# Fuddly: un *framework* de fuzzing et de manipulation de données

Éric Lacombe

Airbus Operations S.A.S. – France

eric.lacombe@airbus.com
eric.lacombe@security-labs.org

**Résumé** `fuddly` est un framework de fuzzing et de manipulation de données. Il est actuellement utilisé dans le cadre de tests d'équipements avioniques qui mettent souvent en oeuvre des protocoles spécifiques. D'autres formats de données et protocoles plus génériques ont également été testés dans ce contexte (`JPG`, `PNG`, `ZIP`, `PDF` et `USB`). Les principales caractéristiques de `fuddly` sont de permettre à l'utilisateur : de modéliser des formats de données très variés ; de les combiner entre eux ; de faciliter la mise en oeuvre de transformations complexes sur ces données ; d'autoriser la dissection de données existantes ; et d'allier des techniques de génération et de mutation. Enfin, il fournit des moyens pour automatiser le processus de fuzzing en s'appuyant sur différentes briques permettant de communiquer avec la cible, de suivre et d'observer son comportement, et d'agir en conséquence.

## 1 Introduction

Parmi la variété d'approches complémentaires que l'on retrouve dans l'évaluation de la sécurité d'une cible (ex : logiciel, équipement embarqué, *etc.*), le *fuzzing* est reconnu pour être une méthode efficace d'aide à la découverte de faiblesses et de vulnérabilités.

Le *fuzzing* est une approche de test, dont l'objectif est de trouver des défauts d'implémentation ou de conception dans une cible. Dans cette approche, la façon d'interagir avec la cible est un point clé. Elle vise le plus souvent à transgresser les attentes qu'une cible pourrait avoir sur la façon dont on devrait communiquer avec elle, ainsi que sur le contenu de cette communication. Cette transgression doit toutefois palier les possibles tests de conformité effectués par la cible, afin d'éviter le rejet de la communication, tout en conservant un potentiel de déclenchement de *bugs*. Afin d'atteindre cet objectif, différentes méthodes sont à considérer, comme l'utilisation de données mal-formées, les altérations du séquencement nominal du protocole, *etc.* À noter que le fuzzing est une approche similaire à l'*injection de fautes* du domaine de la sûreté de fonctionnement.

L'expérience acquise sur le fuzzing d'équipements avioniques dans le cadre de notre travail, nous a poussé à développer notre propre *framework* de fuzzing. La raison à cela est que nous souhaitions laisser à l'utilisateur la possibilité : (1) de spécifier des formats de données de façon plus flexible que ce qui est permis au travers des autres *frameworks*[1], et (2) d'effectuer des transformations de données plus élaborées. En outre, développer un nouveau *framework* fut considéré comme la solution la plus adaptée pour expérimenter de nouveaux concepts. De ce choix découle également une liberté totale quant à son évolution. `fuddly` est cette tentative visant à construire sur des concepts existants et à mettre en œuvre nos idées, tout en restant compatible avec nos contraintes : pas de disponibilité du code source des cibles, architectures non-x86 (voire exotiques), moyens d'observabilité spécifiques à l'avionique.[2]

Une vue d'ensemble de fuddly est donnée en section 2. La modélisation des données est ensuite abordée en section 3, avant d'introduire les manipulations permises sur ces données en section 4. L'automatisation du processus de fuzzing est alors présentée en section 5. Enfin, les perspectives d'évolution envisagées sont examinées en section 6.

## 2 Présentation de `fuddly`

Notre *framework* de fuzzing et de manipulation de données est disponible sous licence GPL à l'adresse `https://github.com/k0retux/fuddly`. Il est écrit en *python* (compatible avec la version 2.7 jusqu'à la version 3.4) et peut être utilisé directement depuis un interpréteur *python* (comme `ipython`[3]), ou bien depuis un *shell* dédié (simplifiant certaines opérations). Ses principales caractéristiques sont de permettre à l'utilisateur :

1. de modéliser des formats de données très variés, qui :

1. Comme par exemple `sulley` [1] ou `Peach` [2].
2. Ces contraintes rendent difficile, voire impossible, l'utilisation de fuzzers s'appuyant sur des DBI comme `Fuzzgrind` [3] ou `Flayer` [4]. De même, les fuzzers qui instrumentent le code source comme `american fuzzy lop` [5] sont exclus.
3. `http://ipython.org/`

