarbrucken, Saarland, Germany

Open Access Support provided by:

ZHAW Zurich University of Applied Sciences

CISPA - Helmholtz Center for Information Security

PDF Download
3713081.3731722.pdf
28 March 2026
Total Citations: 0
Total Downloads: 590

Published: 11 June 2025

Citation in BibTeX format

ISSTA Companion '25: 34th ACM SIGSOFT International Symposium on Software Testing and Analysis
June 25 - 28, 2025
Trondheim, Norway

Conference Sponsors:
SIGSOFT

ISSTA Companion '25: Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (June 2025)
https://doi.org/10.1145/3713081.3731722
ISBN: 9798400714740

Personalized Fuzzing: A Case Study with the FANDANGO Fuzzer on a GNSS Module (Short Paper)

Stephan Neuhaus
neut@zhaw.ch
ZHAW School of Engineering
Zurich, Austria

José Antonio Zamudio Amaya
jose.zamudio@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Andreas Zeller
zeller@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Abstract

Fuzzing is a widely used technique for uncovering vulnerabilities in software systems, but traditional fuzzers often struggle with generating valid and meaningful test cases for complex input formats. Grammar-based fuzzers address this issue by ensuring syntactic correctness, but they frequently lack fine-grained control over generated inputs to trigger specific behaviors. In this paper, we demonstrate the flexibility and effectiveness of FANDANGO, a state-of-the-art grammar-based fuzzer that incorporates constraint solving to produce 100% valid inputs while also guiding the generation process toward desired edge cases. Using a GNSS (Global Navigation Satellite System) module as a case study, we showcase how FANDANGO enables the specification of constraints to explore the module's behavior. Our experiments highlight FANDANGO's ability to generate targeted test cases that expose potential weaknesses. This study reinforces the practical applicability of constraint-guided grammar fuzzing in security testing and reliability analysis.

CCS Concepts

• **Hardware** → Hardware test; • **Theory of computation** → *Grammars and context-free languages*; • **Software and its engineering** → **Software testing and debugging**.

Keywords

Grammar-based fuzzing, Hardware fuzzing, GNSS

ACM Reference Format:
Stephan Neuhaus, José Antonio Zamudio Amaya, and Andreas Zeller. 2025. Personalized Fuzzing: A Case Study with the FANDANGO Fuzzer on a GNSS Module (Short Paper). In *34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA Companion '25), June 25–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3713081.3731722

1 Introduction

Fuzzing [6] is a widely adopted technique in software security testing, aimed at uncovering vulnerabilities by providing programs with malformed or unexpected inputs. Traditional fuzzers, such as random and coverage-guided fuzzers, have demonstrated remarkable effectiveness in finding security flaws [5]. However, they often struggle with complex input formats, where invalid test cases are quickly discarded by strict input parsers, limiting the fuzzer's ability to reach deeper program states [3]. Grammar-based fuzzing [7] addresses this limitation by ensuring that generated inputs conform to a specified *syntax*, increasing the likelihood of meaningful interactions with the target system. However, grammar fuzzing lacks *semantic* features for fine-grained control over input generation, making it challenging to produce targeted test cases that explore specific system behaviors or trigger rare edge cases [1].

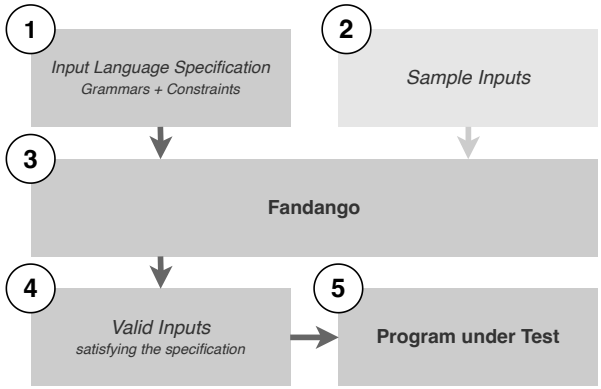


Figure 1: How FANDANGO works. FANDANGO takes a `.fan` input specification (1) with grammar, constraints, and Python functions. From the grammar, it creates an initial population (unless given (2)). It evolves the population through mutations and crossovers (3) to meet the Python-defined constraints, and produces a set of valid inputs (4) for the program under test (5).

