Rapport de projet long
Exploration de vulnérabilités dans les jeux vidéo pour la triche et le détournement de l'exécution.

Louis Blenner, Mahé Tardy

14 mars 2021

Table des matières

Abstract

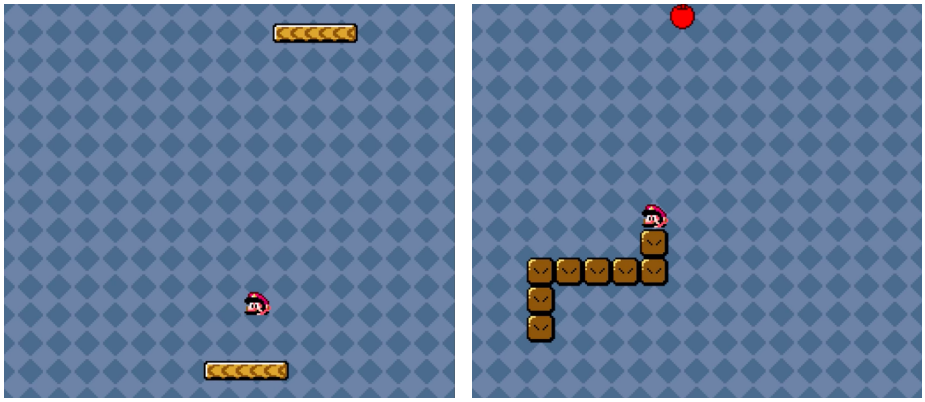
Lors de ce projet, nous avons tenté de répondre à la question suivante : « peut-on injecter du code et prendre le contrôle de l'exécution d'un jeu vidéo uniquement par ses éléments de *gameplay* ? ». La réponse est un franc « oui » et durant ce projet, nous avons découvert tout l'univers des *tool-assisted speedrun* [1] et sa communauté. De l'injection manuelle de Flappy Bird dans Super Mario World [1] au passage d'un appel Skype via Zelda sur une Super Nintendo Entertainment System [2] [3], les possibilités offertes sont infinies et les performances réalisées incroyables.

1. Le *speedrun* est l'activité qui consiste à tenter de finir un jeu aussi vite que possible. Le *tool-assisted speedrun* est une catégorie qui permet l'utilisation de divers outils logiciles pour créer le *run* parfait qui va être exécuté par un ordinateur.

1 Introduction

L'injection de code est une affaire de gestion des entrées utilisateur, qui peuvent être potentiellement malveillantes. Une injection dans un tableau en C prévu peut pour contenir une entrée utilisateur peut permettre de détourner l'exécution en écrasant la pile, une injection dans un champ HTML peut permettre l'exécution d'une commande SQL malveillante. Nous nous sommes alors posé la question : est-ce qu'un joueur peut injecter du code et prendre le contrôle de l'exécution en se servant uniquement des outils de gameplay que le jeu lui met à disposition ?

La réponse est « oui », et le premier jeu dont nous avons personnellement découvert des *runs*[2] qui détournent l'exécution via ce moyen est Super Mario World, un jeu de Super Nintendo Entertainment System (SNES) de 1990. Nous avons découvert un premier run par Masterjun dans lequel il réalise une série de glitchs[3] qui lui permettent de passer la majeure partie du jeu pour arriver directement à la scène finale et ainsi finir le jeu extrêmement vite [4]. Nous avons ensuite un autre *run* sur Super Mario World où le *runner* injecte davantage que le code suffisant pour déclencher la fin du jeu : il injecte d'autres jeux, une version de pong et de snake [5], voir figure 1. Il est désormais possible de finir Super Mario World dans un temps records tout en jouant à pong et snake, donc avec une classe absolue.



(a) Mario Pong (b) Mario Snake

FIGURE 1 – Pong et Snake injectés dans Super Mario World.

Via ce premier jeu et ces deux *speedruns* nous avons découvert la communauté du *Tool Assisted Speedrun* (TAS). Cette communauté se rassemble notamment sur le fabuleux site `http://tasvideos.org/` « *When human skills are just not enough* » qui regroupe beaucoup de documentation et recommandations sur les outils à utiliser et les soumissions des *speedruns*.