— mêlent des représentations très fines pour certains aspects avec d'autres plus grossières, tout en offrant la possibilité de les raffiner au besoin ;
— peuvent être combinés entre eux ;
— autorisent la dissection, de façon à manipuler facilement des données existantes ;
— permettent d'allier des techniques de génération de données avec des techniques de mutations ;

2. de faciliter la mise en œuvre de transformations complexes. Nous entendons par "complexe", la capacité d'agir sur n'importe quelle partie d'une donnée (composée d'éléments non nécessairement contiguës) tout en préservant la cohérence des dépendances, si l'utilisateur le souhaite. Cela revient à rendre possible des transformations articulées autour de critères syntaxiques (ex : altération d'une valeur entière en fonction de la taille du champ l'hébergeant) et sémantiques (ex : altération d'une valeur en fonction de sa signification pour un format ou protocole donné, altération d'un ensemble d'éléments d'une donnée formant un tout cohérent pour un format ou protocole donné) ;
3. d'automatiser le processus de fuzzing en s'appuyant sur différentes briques permettant de communiquer avec la cible, de suivre et d'observer son comportement, et d'agir en conséquence (ex : dévier des exigences du protocole vis-à-vis du séquencement, des contraintes temporelles, *etc.*) en bénéficiant de primitives de recherche et de transformation du modèle de données ; tout en consignant les différentes informations générées lors de ce processus, et d'en permettre le rejeu.

Ce *framework* est actuellement utilisé dans le cadre de tests d'équipements avioniques qui mettent souvent en oeuvre des protocoles spécifiques. D'autres formats de données et protocoles plus génériques ont également été testés dans ce contexte (JPG, PNG, ZIP, PDF et USB), avec des modélisations plus ou moins fines. La variété des formats de données auxquels nous avons été confrontés d'une part, et des cibles d'autre part, nous ont permis de maturer notre façon de modéliser les données, mais également le *framework* dans son ensemble.

Enfin, bien que `fuddly` dispose d'une infrastructure pensée pour le fuzzing, sa modélisation des données et la liberté qu'il offre pour les manipuler, peut s'avérer utile dans d'autres contextes. [4]

## 3 La modélisation des données

### 3.1 Vue d'ensemble

La représentation des données est effectuée dans `fuddly` au travers de la description d'un graphe orienté acyclique, dont les nœuds terminaux décrivent les différentes parties d'un format de données, et les arcs (qui peuvent être de natures différentes) capturent sa structure. Ce graphe intègre à la fois les informations syntaxiques et sémantiques sur le format de données. Dans la suite, nous appelons cette description de graphe un "modèle de données". Une donnée particulière est alors appelée "une instance" de ce modèle. Le choix de cette modélisation permet de représenter avec flexibilité des formats de données de nature très différentes. Par "flexibilité", nous entendons d'une part la possibilité de modéliser de façon plus ou moins fine les différentes parties d'un format (par exemple une modélisation fine sur des parties jugées plus complexe à traiter par la cible, et une modélisation grossière sur le reste), et d'autre part un niveau d'expressivité important.

À partir de ce modèle, nous pouvons générer des données et absorber des données existantes (projection d'une donnée existante dans le modèle de données). La génération de données autorise la création de données conformes au modèle si l'on souhaite interagir de façon correcte avec la cible ; ou la création de données dégénérées afin d'éprouver la robustesse de la cible. L'absorption des données peut servir à générer des données à partir de données existantes lorsque la modélisation n'est pas assez fine pour générer des données suffisamment correctes par elles-mêmes ; ou encore de comprendre les sorties de la cible, afin d'y répondre de façon adéquate ou non.

La génération de données revient à parcourir le graphe qui modélise le format de données. Après chaque parcours, une donnée est produite et chaque parcours fait évoluer le graphe, au choix, de façon déterministe ou arbitraire. Le parcours du graphe peut également être accompagné d'altération sur les nœuds (*cf.* section 4 page 6). Différents types de nœuds ont été définis pour la modélisation des données :

— Des nœuds terminaux qui hébergent des variables typées (ex : `UINT16`, `BitField`, `String`, *etc.*).
— Des nœuds non-terminaux qui définissent la structure des données. Ils servent à ordonner les différentes parties d'un format de données, et peuvent aller jusqu'à décrire une grammaire.
— Des nœuds *générateurs* qui servent à générer dynamiquement une partie du graphe en fonction d'autres nœuds (inclus ou non dans le graphe) et/ou d'autres critères, passés en paramètre.