In this paper, we present a case study of fuzzing a Global Navigation Satellite System (GNSS) [4] module using FANDANGO [2], a novel language-based fuzzer with constraint solving capabilities (Figure 1). By leveraging FANDANGO's ability to enforce syntactic correctness while also guiding the generation of test cases toward specific conditions, we showcase its potential for security testing. Our case study focuses on a u-blox MAX-M10S[1] GNSS module, a widely used navigation device that communicates using standardized NMEA messages and the proprietary UBX protocol [8]. By applying FANDANGO to the MAX-M10S module, we showcase how advanced fuzzing techniques can enhance security testing for embedded systems and proprietary communication protocols.

[1] https://www.u-blox.com/en/product/max-m10-series

This work is licensed under a Creative Commons Attribution 4.0 International License.
ISSTA Companion '25, Trondheim, Norway
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1474-0/2025/06
https://doi.org/10.1145/3713081.3731722

The rest of this paper is structured as follows: Section 2 provides background on GNSS communication protocols including the GNSS module's input structure. Section 3 introduces FANDANGO and its constraint-solving approach. Section 4 showcases the specified constraints in four different categories, and how these can be specified using FANDANGO. Section 5 discusses the fuzzing setup and the experimental results. Section 6 outlines limitations and future work. Section 7 concludes with key takeaways on the effectiveness of constraint-guided grammar fuzzing in security testing.

2 Background

Creating complex binary inputs is a challenge. We illustrate the problem with UBX, a binary protocol used by u-blox Global Navigation Satellite System (GNSS) modules.

u-blox is a well-known manufacturer of GNSS modules. One of its products is the MAX-M10S, a compact (9.7 mm by 10 mm) low-power (25 mW) device supporting multiple GNSS constellations, including GPS (US), Galileo (EU), BeiDou (China), and GLONASS (Russia). The module includes security features such as jamming detection, making it suitable for various applications requiring precise positioning. The MAX-M10S communicates via UART (serial), I²C, or SPI interfaces. It supports two primary protocols:

- **NMEA 0183** (ASCII-based, standardized) – Widely used across GNSS hardware and software but provides limited access to module features. An example NMEA message containing a position and time fix is shown in Figure 2.
- **UBX** (Binary, proprietary to u-blox) – Fully documented and allows access to the module's entire feature set. An example of a UBX message representing configuration message is given in Figure 3.

The UBX protocol has a frame structure, shown in Figure 4. Configuration messages are more complicated still. In order to set a configuration value, one must first obtain the specific configuration item's *key ID*, a four-byte unsigned integer, and then append the desired new value, which can have various lengths, depending on the key ID. Key IDs have some structure, shown in Figure 5. The size field is itself encoded. The value 1 indicates a one-bit field; in reality, this takes up one byte, of which only the least significant bit is used. The values 2 to 5 indicate one-, two, four, and eight-byte values, respectively. These may be signed or unsigned integers or

```
$GNGGA,091802.00,4722.67705,N,00831.97366,
E,2,12,0.65,429.1,M,47.3,M,,*4A
```

Figure 2: Example NMEA message containing a position and time fix. The time of this fix is 09:18:02 UTC. The position is 47 **degrees,** 22.67705 **minutes North,** 8 **degrees,** 31.97377 **minutes East. This fix is based on a number of different GNSS receivers. This is a differential GNSS fix, based on 12 satellites. The fix suffers from a horizontal dilution of precision (HDOP) of** 0.65**, an ideal value. Altitude above mean sea level is reported as** 429.1 **m, where the difference between the geoid and mean sea level is** 47.3 **m. The checksum is 4A. The reported position is a window on the second floor in Zurich's Lagerstrasse, where this fix was indeed taken.**

```
b562 068a 2900 0001 e077 6100 1110 0163  .b..)....wa....c
0011 5000 0000 0000 0079 4062 0011 5000  ..P......y@b..P.
0000 00f8 8e59 4164 0011 4000 401c 464b  .....YAd..@.@.FK
f3                                       .
```