Pour préparer ce genre de *runs*, les *runners* confectionnent la série d'entrées utilisateurs la plus optimale sur un émulateur sur lequel ils peuvent librement

2. Un *run* est une partie jouée du jeu.

3. Un glitch est un bug dans un jeu vidéo, qui peut permettre par exemple de « *clipper* », c'est-à-dire passer à travers un mur, ou éviter le déclenchement de certains scripts en jeu. Plus généralement c'est un bug qui permet de tricher sans faire planter l'exécution du jeu.

lire toute la RAM, debugger et faire tourner des scripts LUA [6]. Ensuite, ils enregistrent la série d'entrées utilisateur et forment ce qu'on appelle un *movie*, qui contient sur quelle touche l'utilisateur a appuyé *frame*[4] par *frame* en plus d'autres meta-données comme des sous-titres par exemple [7] [8].

Il est alors possible, en récupérant cette série d'entrées utilisateur de les jouer sur un vraie console, via un contrôleur modifié. C'est ce que fait le célèbre TASBot [3], il s'agit d'un Raspberry Pi déguisé en R.O.B (voir figure 2) capable d'émuler un contrôleur en envoyant les entrées utilisateurs de manière très précise à la console. En 2017, TASBot a réalisé l'exploit de jouer à Super Mario 64 et Portal sur la SNES et également de passer un appel Skype (!) avec une bande passante qui permettait d'afficher une vidéo à 10 images par secondes en 128x112 [2].

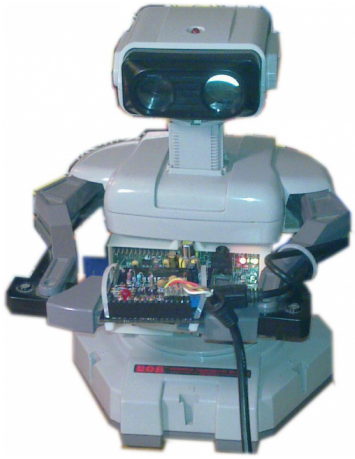


FIGURE 2 – TASBot

Ce qui avait été fait sur Super Mario World est de l'injection réalisée sur émulateur et sur console réelle à partir de programmes qui jouent les entrées utilisateur. Nous avons découvert un *run* du *youtubeur*, *streamer* et *speedrunner* Seth Bling qui a décidé d'injecter le code *Flappy Bird* (optimisé sur 331 octets) à la main, octet par octet, sur une vraie console. Seth Bling réussit en 50 min à injecter et exécuter un *Flappy Bird*, entièrement jouable [1].

Nous avons aussi découvert d'autres *glitchs* dans d'autres jeux qui permettent de détourner l'exécution du code vraiment facilement. Le cas de la « *Memory exploration* » [9] de Super Mario Land 2 est particulièrement hallucinant. Il permet littéralement de se balader dans la mémoire de l'intégralité de la console (en l'occurrence le *Gameboy*) comme s'il s'agissait d'un niveau du jeu.

Le jeu sur lequel nous nous sommes arrêtés est Super Mario Bros 3 de la NES (1988), un bug intitulé « *Wrong warp* » permet de faire planter le jeu très astucieusement et d'exécuter une zone mémoire manipulable par le joueur [10]. Ce bug a été exploité de la même manière que celui de Super Mario World, des premiers *speedrunners* ont réussi à sauter à la scène finale du jeu [11] et d'autres à injecter du code arbitraire pour prendre le contrôle total de l'exécution du jeu

4. Une *frame* est une image du jeu.

[12]. Nous expliquerons le fonctionnement de ce bug en détail dans les parties suivantes.