La structure d'un format de données est capturée par les liaisons entre les nœuds du graphe. Différents type de liens se distinguent :

— les liens de parenté qui définissent une structure de base entre les nœuds du graphe. Ils sont régis par les nœuds non-terminaux.

4. Comme la construction de données spécifiques visant à mettre en exergue une vulnérabilité dévoilée au cours d'un processus de fuzzing par exemple.

— les liens associés à des critères spécifiques qui conditionnent des parties du graphe. Par exemple, la génération d'un nœud spécifique peut être associée à l'existence d'un autre, ou encore différents ensembles de nœuds peuvent être synchronisés sur leur cardinalité.
— les liens définis entre les nœuds générateurs et les nœuds qu'ils prennent en paramètres. Ils sont notamment utiles dès lors qu'une relation évoluée existe entre plusieurs nœuds. Les nœuds générateurs servent alors à régir cette relation, en la définissant au travers d'une fonction.

En outre, à chacun de ces nœuds peut être associé des configurations alternatives (un nœud terminal peut être changé dynamiquement en un nœud non-terminal ou encore en générateur) qui peuvent être choisies dynamiquement, et même au cours du parcours du graphe. Cette fonctionnalité permet de capturer différents aspects d'un format de données au sein d'un même modèle, tout en offrant la possibilité de ne travailler que sur un seul aspect à la fois. Elle peut s'avérer utile également pour l'absorption. En effet, cette opération peut réclamer une modélisation différente de certains aspects du format de données par rapport à la stratégie de génération. Les configurations alternatives des nœuds rendent possible l'agrégation de ces différences au sein d'un même modèle.

Enfin, il est également possible d'associer (statiquement et dynamiquement) une sémantique aux nœuds du graphe ou des méta-informations quelconques, rendant possible des altérations plus évoluées sur le modèle (*cf.* section 4 page 6).

### 3.2 Description d'un format de données

La création d'un modèle de données peut être effectuée en utilisant les primitives bas-niveaux de `fuddly`, ou plus simplement en utilisant l'infrastructure de fuddly qui s'occupe de créer le modèle à partir d'une description structurée (ressemblant à une notation JSON). Pour des cas très complexes, il sera judicieux de mêler les deux approches. Il est également possible de profiter d'un modèle de données pour l'intégrer dans un autre (par exemple, pour ajouter le modèle d'une image JPG à celui d'un PDF).

Pour illustrer cela, nous présentons tout d'abord une description très simplifiée d'une donnée de type `PNG` (*Portable Network Graphics*) :

```
1  png_desc = \
2  {'name': 'PNG_model',
3   'contents': [
4       {'name': 'sig',
5        'contents': String(val_list=[b'\x89PNG\r\n\x1a\n'], size=8)},
6       {'name': 'chunks',
7        'qty': (2,200),
8        'contents': [
9             {'name': 'len',
10             'contents': UINT32_be()},
11            {'name': 'type',
12             'contents': String(val_list=['IHDR', 'IEND', 'IDAT', 'PLTE'], size=4)},
13            {'name': 'data_gen',
14             'type': MH.Generator,
15             'contents': lambda x: Node('data', value_type= \
16                                        String(size=x[0].get_raw_value())),
17             'node_args': ['len']},
18            {'name': 'crc32_gen',
19             'type': MH.Generator,
20             'contents': g_crc32,
21             'node_args': ['type', 'data_gen'],
22             'clear_attrs': [NodeInternals.Freezable]}
23       ]}
24  ]}
```

**Listing 1.1.** Représentation basique d'un PNG

Sans entrer dans les détails, on remarque que le graphe a pour nœud initial `'PNG_model'`, parent d'un nœud terminal `'sig'` (lignes 4-5) qui représente la signature d'un fichier `PNG`, et d'un nœud non-terminal `'chunks'` (lignes 6-23) qui représente les *chunks* du fichier.[5] Ce dernier nœud capture la structure d'un fichier `PNG` en définissant le contenu d'un *chunk* (lignes 9-22)[6] et le nombre de chunks que l'on peut retrouver dans un fichier `PNG` (de 2 à 200[7], ligne 7).