Figure 3: Hex dump of an example UBX message containing some configuration commands. This message instructs the module to change the navigation configuration to use user-specified datum parameters, including new values for the semi-major axis, the 1.0/**flattening, and the** x**-axis shift at the origin. None of the parameters supplied are in their legal range, as set out by the manual [8, p. 141]**

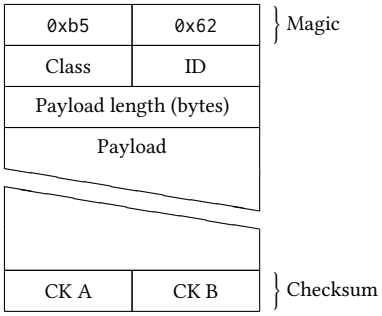


Figure 4: UBX frame structure. The number of bytes in a frame does not have to be even. The magic bytes are the letters 'µ' and 'b' (in ISO8859-1 encoding). Numerical values are transmitted in little-endian byte order. The manual claims that "values are placed in such an order that structure packing is not a problem. This means that two-byte values shall start on offsets that are a multiple of two; four-byte values shall start at a multiple of four; and so on" ([8, Section 3.3.1, p. 42]). As well as 1-, 2-, 4-, and 8-byte integer values (signed, unsigned, and bitfield), the protocol has 32- and 64-bit IEEE 754 binary floating-point values and ISO8859-1-encoded characters. Each frame has a checksum, computed using the 8-bit Fletcher algorithm, a nonstandard alternate TCP checksum extension. [9]

bitfields, enumerations (with a length up to four bytes), and four- or eight-byte IEEE 754 binary floating-point values.

The payload of a configuration message contains key IDs and values, stuffed without padding. Thus, the payload of a configuration message that sets four configuration keys to new values of 1, 4, 1, and 2 bytes respectively, will have these values starting at offsets 0, 5, 13, and 18, respectively; see Figure 6. The configuration of the navigation configuration is done with the NAVSPG

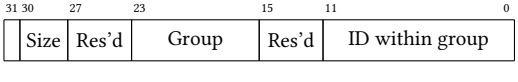


Figure 5: Structure of a UBX configuration key ID, according to [8, Section 4.2, p. 124]. Offsets are in bits.

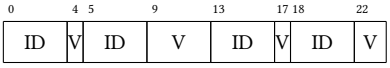


Figure 6: Layout of the payload of a configuration message that changes key IDs with value lengths of 1, 4, 1, and 2 bytes, respectively. Offsets are in bytes.

group of configuration key IDs. There are 31 configuration key IDs in that group, and they cover almost all possible data types (1-bit booleans; 1-byte signed and unsigned integers, enumerations, and bitfields; 2-byte unsigned values; 4-byte signed and unsigned integers; and 4- and 8-byte IEEE 754 binary floating-point values). Especially IEEE 754 binary floating-point values have a few gotchas of which programmers are often not aware, such as: not-a-number bit patterns (bit patterns that do not have a numerical value in the format); minus zero (a value that compares equal to plus zero, but which has a bit pattern distinct from it); and plus and minus infinity (bit patterns that always compare "greater than" (for $+\infty$) or "less than" (for $-\infty$) to any finite number). The subset of configuration keys that we are interested in, namely the ones that change the geoid on which fixes are reported (`CFG-NAVSPG-USRDAT_xxx`), has only these 4- and 8-byte IEEE 754 binary floating-point values.

To send a configuration message, the respective message class and ID are `0x06` and `0x8a`, respectively; this message is called `UBX-CFG-VALSET` in the documentation [8, Section 3.10.5].

The fact that the module allows user commands to change important navigation parameters, including the geoid on which position fixes are based, raises the risk when the validation of such parameters is not done properly. We have therefore chosen to fuzz this part of the UBX protocol. We have also chosen to allow configuration of these parameters only in the module's RAM, and neither in battery-backed RAM (BBR) or Flash, to allow any potentially serious misconfigurations to be erased by resetting the module, which deletes all configuration in RAM.