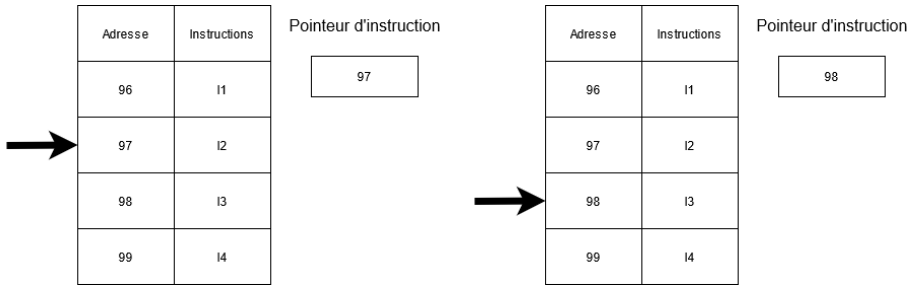
2 Vulgarisation

Cette partie permet à quelqu'un sans un gros bagage technique de comprendre le déroulement de notre projet et nos objectifs.

2.1 Le pointeur d'instruction

Notre but est de prendre le contrôle de la machine sur laquelle tourne un jeu pour pouvoir lui faire exécuter du code arbitraire. Pour cela nous allons utiliser Super Mario Bros 3 sur NES car il est connu comme possédant des vulnérabilités permettant d'atteindre cet objectif.

Pour expliquer la méthode que l'on va appliquer, il faut comprendre ce qu'est le pointeur d'exécution. Lorsque l'on fait tourner un jeu vidéo ou plus généralement un programme, la machine qui l'exécute va lire les instructions décrites par le code l'une après l'autre. Pour savoir quelle instruction la machine doit exécuter à chaque étape, elle se souvient de la position de l'instruction à exécuter dans un registre. On appelle le contenu de ce registre le pointeur d'instruction.



(a) Exécution de l'instruction 2 (b) Exécution de l'instruction 3

FIGURE 3 – Le pointeur d'instruction lors de l'exécution d'un programme

Ainsi, dans la situation décrite par la figure 3, on augmente le pointeur d'instruction de un pour passer à l'instruction suivante. Cependant, il arrive aussi qu'on le modifie complètement pour aller exécuter d'autres parties du code.

Notre but est de faire en sorte que la machine se mette à exécuter des instructions que nous avons écrites. Pour cela il faut faire deux choses : écrire les instructions que l'on a envie d'exécuter quelque part dans la mémoire et faire en sort que le pointeur d'instruction se mette à indiquer l'endroit que l'on a modifié.

2.2 Écriture des instructions

Lorsque l'on exécute un programme, celui-ci va utiliser de la mémoire pour stocker différentes valeurs nécessaires à son fonctionnement. Dans notre cas, avec Super Mario Bros 3, le jeu va devoir stocker des choses comme la position de Mario, sa vitesse ou bien le temps passé dans le niveau.

En particulier, nous avons 5 adresses les une à la suite des autres qui contiennent les positions des 5 potentiels ennemis qui peuvent se trouver à l'écran. Et, si un des ennemis venait à mourir (et donc disparaître de l'écran), sa position ne serait pas modifiée jusqu'à ce qu'un nouvel ennemi apparaisse à l'écran. De cette manière, il possible de tuer différents ennemis à des positions bien particulières pour écrire des instructions précises dans ces zones mémoires.

Nous allons donc, par des éléments du *gameplay* du jeu, écrire un programme compréhensible par la console en manipulant des ennemis.

2.3 Détournement du pointeur

Lorsque le jeu Super Mario Bros 3 est lancé, le pointeur d'instructions va pointer vers des instructions qui représentent le code du jeu. Heureusement pour nous, ce code n'est pas infaillible. Il contient des bugs qui peuvent amener le pointeur d'instruction à se trouver dans un endroit imprévu. Dans la plupart des cas, cela provoque le crash du jeu car la zone mémoire pointée contient des données qui ne représentent pas du code, c'est-à-dire que la console ne peut pas interpréter et exécuter.

Dans le cas du bug que nous avons utilisé, il est possible de placer le pointeur d'exécution très proche de l'adresse de la position des ennemis. Étant donné que l'on peut contrôler cette plage mémoire, on peut donc faire exécuter quelques instructions à la console.