5. Ces *chunks* sont des blocs d'information qui structurent les fichiers `PNG`.
6. Dans cette modélisation très simplifiée, aucune distinction n'est effectuée sur la nature des *chunks*. Mais cette modélisation peut être enrichie très facilement.
7. La limite de 200 a été choisie artificiellement dans cet exemple.

Afin d'illustrer un certain nombre des possibilités offertes par `fuddly` pour décrire un format de données, nous abordons ci-dessous, un autre exemple, n'ayant pour seul intérêt que de mêler ces différentes constructions :

```
model_desc = \
{'name': 'TestNode',
 'contents': [
     # bloc 1
     {'section_type': MH.Ordered,
      'duplicate_mode': MH.Copy,
      'contents': [
          {'contents': BitField(subfield_sizes=[21,2,1], endian=VT.BigEndian,
                                subfield_val_lists=[None, [0b10], [0,1]],
                                subfield_val_extremums=[[0, 2**21-1],None,None]),
           'name': 'val1',
           'qty': (1, 5)},

          {'name': 'val2'},

          {'name': 'middle',
           'mode': MH.NotMutableClone,
           'contents': [
              {'section_type': MH.Random,
               'contents': [

                   {'contents': String(val_list=['OK', 'KO'], size=2),
                    'name': 'val2'},

                   {'name': 'val21',
                    'clone': 'val1'},

                   {'name': 'USB_desc',
                    'export_from': 'usb',
                    'data_id': 'STR'},

                   {'type': MH.Leaf,
                    'contents': lambda x: x[0] + x[1],
                    'name': 'val22',
                    'node_args': ['val1', 'val3'],
                    'mode': MH.FrozenArgs}
               ]}]},

          {'contents': String(max_sz = 10),
           'name': 'val3',
           'sync_qty_with': 'val1',
           'alt': [
               {'conf': 'alt1',
                'contents': SINT8(int_list=[1,4,8])},
               {'conf': 'alt2',
                'contents': UINT16_be(mini=0xeeee, maxi=0xff56),
                'determinist': True}]}
      ]},

     # bloc 2
     {'section_type': MH.Pick,
      'weights': (10,5),
      'contents': [
          {'contents': String(val_list=['PLIP', 'PLOP'], size=4),
           'name': ('val21', 2)},

          {'contents': SINT16_be(int_list=[-1, -3, -5, 7]),
           'name': ('val22', 2)}
      ]}
 ]}
```

**Listing 1.2.** Exemple du modèle TestNode

En premier lieu, nous constatons que ce modèle décrit un format composé de deux parties, le “bloc 1” (ligne 5-48) et le “bloc 2” (ligne 51-59). Au sein de ces blocs, des constructions variées sont utilisées. Nous les présentons succinctement ci-dessous :

**ligne 5, ligne 19, ligne 51 :** Le mot-clé `section_type` permet de choisir l'ordre imposé par un nœud non-terminal à ses fils. `MH.Ordered` positionne les fils dans l'ordre de la description. `MH.Random` ordonne les fils de façon aléatoire (sauf si le nœud parent est déterministe). `MH.Pick` ne conserve qu'un seul nœud parmi les fils ; le choix est effectué avec un aléa (sauf si le nœud parent est déterministe) suivant le poids associé à chacun des fils (`weights`, ligne 52).

**ligne 8-12 :** Un nœud terminal à valeur typée est défini. Il s'agit d'un `BitField`. Ce nœud possède une autre caractéristique via le champ “`'qty': (1,5)`” (ligne 12). Il signifie que son nœud parent peut le générer de 1 à 5 fois. [8]

**ligne 14 :** Cette construction permet d'utiliser un noeud déjà défini par ailleurs. Dans notre cas, il s'agit de `val2` défini par les lignes 22 à 23.

**ligne 25-26 :** Cette construction sert à créer un nœud en copiant un nœud existant via le mot-clé `clone`.

**ligne 28-30 :** Les mots-clés `export_from` et `data_id` servent à exporter un type de donnée issu d'un autre modèle (dans cet exemple, il s'agit d'une donnée de type `STRING Descriptor` issue du modèle `USB`).

**ligne 32-36 :** Il s'agit ici d'un nœud terminal de type *fonction* (variation d'un nœud *générateur*), prenant en paramètre les nœud `val1` et `val3`.