3 Evolutionary Language-based Fuzzing

Traditional fuzzers often rely on random mutations or heuristic-driven techniques, which can be effective in many scenarios but fall short when a more targeted approach is necessary. This is especially true for complex input formats like UBX, where generating inputs that are syntactically correct while also challenging the system with specific conditions can be a difficult task. The FANDANGO Fuzzer [2] addresses this challenge by combining grammar-based fuzzing with constraint-solving capabilities, providing a tool for generating structured inputs. At its core, FANDANGO operates by specifying an input language in a *.fan* file, which defines the syntax and constraints of the inputs the fuzzer should generate. The specification language is highly expressive and combines traditional grammar definitions with Python-based constraints, allowing testers to define detailed requirements for the generated inputs. This makes FANDANGO a solid choice for testing protocols like UBX, where precise control over the input format is necessary to uncover subtle vulnerabilities.

Testers can specify particular values, distributions, or combinations of parameters that they want to test, ensuring that the fuzzer produces inputs that align with the desired conditions. Under the hood, FANDANGO leverages evolutionary algorithms to refine input generation. It starts with a population of random inputs and evolves these inputs through a series of mutations and crossovers, gradually adapting to fit the constraints specified by the tester. This evolutionary approach ensures that the generated inputs are diverse, realistic, and more likely to trigger bugs or vulnerabilities that might otherwise remain hidden.

```
# ubx_cfg_navspg.fan (no constraints added)
<start> ::= <ubx_message>

<ubx_message> ::= <ubx_header> <ubx_payload> <ubx_checksum>
<ubx_header> ::= <ubx_magic><ubx_msg_class><ubx_msg_id><ubx_msg_len>
<ubx_magic> ::= b'\xb5\x62'
<ubx_msg_class> ::= b'\x06'   # CFG messages
<ubx_msg_id> ::= b'\x8a'      # CFG-VALSET message
<ubx_msg_len> ::= <u2>

<ubx_payload> ::= <version> <layers> <reserved0> <cfgdata>
<version> ::= b'\x00'
<layers> ::= b'\01'           # RAM only
<reserved0> ::= b'\x00\x00'
<cfgdata> ::= <use_usrdat> <usrdat_maja> <usrdat_flat> \
    <usrdat_dx> <usrdat_dy> <usrdat_dz> <usrdat_rotx> \
    <usrdat_roty> <usrdat_rotz> <usrdat_scale>

<use_usrdat> ::= b'\x25\x00\x11\x10' b'\x01'
<usrdat_maja> ::= b'\x62\x00\x11\x50' <r8_maja>
<usrdat_flat> ::= b'\x63\x00\x11\x50' <r8_flat>
<usrdat_dx> ::= b'\x64\x00\x11\x40' <r4_d>
<usrdat_dy> ::= b'\x65\x00\x11\x40' <r4_d>
<usrdat_dz> ::= b'\x66\x00\x11\x40' <r4_d>
<usrdat_rotx> ::= b'\x67\x00\x11\x40' <r4_rot>
<usrdat_roty> ::= b'\x68\x00\x11\x40' <r4_rot>
<usrdat_rotz> ::= b'\x69\x00\x11\x40' <r4_rot>
<usrdat_scale> ::= b'\x6a\x00\x11\x40' <r4_scale>
<ubx_checksum> ::= <u2>
<u1> ::= <byte>
<u2> ::= <byte>{2}
<r8_maja> ::= <byte>{8}
<r8_flat> ::= <byte>{8}
<r4_d> ::= <byte>{4}
<r4_rot> ::= <byte>{4}
<r4_scale> ::= <byte>{4}
```

Listing 1: FANDANGO specification for the UBX protocol. It targets the RAM layer, and fixes the message class and ID to set values. Constraints are added in Section 4.

3.1 Fandango Specification

The grammar for the UBX protocol input is defined using the FANDANGO specification language, which combines a *formal grammar* with *Python-based constraints* to create valid and complex inputs. The grammar starts by defining the structure of a complete UBX message, which consists of a header, payload, and checksum. Each part of the message is broken down into specific components, as shown in Listing 1.

4 Personalized Fuzzing

A strength of FANDANGO is its ability to *shape* inputs to meet specific requirements. By specifying additional constraints, grammar rules, or functions, testers can direct the fuzzer to generate inputs that target assumed or known weaknesses in the system under test. In this section, we illustrate some examples of such personalizations.