2.4 Code arbitraire

Dans l'état actuel des choses, deux problèmes se posent avec cette approche. Premièrement la taille des programmes que l'on peut écrire est extrêmement petite : 5 zones mémoire pour les 5 positions des ennemis plus 5 autres contenant leurs compteurs de pas. Deuxièmement, nous sommes obligés d'élaborer et de mettre à exécution une stratégie pour éliminer les ennemis à des positions et des timings particuliers pour chaque programme que l'ont voudrait exécuter.

Pour pallier à ces deux problèmes, nous allons écrire dans la zone mémoire que nous contrôlons un programme qui aura pour fonction de lire les boutons enclenchés sur le contrôleur et d'écrire en mémoire la valeur correspondant à la combinaison à chaque image du jeu. Ce programme va ensuite lancer l'exécution de la mémoire qui vient d'être écrite.

Grâce à cette technique, nous allons pouvoir modifier toute la mémoire de la machine en appuyant successivement sur les bonnes combinaisons de boutons. Nous allons aussi pouvoir enregistrer une unique combinaison de commande qui permet d'écrire ce programme et de le lancer en jouant à Mario. Il nous suffira alors d'ajouter à la fin de cette suite de commande la liste des commandes qui correspondent au programme que l'on veut exécuter et nous obtiendront son exécution. Nous avons pour cela écrit un programme qui prend un code et le transforme en une liste de combinaison de pression de boutons.

3 Détails

Cette partie explique le projet avec davantage de précisions et de détails techniques.

3.1 Architecture de la Nintendo Entertainment System

La NES est sortie en 1983 et a eu un impact considérable sur l'histoire des consoles et du jeu vidéo. Nous allons nous intéresser au côté technique de la console, et plus précisément de sa mémoire et de son CPU. Nous n'aborderons pas le fonctionnement de la *Picture Processing Unit* (PPU) [13] et *l'Audio Processing Unit* (APU).

Le processeur de la NES est un `2A03` produit par Ricoh, il se base sur le jeu d'instruction du `6502` [5] et a un bus d'adresse 16 bits avec un accumulateur 8 bits, 2 registres X, Y, 8 bits et 8 bits de *flag* processeur [14]. Le 6502 est un processeur *little endian*, utilise des instructions codées sur 1, 2 ou 3 octets en fonction de l'opération réalisée et possède 13 types d'adressage différents (absolue, indirect, relatifs, zeropage, indexé par un registre...) [15]. C'est un processeur assez simple : il possède 56 instructions différentes mais dont les variations dues aux modes d'adressages font qu'il existe 151 *opcodes* différents.

Concernant la mémoire de la console, elle possède 2K octets de RAM interne dédiée au CPU mais tout le reste (par exemple mémoire et registres de la PPU, la ROM...) est mappé sur la plage d'adresse du bus 16 bits. On peut voir une carte donnant un certain nombre de détails figure 4 [16]. On y voit par exemple où est la pile de la console, où sont mappés la ROM, les registres d'input/output. C'est ensuite au jeu de faire leur RAM map sur la mémoire interne de la console, certaines personnes ont reversé la cartographie mémoire et l'ont rendu disponible, notamment pour Super Mario Bros 3 [17].

5. Le jeu d'instruction 6502 est également utilisé par des ordinateurs de la famille des Apple II puis dans des Commodore, ACORN, Atari, BBC micro...