**ligne 42-47 :** Deux configurations alternatives au nœud `val3` sont définies par cette construction.

**ligne 41 :** Le mot-clé `sync_qty_with` permet de synchroniser le nombre de nœud à générer ou à absorber avec celui dont le nom est précisé. Dans cet exemple il s'agit du nœud `val1` définit par les lignes 8 à 12.

Enfin, la représentation visuelle d'une instance de modèle (c'est-à-dire d'une donnée issue du modèle décrit) est réalisée en ASCII-art par `fuddly`. La figure 1 présente une donnée issue du modèle précédent.

```
[TestNode]
----------
[0] TestNode [NonTerm]
 \__(1) TestNode/val1 [BitField] size=3B
 |        \_ (+|2: 1 |1: 10 |0: 00111000011111100110 |-) 13043686
 |        \_raw: b'\xc7\x07\xe6'
 \__(1) TestNode/val1:2 [BitField] size=3B
 |        \_ (+|2: 0 |1: 10 |0: 00100110010100100011 |-) 4351267
 |        \_raw: b'Be#'
 \__(1) TestNode/val2 [String] size=2B --> M
 |        \_raw: b'OK'
 \__[1] TestNode/middle [NonTerm]
 |   \__(2) TestNode/middle/val22 [Func | node_args: TestNode/val1, TestNode/val3] size=13B
 |   |        \_raw: b'\xc7\x07\xe6.vHddW.7Kt'
 |   \__[2] TestNode/middle/USB_desc [NonTerm]
 |   |   \__[3] TestNode/middle/USB_desc/bLength [GenFunc | node_args: TestNode/middle/USB_desc/contents]
 |   |   |   \__(4) TestNode/middle/USB_desc/bLength/dyn [UINT8] size=1B
 |   |   |            \_ 20 (0x14)
 |   |   |            \_raw: b'\x14'
 |   |   \__(3) TestNode/middle/USB_desc/bDescType [UINT8] size=1B
 |   |   |        \_ 3 (0x3)
 |   |   |        \_raw: b'\x03'
 |   |   \__(3) TestNode/middle/USB_desc/contents [String] size=18B
 |   |            \_raw: b'b\x00l\x00a\x00b\x00l\x00a\x00.\x00.\x00.\x00'
 |   \__(2) TestNode/middle/val21 [BitField] size=3B
 |   |        \_ (+|2: 1 |1: 10 |0: 1000110100011100011100 |-) 13738892
 |   |        \_raw: b'\xd1\xa3\x8c'
 |   \__(2) TestNode/middle/val2 [String] size=2B --> M
 |            \_raw: b'OK'
 \__(1) TestNode/val3 [String] size=10B
 |        \_raw: b'.vHddW.7Kt'
 \__(1) TestNode/val3:2 [String] size=10B
 |        \_raw: b'.vHddW.7Kt'
 \__(1) TestNode/val21 [String] size=4B
          \_raw: b'PLIP'
```

**Figure 1.** Exemple d'affichage d'une instance du modèle de données artificiel TestNode

8. À noter que cette caractéristique contraint également l'absorption d'une donnée existante.

La vue du graphe (*cf.* figure 1) étant présentée sous forme éclatée, des motifs `--> M` sont ajoutés pour matérialiser les nœuds ayant plusieurs arcs entrants.

## 4 La manipulation des données

`fuddly` définit des primitives pour la manipulation des données modélisées. Certaines facilitent la recherche (sémantique ou syntaxique) au sein de cette donnée, d'autres servent à sa modification (aussi bien dans sa structure que dans son contenu), tout en laissant le choix à l'utilisateur de préserver les contraintes du modèle. Par exemple, la simple modification du contenu d'un fichier dans une archive `ZIP`, modélisée dans `fuddly`, se résume à la modification d'un nœud du graphe qui représente l'archive. L'ensemble des contraintes associées est ensuite recalculé automatiquement par `fuddly` (comme par exemple les champs `compressed_size`, `crc`, les *offsets* des fichiers suivants, *etc.*). Cet exemple est illustré par l'extrait de code 1.3 suivant :

```
1 abszip = dm.get_data('ZIP')
2 abszip.set_current_conf('ABS', recursive=True)
3 abszip.absorb(zip_buff, constraints=AbsNoCsts(size=True,struct=True)
4
5 abszip['ZIP/file_list/file:2/data'].absorb(b'TEST', constraints=AbsNoCsts())
6 abszip.unfreeze(only_generators=True)
7 abszip.get_value()
```

**Listing 1.3.** Exemple d'absorption et de manipulation d'une archive ZIP

La première ligne permet de récupérer le modèle d'une archive ZIP. La ligne 2 change la configuration du modèle pour autoriser l'absorption. La ligne 3 effectue l'absorption d'un ZIP existant (`zip_buff` est la variable contenant l'archive), alors que la ligne 5 modifie le contenu du deuxième fichier. Enfin, le recalcul du ZIP est réalisé aux lignes 6 et 7.