4.1 Distributions

Fuzzing grammar rules often define fields as sequences of bytes, but in many cases, these fields must adhere to specific distributions. For instance, floating-point values in a binary format may need to be sampled from a constrained range rather than being purely random. To achieve this, we introduce *generators*—Python functions invoked during fuzzing to produce specific values required for some fields. For example, in our grammar, we define floating-point fields using generator functions. The corresponding generator functions ensure that values adhere to meaningful minimum and maximum values, while shaping an uniform distribution across the entire test suite:

```
<r8_maja> ::= <byte>{8} := make_ieee_double_maja()
<r8_flat> ::= <byte>{8} := make_ieee_double_flat()
<r4_d> ::= <byte>{4} := make_ieee_float_d()
<r4_rot> ::= <byte>{4} := make_ieee_float_rot()
<r4_scale> ::= <byte>{4} := make_ieee_float_scale()

import struct
import random

def make_ieee_double_maja() -> bytes:
    # Major axis can be between 6,300,000 and 6,500,000 metres
    return struct.pack('<d', random.uniform(6300000.0, 6500000.0))

def make_ieee_double_flat() -> bytes:
    # Flatness can be between 0 and 500
    return struct.pack('<d', random.uniform(0.0, 500.0))

def make_ieee_float_d() -> bytes:
    # Displacement can be between -5000.0 and +5000.0 metres
    return struct.pack('<f', random.uniform(-5000.0, +5000.0))

def make_ieee_float_rot() -> bytes:
    # Rotation can be between -20.0 and +20.0 milli arcseconds
    return struct.pack('<f', random.uniform(-20.0, +20.0))

def make_ieee_float_scale() -> bytes:
    # Scale can be between 0.0 and 50.0 ppm
    return struct.pack('<f', random.uniform(0.0, 50.0))
```

4.2 Invalid Values

Certain edge cases in software arise from invalid values such as denormalized floating-point numbers, positive and negative infinity and NaN (Not-a-Number) values. To ensure that fuzzing explores these edge cases, we modify our value generators to include them probabilistically. This approach ensures that fuzzing includes a realistic mix of typical invalid values, increasing the likelihood of discovering edge-case vulnerabilities.

```
import sys
def make_ieee_float_scale() -> bytes:
    PROB_LEGAL  = 97.0  # Probability for a legal value, [-50.0, 50.0]
    PROB_POSINF =  1.0  # Probability for positive infinity
    PROB_NEGINF =  1.0  # Probability for negative infinity
    PROB_NAN    =  1.0  # Probability for NaN

    x = random.uniform(0, 100)  # Generate a random percentage
    if x <= PROB_LEGAL:
        return struct.pack('<f', random.uniform(-50.0, 50.0))
    elif x <= PROB_LEGAL + PROB_POSINF:
        return struct.pack('<f', float('inf'))
    elif x <= PROB_LEGAL + PROB_POSINF + PROB_NEGINF:
        return struct.pack('<f', float('-inf'))
    elif x <= PROB_LEGAL + PROB_POSINF + PROB_NEGINF + PROB_NAN:
        return struct.pack('<f', float('nan'))
```

4.3 Invalid Length Encodings

In structured binary formats such as UBX, message lengths must be correctly encoded. The UBX format specifies the message length as a 2-byte unsigned integer field, which represents the size of the payload in bytes. This length must be computed dynamically during fuzzing rather than assigned arbitrarily. We can check for this requirement as follows:

```
where <ubx_msg_len> != uint16(len(bytes(<ubx_payload>)))

def uint16(n: int) -> bytes:
    return n.to_bytes(2, 'little')
```

If instead, we desired to produce only valid values for the message length field, the constraint should be as follows:

```
where <ubx_msg_len> == uint16(len(bytes(<ubx_payload>)))
```