Address Range	Size	Notes (Page size is 256 bytes)	
$0000–$00FF	256 bytes	Zero Page — Special Zero Page addressing modes give faster memory read/write access	
$0100–$01FF	256 bytes	Stack memory	
$0200–$07FF	1536 bytes	RAM	
$0800–$0FFF	2048 bytes	Mirror of $0000–$07FF	$0800–$08FF Zero Page $0900–$09FF Stack $0A00–$0FFF RAM
$1000–$17FF	2048 bytes	Mirror of $0000–$07FF	$1000–$10FF Zero Page $1100–$11FF Stack $1200–$17FF RAM
$1800–$1FFF	2048 bytes	Mirror of $0000–$07FF	$1800–$18FF Zero Page $1900–$19FF Stack $1A00–$1FFF RAM
$2000–$2007	8 bytes	Input/Output registers	
$2008–$3FFF	8184 bytes	Mirror of $2000–$2007 (multiple times)	
$4000–$401F	32 bytes	Input/Output registers	
$4020–$5FFF	8160 bytes	Expansion ROM — Used with Nintendo's MMC5 to expand the capabilities of VRAM.	
$6000–$7FFF	8192 bytes	SRAM — Save Ram used to save data between game plays.	
$8000–$FFFF	32768 bytes	PRG-ROM	
$FFFA–$FFFB	2 bytes	Address of Non Maskable Interrupt (NMI) handler routine	
$FFFC–$FFFD	2 bytes	Address of Power on reset handler routine	
$FFFE–$FFFF	2 bytes	Address of Break (BRK instruction) handler routine	

FIGURE 4 – Carte mémoire de la NES

3.2 Les outils

3.2.1 Les émulateurs

Pour travailler sur ces jeux, nous avons utilisé des émulateurs. Ces programmes nous permettent de lire l'image d'un jeu (ROM) et d'émuler le fonctionnement de la console sur laquelle il devrait tourner. Non seulement ils nous permettent de jouer à ses jeux mais fournissent aussi des outils pour observer et contrôler l'exécution du jeu, on peut ainsi ralentir, accélérer l'exécution du jeu, la mettre en pause, débugger et modifier n'importe quel octet dans la mémoire.

Il existe une communauté qui développe ces outils et les utilise pour faire du *tool-assisted speedrun* (TAS). Le site `https://tasvideo.org` regroupe les soumissions et le classement des meilleurs TAS pour de nombreux jeux.

Le but d'un TAS est, en temps normal, de terminer un jeu le plus rapidement possible en s'aidant d'outils. Un TAS peut se définir simplement par une suite d'instructions qui décrivent pour chaque image du jeu quels boutons de la manette doivent être appuyés. Nous avons voulu enregistrer un TAS qui, au lieu d'essayer de terminer le jeu le plus rapidement possible, permettait d'exécuter du code arbitraire. La première étape a été de télécharger un TAS qui lance l'exécution de code arbitraire dans Super Mario Bros 3 [12] et de la lancer dans l'émulateur FCEUX pour constater son bon fonctionnement.

Nous avons ensuite découvert que cet émulateur en particulier n'était plus recommandé par TASVideo en faveur l'émulateur BizHawk. En effet, ces émulateurs sont imparfaits dans la simulation du comportement d'une vraie NES. Ce que nous avons pu constater en essayant de lancer le TAS précédent sur l'émulateur BizHawk. Dans cette configuration, le TAS n'a pas le comportement attendu,

Mario meurt dans le premier niveau et se retrouve bloqué dans le menu de changement de niveau.

3.2.2 Les formats

Par la suite nous avons utilisé les deux émulateurs, FCEUX et BizHawk qui utilisent des formats différents pour décrire un movie. Pour BizHawk, le format `.bk2` est utilisé [18], c'est une archive `zip` qui contient des fichiers textes décrivant le TAS. Voici les fichiers et leur contenu :

- `Header.txt` : contient des informations générales comme le hash de la ROM associée, la version de l'émulateur utilisé et le nom de l'auteur.
- `Input Log.txt` : contient la suite des boutons enclenchés à chaque *frame* du jeu.
- `Comments.txt` : contient des commentaires de l'auteur.
- `Subtitles.txt` : contient du texte informatif qui va s'afficher à l'écran lors de la lecture du *movie*.
- `SyncSettings.txt` : contient la configuration de l'émulateur à utiliser.
- `CoreState.txt` : contient l'état de la mémoire lorsque le TAS est lancé, si non présent, l'état initial de la console au démarrage est utilisé.