Les manipulations sur une donnée peuvent être mises en œuvre au sein d'entités, appelées "disrupteurs", briques de base dans l'automatisation du fuzzing. Un certain nombre de disrupteurs génériques, implémentés dans fuddly, sont applicables à n'importe quel modèle de données. D'autres sont par contre spécifiques à un modèle en particulier.

Les disrupteurs génériques implémentés peuvent être paramétrés pour effectuer des modifications suivant différents critères (chemin dans le graphe, propriétés syntaxiques ou sémantiques, *etc.*). L'ordre du parcours peut également être affecté par des méta-informations éventuellement renseignées dans le modèle, comme le "niveau d'intérêt" des nœuds (capturant l'intuition de l'évaluateur, quant à l'impact sur la cible, qu'entraînerait la modification de ces nœuds). Parmi ces disrupteurs, certains conservent un état d'exécution d'un appel sur l'autre. Ils prennent en entrée, lors du premier appel, une donnée modélisée, et fournissent ensuite après chaque appel une nouvelle donnée issue de celle-ci. Citons par exemple :

- `tTYPE` qui consomme les nœuds terminaux à variables typées, et qui génère de nouvelles données en fonction des informations fournies dans le modèle, qu'elles soient de nature syntaxique (champ de 4 bits dans un `BitField`) ou sémantique (seules les valeurs 1 et 3 ont un sens dans ce champ de 4 bits) ;
- `tALT` qui itère sur les potentielles configurations alternatives des nœuds du graphe.

D'autres disrupteurs ne conservent aucun état. À partir d'une donnée fournie en entrée, ils produisent une nouvelle donnée unique. Citons par exemple :

- `STRUCT` qui procède à des modifications structurelles du graphe ;
- `C` qui peut corrompre de façon basique (*bitflip*) un ou plusieurs nœuds du graphe ;
- `SIZE` qui sert à tronquer les données ;
- `EXT` qui exporte la donnée fournie en entrée (ou une partie de la donnée) à un outil externe et en récupère la sortie.

Enfin, concernant les disrupteurs à état, leur implémentation peut s'appuyer sur une infrastructure de parcours des données modélisées (cas des disrupteurs `tTYPE` et `tALT` cités en exemple). Cette infrastructure met en œuvre un algorithme paramétrable de parcours de graphe. À chaque étape de ce parcours l'algorithme fait appel à un objet "visiteur-consommateur" (fourni préalablement en paramètre). Ce dernier dispose alors de la possibilité de consulter le nœud sur lequel il est appelé, d'effectuer autant de modifications qu'il le souhaite (chaque modification entraînant la génération d'une nouvelle donnée), avant de redonner la main à l'algorithme de parcours. Cet algorithme pourra alors réévaluer le parcours du graphe suivant les modifications engendrées (par exemple, si la modification a altéré un nœud non-terminal, alors que l'algorithme venait de parcourir tous les nœuds fils, il peut décider de parcourir à nouveau les nœuds fils qui sont à présent peut être différents).

## 5 L'automatisation du processus de fuzzing

L'automatisation du processus de fuzzing s'effectue dans fuddly à deux niveaux : d'une part, au travers du chaînage des disrupteurs, et d'autre part, via une infrastructure d'opérateur virtuel.

### 5.1 Le chaînage des disrupteurs

Les disrupteurs peuvent être chaînés entre eux, qu'ils soient génériques ou spécifiques, ou encore avec ou sans état. `fuddly` se charge de dérouler les opérations dans l'ordre demandés et d'alimenter les disrupteurs au besoin.