4.4 Partial Checksums

In some cases, we would like to check if checksum can be partially corrupted. We enforce a constraint to generate inputs with mismatched checksums to test how the system handles such errors. The following grammar rule generates a message where the checksum does not match the expected value.

```
where <ubx_checksum>
      != ubx_checksum(bytes(<ubx_header>), bytes(<ubx_payload>))

def ubx_checksum(header: bytes, payload: bytes) -> bytes:
    ck_a = 0
    ck_b = 0

    for c in (header + payload)[2:]:  # Skip the first two bytes
        ck_a = (ck_a + c) & 0xff
        ck_b = (ck_b + ck_a) & 0xff

    return bytes([ck_a, ck_b])
```

If instead, we desired to produce only valid values for the checksum field, the constraint should be as follows:

```
where <ubx_checksum>
      == ubx_checksum(bytes(<ubx_header>), bytes(<ubx_payload>))
```

4.5 Invalid Field Combinations

Some protocol fields may require that certain other fields have specific values for them to be valid. We define constraints to generate invalid combinations of these fields, ensuring that the system can handle them correctly. This constraint ensures that the system handles cases where the field combination might be logically inconsistent, such as a displacement being zero when the user data is in use.

```
where <use_usrdat> == b'\x01' and <usrdat_dx> == b'\x00\x00\x00\x00'
```

5 Results

We used FANDANGO to create 100 test inputs in the directory `corpus` as follows:

```
$ fandango fuzz --file-mode=binary -f ubx.fan -n 100 -d corpus
```

This took between 5.2 and 10.3 seconds on a laptop with an Intel i7-1365U CPU running at 1.6 GHz. Then we fed the resulting files to the module. The scripts used, inputs generated and the full FANDANGO specification can be found in https://anonymous.4open.science/r/gnss-fuzzing-0143.

If the module took longer than two seconds to answer a VALSET message, we interpreted that as a crash of the module; according to the documentation a response (positive or negative) is guaranteed

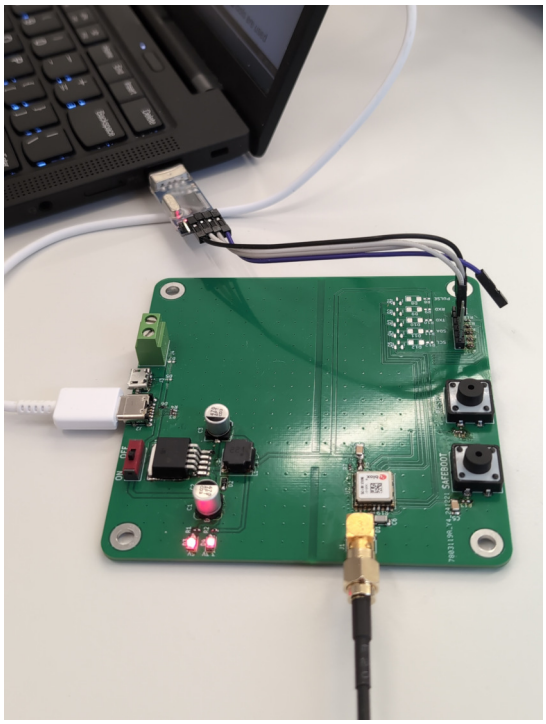


Figure 7: Test setup for this fuzzing run (author's photograph). The PCB was designed and populated by one of the authors. On the bottom is the antenna cable with an active GNSS antenna attached, on the top is a UART-to-USB converter. The module is powered by USB-C, seen on the left.

to happen inside of 500 milliseconds, but in this test run, the NMEA protocol was running simultaneously on the same UART, and transmitting a long batch of messages can take longer than this at the default baud rate of 9600 baud. In case of a crash, the input that caused the crash would be written to the `crashes` directory. In our test, all of the legal messages produced positive acknowledgment messages, and the ilegal messages (such as the ones with infinity or NaN values) produced negative acknowledgments in all cases. Lucky us!

6 Limitations

While FANDANGO offers an approach to fuzzing through its use of genetic algorithms and user-defined constraints, there are some inherent limitations associated with the tool and the underlying methodology. One of the primary limitations of genetic algorithms, such as those used in FANDANGO, is the potential inefficiency when dealing with non-monotonic constraints. Unlike exact algorithms that can rigorously solve problems with well-defined, monotonic constraints, genetic algorithms rely on probabilistic methods to explore the input space. This approach is inherently more flexible but may struggle with constraints that require precise, non-monotonic relationships between variables. As a result, genetic algorithms may not always guarantee that all constraints will be met, especially when complex interdependencies exist between fields.