Pour FCEUX 2, le format `.fm2` est utilisé [19]. C'est un fichier texte qui va contenir les mêmes sections que précédemment avec des syntaxes légèrement différentes et séparées par une ligne vide. Voici les premières lignes d'un fichier `.fm2` d'exemple :

```
[Input]
LogKey:#Power|Reset|#P1 Up|P1 Down|P1 Left|P1 Right|P1
↪  Start|P1 Select|P1 B|P1 A|#P2 Up|P2 Down|P2 Left|P2
↪  Right|P2 Start|P2 Select|P2 B|P2 A|
|..|...R..B.|........|
|..|...R..B.|........|
|..|...R..B.|........|
|..|...R..B.|........|
|..|U.....BA|........|
|..|U.....BA|........|
|..|U.....BA|........|
|..|........|........|
|..|........|........|
```

Chaque ligne représente les boutons enclenchés à chaque *frame*, un point représente un bouton non enclenché. La première section de taille 2 représente les deux boutons se trouvant sur la console *POWER* et *RESET*, la seconde section représente la manette 1 et la seconde la manette 2. On peut voir que les boutons `A`, `B`, `U` (up) et `R` (right) sont enclenchés sur cet exemple.

En plus des outils fournit par les émulateurs, nous avons aussi utilisé une cartographie de la RAM [17] et un code source désassemblé [20] de Super

Mario Bros 3 afin de comprendre la signification des adresses mémoires et le fonctionnement du jeu dans les détails.

3.3 Fonctionnement de la vulnérabilité

La communauté des speedrunners s'intéresse aux bugs qui peuvent exister dans divers jeux. Un bug bien utilisé peut permettre de terminer un jeu bien plus rapidement. Ainsi la communauté recense et expérimente avec les bugs présents dans les jeux [21]. Dans le cas de Super Mario Bros 3, le record du monde actuel utilise un bug qui permet dans une situation particulière de lancer l'exécution d'une partie de la mémoire manipulable par le joueur. Le record mondial écrit dans cette mémoire une instruction qui permet de se rendre à la fin du jeu : l'affichage des crédits. C'est ce bug que nous allons .

Dans Super Mario Bros 3, Mario est capable d'entrer dans un tuyau en se plaçant en dessous ou au-dessus, en fonction de l'orientation, en appuyant sur haut ou bas. Le tuyau va ensuite téléporter Mario au niveau d'un autre tuyau correspondant. Or il s'avère que si Mario court vite sur le côté d'un tuyau, il est possible de déclencher l'activation du comportement d'un tuyau sans être en dessous ou au-dessus. Quand cela se produit, le code qui est censé téléporter Mario au niveau du tuyau de sortie fonctionne de manière imprévue et envoie le joueur ailleurs.

Lorsque ce bug est effectué dans le niveau 1 du monde 7 de Super Mario Bros 3, Mario se retrouve dans une zone mémoire qui n'est pas censée représenter un niveau. Le joueur se retrouve sur le 19e écran du niveau alors que le jeu ne prévoit que 16 écrans maximum par niveau. L'indexation du niveau étant anormale, il s'avère que le code qui gère certaines interactions du jeu va mal fonctionner. On a notamment l'affichage du niveau qui n'interprète pas la zone mémoire de la même façon que le reste du jeu et donc certains éléments à l'écran sont invisibles alors que d'autres ne sont que visibles sans interaction possible.

Dans notre cas, la zone mémoire dans laquelle Mario est envoyé contient un bloc « note » invisible et lorsque que le joueur interagit avec, le jeu va chercher l'image associée à l'animation du bloc mais va se tromper à cause de la mauvaise indexation de la l'écran du niveau et aller la chercher en `0x9C70`. Cette adresse pointe vers le mapper de mémoire de la ROM et le code qui gère l'animation du bloc note va déclencher le remplacement d'une zone mémoire de la RAM pour un morceau du code du jeu. Le rôle du mapper est de charger en RAM les données que l'ont a besoin depuis la ROM car la RAM n'est pas assez grande pour accueillir la totalité de la ROM.