Dans l'exemple suivant une donnée modélisée est fournie au disrupteur `tALT`. Ce dernier, après chaque appel, change la configuration d'un nœud ayant une configuration alternative `'alt2'` (et restaure le précédent). Cette donnée modifiée est alors fournie à `C` qui effectue un *bitflip* sur un nœud choisi au hasard, puis ensuite à `tTYPE` qui va altérer les nœuds à variable typée (étape après étape), pour qu'au final le résultat soit passé à `SIZE` dont le rôle est de tronquer la donnée à 256 octets :

```
étape 1: Donnée --> tALT(conf='alt2') --> C(nb=1) --> tTYPE --> SIZE(sz=256) --> Donnée altérée 1
étape 2:                                              tTYPE --> SIZE(sz=256) --> Donnée altérée 2
[...]
étape N:                                              tTYPE --> SIZE(sz=256) --> Donnée altérée N
```

À l'issue de la première étape, `tTYPE` est alimenté et capable ainsi de générer un certain nombre de données jusqu'à l'étape `N`. À cette étape, le disrupteur `tTYPE` notifie à `fuddly` qu'il en a terminé avec la donnée fournie en entrée. `fuddly` s'occupe alors de réactiver le premier producteur de donnée qu'il trouve, capable de générer de nouvelles données, afin de continuer à alimenter `tTYPE`. En l'occurrence, il s'agit de `tALT`. La suite du processus de fuzzing se déroule alors de la façon suivante :

```
étape N+1:           tALT(conf='alt2') --> C(nb=1) --> tTYPE --> SIZE(sz=256) --> Donnée altérée N+1
étape N+2:                                             tTYPE --> SIZE(sz=256) --> Donnée altérée N+2
[...]
étape N+M:                                             tTYPE --> SIZE(sz=256) --> Donnée altérée N+M
```

Les étapes `N` et `N+1` sont illustrées par la figure 2, où `N` est dans cet exemple égal à 76.

### 5.2 les opérateurs virtuels

Le processus de fuzzing peut être implémenté au sein d'un opérateur virtuel, depuis lequel est réalisée la planification des opérations à effectuer sur la cible (type de données à envoyer, type de modifications à appliquer avant l'envoi, *etc.*). L'opérateur est par défaut rappelé après chaque émission de données vers la cible, mais il peut également fournir un lot d'instructions à exécuter avant d'être rappelé. Cette approche autorise la stimulation simultanée d'une cible sur des entrées différentes, chacune assujettie à un protocole spécifique. Avant de décider de la marche à suivre, l'opérateur peut s'appuyer sur l'infrastructure de *monitoring*[9] de *fuddly*, pour déterminer l'état de la cible, ainsi que pour récupérer ses réponses. De plus, suivant la complexité des protocoles en jeu, et des choix à effectuer par l'opérateur, il sera judicieux de s'appuyer sur une bibliothèque de machines à états telle que `toysm` [6] (publicité éhontée à la production d'un collègue).

9. L'infrastructure de *monitoring* autorise la mise en œuvre de sondes indépendantes pour mesurer ou surveiller des paramètres quelconques liés ou non à la cible.