Additionally, while FANDANGO allows users to specify constraints, goals and distributions in a Pythonic way, its performance may degrade when working with very large input spaces or highly complex formats. The evolutionary process, though powerful, is computationally intensive and may require substantial resources for extensive fuzzing campaigns. This can limit its scalability for certain use cases. Despite these limitations, FANDANGO is a valuable tool for fuzz testing, particularly when tailored goals and constraints are well-defined. Its flexibility and adaptability provide significant advantages in discovering vulnerabilities across a range of input specifications.

7 Conclusions and Future Work

In this paper, we showcased how FANDANGO can be used to generate tailored test cases for fuzz testing. By leveraging the power of its grammar-based Pythonic specification language and incorporating custom constraints, testers can target specific aspects of an input specification, ensuring that a variety of corner cases and edge cases are covered during fuzz testing.

One of the key strengths of FANDANGO lies in its ability to incorporate user-defined constraints, whether they involve distributions, extreme values, invalid combinations, or other domain-specific requirements. This allows testers to focus on particular failure modes that are relevant to their system, significantly increasing the chances of identifying bugs or vulnerabilities.

Our future work will focus on the following topics:

- While FANDANGO offers an efficient approach to fuzzing, its heuristic nature means that certain constraints may not always be fully satisfied, especially when complex interdependencies are involved. Future work could explore ways to combine the flexibility of genetic algorithms with more deterministic or exact approaches to improve coverage and efficiency, particularly in environments where precision is critical.
- Further improvements to FANDANGO could include better optimization techniques to reduce the computational overhead associated with large input spaces, as well as incorporating additional support for more complex input specifications. Another avenue for future work would be to explore hybrid fuzzing techniques that combine the strengths of black-box fuzzing with white-box approaches, enabling more effective exploration of the input space.

In summary, we consider FANDANGO a step forward in automated fuzz testing by allowing the creation of highly tailored test cases through flexible constraints and evolutionary techniques. With ongoing improvements and further integration of hybrid methods, such as program coverage guidance, better diversification algorithms or execution feature information, FANDANGO has the potential to become a robust and efficient tool for testing complex systems and uncovering vulnerabilities.

Acknowledgments.
This research was funded by the European Union (ERC S3, 101093186). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Hamad Al Salem and Jia Song. "A review on grammar-based fuzzing techniques". In: *International Journal of Computer Science & Security (IJCSS)* 13.3 (2019), pp. 114–123 (cit. on p. 1).

[2] José Antonio Zamudio Amaya, Marius Smytzek, and Andreas Zeller. *Fuzzing with FANDANGO*. https://fandango-fuzzer.github.io/. 2025 (cit. on pp. 1, 3).

[3] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. "Fuzzing: Challenges and reflections". In: *IEEE Software* 38.3 (2020), pp. 79–86 (cit. on p. 1).

[4] Richard B Langley, Peter JG Teunissen, and Oliver Montenbruck. "Introduction to GNSS". In: *Springer handbook of global navigation satellite systems* (2017), pp. 3–23 (cit. on p. 1).

[5] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* 1 (2018), pp. 1–13 (cit. on p. 1).

[6] Barton P Miller, Lars Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM* 33.12 (1990), pp. 32–44 (cit. on p. 1).

[7] Sevak Sargsyan et al. "Grammar-based fuzzing". In: *2018 Ivannikov Memorial Workshop (IVMEM)*. IEEE. 2018, pp. 32–35 (cit. on p. 1).

[8] u-blox. *u-blox M10 SPG 5.10. Standard precision GNSS firmware*. R03. Interface description. June 27, 2023. URL: https://content.u-blox.com/sites/default/files/u-blox-M10-SPG-5.10_InterfaceDescription_UBX-21035062.pdf (visited on 09/24/2024) (cit. on pp. 1–3).

[9] Johnny Zweig and Dr. Craig Partridge. *TCP alternate checksum options*. RFC 1145. Feb. 1990. DOI: 10.17487/RFC1145. URL: https://www.rfc-editor.org/info/rfc1145 (cit. on p. 2).