On se retrouve à exécuter une partie du code jeu qui n'a rien à voir avec le bloc note. Or, dans notre cas, il se trouve qu'à ce moment, notre pile est presque vide. Elle se trouve à l'index `0xFD` sur `0xFF`. Le code qui a été remplacé va exécuter une première instruction `RTS`, qui consiste en le retour d'une routine, donc récupération des valeurs sur la pile pour restaurer le contexte et reprendre l'exécution au bon endroit. Cette action va vider la pile et lancer l'exécution d'un autre bout de code se trouvant à l'adresse `0x8F4D` qui se trouvait dans la pile. Ce bout de code va lui aussi exécuter une instruction `RTS` mais cette fois si, comme la pile est vide, un *stack underflow* va se produire et on va lire le haut de la pile.

Finalement, il s'avère que l'adresse du haut de la pile `0x0100` est utilisée par les développeurs pour stocker le type du niveau actuel : défilement vertical ou horizontal. Les développeurs sont partis du principe que la pile ne serait jamais aussi pleine et ont utilisé cet espace. Or comme le niveau 1 du monde 7 est un niveau vertical, la valeur qui est stockée à cet endroit est `0x0080`. L'instruction `RTS` va donc renvoyer l'exécution à cette adresse et c'est justement vers cette adresse que sont stockées les variables d'exécution du jeu comme la position horizontale de Mario ou des ennemies. Cet enchaînement nous permet donc d'exécuter une zone de mémoire que l'on va pouvoir manipuler.

3.4 Exploitation de la vulnérabilité

La vulnérabilité permet donc de déplacer le *program counter* jusqu'à l'adresse `0x0081` dans la RAM de la console. Or il s'avère, que certains octets non loin de cet espace sont manipulables indirectement par le joueur. En effet les 5 octets aux adresses `0x91` à `0x95` contiennent la position horizontale des ennemis à l'écran, et les 5 octets aux adresses `0x9A` à `0x9E` représentent les nombres de pas des ennemis. Ce sont exactement ces 10 octets qui vont permettre d'injecter du code dans le jeu.

De façon anecdotique, nous savons que l'octet à l'adresse `0x90` contient la position horizontale de Mario à l'écran, que `0x96` contient la position d'un *power-up* à l'écran et que `0x97` s'incrémente uniquement quand Mario tape un bloc brique contenant une pièce. Nous ignorons ce que contiennent les octets aux adresses `0x98` et `0x99` mais nous verrons que ce n'est pas très important car ces 2 vont être les premiers octets écrasés par notre injection d'octets. La figure 5 est une RAM watch, c'est-à-dire une sélection d'octets que nous pouvons surveiller grâce à l'émulateur pendant l'exécution du jeu, elle montre ce que nous pouvons observer lors d'un des *runs*.

Address	Value	Changes	Domain	Notes
0090	B0	40	RAM	Mario position
0091	50	134	RAM	Ennemy 5 position
0092	B0	40	RAM	Ennemy 4 position
0093	01	158	RAM	Ennemy 3 position
0094	B0	60	RAM	Ennemy 2 position
0095	1C	827	RAM	Ennemy 1 position
0096	00	289	RAM	Power up position
0097	08	4	RAM	Coin brick count
0098	08	4	RAM	
0099	08	191	RAM	
009A	12	1	RAM	Ennemy 5 steps
009B	10	3	RAM	Ennemy 4 steps
009C	30	13	RAM	Ennemy 3 steps
009D	00	209	RAM	Ennemy 2 steps
009E	00	84	RAM	Ennemy 1 steps

FIGURE 5 – RAM watch de la console

On remarque que, grâce au bug, nous allons décaler l'exécution à `0x0081` mais que notre code injecté commence à l'adresse `0x0091`. C'est un problème qui souvent, s'avère ne pas en être un, car par chance, ces 10 octets varient peu, leur interprétation sous forme de code ne perturbe que rarement l'exécution

et n'a souvent aucun effet de bord. Il arrive cependant lors de *speedrun* et de tentatives d'injection manuelles que ces 10 octets interrompent le *run* en faisant planter le jeu ou en décalant l'exécution. En effet, les instructions du 6502 sont à tailles variables et font 1, 2 ou 3 octets, une instruction mal placée peut décaler le *program counter* et rendre illisible un code écrit pour être aligné sur l'octet `0x0091`.