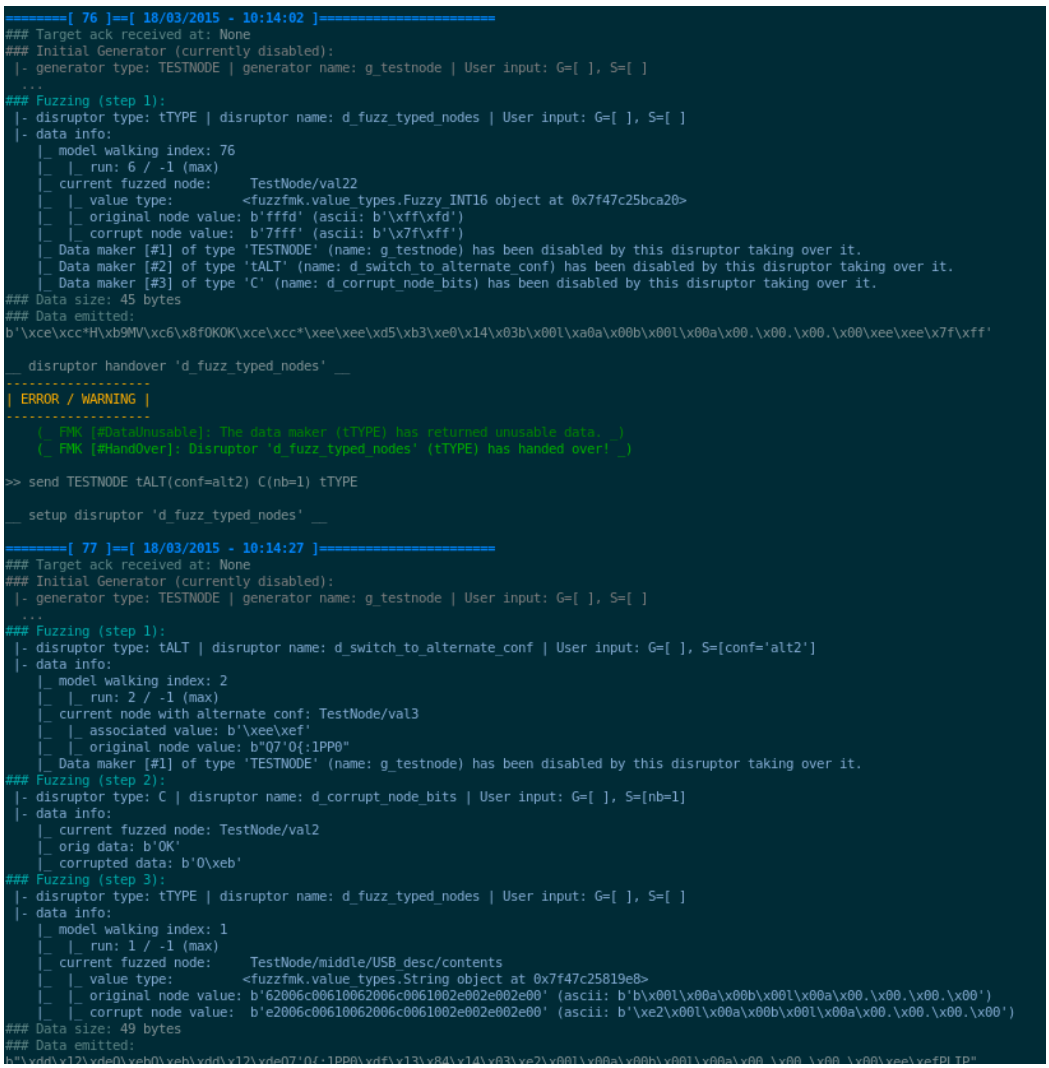


**Figure 2.** Aperçu d'exécution d'une séquence de disrupteurs

## 6 Conclusion et perspectives

Dans ce papier, nous n'avons fait qu'effleurer les points saillants de `fuddly`. Bien que les fonctionnalités offertes par ce *framework* soient totalement utilisables, `fuddly` reste un terrain d'expérimentation, et au-delà de l'implémentation de nouveaux modèles de données, de nombreuses évolutions sont prévues, notamment :

- la définition de nouveaux disrupteurs agissant sur des propriétés syntaxiques et sémantiques des modèles de données ;
- des fonctions génériques qui régissent des relations basiques entre nœuds (longueur, compteur, *etc.*), utilisables directement par les nœuds de type *générateur* ;
- la définition d'un disrupteur à état sur la base de `EXT`, afin de bénéficier d'outils de fuzzing existants tel que `radamsa` [7] ;
- l'intégration du DBI (Dynamic Binary Instrumentation) `PIN` [8] de Intel dans l'infrastructure de *monitoring*, pour les cibles s'exécutant sur architecture x86 ;
- un nouveau type de nœud permettant de définir des parties d'un modèle sous forme de contraintes (programmation par contraintes via `python-constraint` [9] par exemple).

## Références

1. Pedram Amini et Aaron Portnoy. sulley. https://github.com/OpenRCE/sulley.
2. Michael Eddington. Peach. http://sourceforge.net/projects/peachfuzz/.
3. Gabriel Campana. Fuzzgrind. http://esec-lab.sogeti.com/pages/Fuzzgrind.
4. Will Drewry et Tavis Ormandy. Flayer : Exposing application internals. *WOOT*, 7 :1–9, 2007.

5. Michal Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.
6. William Barsse. toysm. https://github.com/willakat/toysm.
7. Oulu University Secure Programming Group. Radamsa. https://github.com/aoh/radamsa.
8. Intel Corporation. Pin - a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
9. Gustavo Niemeyer. python-constraint. http://labix.org/python-constraint.