3.5 Le premier code injecté : le dropper

Les premiers *TAS* réalisant de l'injection de code avaient pour but principal de terminer le jeu le plus rapidement possible plutôt que de prendre le contrôle total du jeu pour détourner son exécution. C'est pourquoi les premiers *speedrunner* se sont contentés de tenter d'injecter trois octets pour se rendre à la scène de fin du jeu. Les octets utilisés sont ceux des trois premières positions horizontales des ennemis : `0x0093`, `0x0094` et `0x0095`. L'enjeu a été d'injecter les octets suivants `0x20`, `0xEF` et `0x8F`. En effet, le CPU interprète ces 3 octets, comme un *opcode* et une adresse absolue en *little endian*, ce qui donne la ligne d'assembleur suivante `JSR $8FE3`. L'*opcode* `JSR` est le *jump* dans une *subroutine*, il permet, en plus de déplacer le *program counter* à l'adresse absolue indiquée, de sauvegarder le contexte sur la pile dans le cadre d'un retour de la *subroutine*. À noter que c'est parfaitement inutile parce que nous ne retournerons jamais dans la RAM, mais que cette instruction a été choisie sans doute en raison de la facilité d'écrire un `0x20`. Enfin, cette adresse n'est pas choisie de manière anodine, `0x8FE3` contient le début de la *subroutine* affichant la scène finale à l'écran et donc « terminant » le jeu. Ce type de *speedrun* a d'abord été effectué en *TAS* mais a également une catégorie « humaine » de *speedrunner* qui réalise cet exploit à l'aveugle sur une vraie console [22].

Cependant, ce qui nous intéresse au-delà de terminer le jeu est de faire de l'injection de code arbitraire à plus grande échelle, et nous avons découvert un *TAS* qui réalise cet exploit [12] ! Le *speedrunner* a eu la gentillesse d'expliquer le bout de code qu'il injecte dans la soumission de son *TAS*. Il va falloir déplacer de manière très précise des *koopas*, notamment leur carapace et s'armer de patience pour injecter exactement ce code et remplir les 10 octets manipulables par le joueur :

```
$0091: JSR $96E5 ; subroutine that waits until frame ends;
↪  NMI updates controllers & $15 counter
$0094: LDA $F7   ; load controller 1 state
$0096: BRK       ; do nothing
$0098: BRK       ; do nothing
$009a: INX       ; increment X register
$009b: STA,X $7d ; store controller data to next spot in
↪  RAM
$009d: BNE $0091 ; if X isn't zero, loop to $0091
```

Nous allons détailler le comportement de ce code ligne par ligne, la numérotation commence à `0x0091` car c'est l'endroit où il va être placé dans la RAM de la console.

— `JSR $96E5` est explicitement commentée, cette ligne permet globalement de « faire tourner le jeu » avant de retourner à la ligne suivante en fin de routine. En effet rappelons que `JSR` déplace `pc` mais permet aussi la sauvegarde du contexte pour le retour de la *subroutine*.

— `LDA $F7` charge dans l'accumulateur la valeur se trouvant à l'adresse `0xF7`. C'est l'endroit où le pilote de contrôleur de Super Mario Bros 3 a écrit l'octet représentant les boutons appuyés. On peut noter l'adressage en *zeropage* : ce qu'on appelle les *zeropage* sont les adresses entre `0x0` et `0xFF`. Elles permettent d'accéder et d'écrire la mémoire en utilisant une adresse sur 1 seul octet, ce qui permet de gagner un cycle CPU. Elles sont abondamment utilisées, notamment comme des registres supplémentaires car à l'époque, les temps de cycles CPU et les temps d'accès à la mémoire sont quasi équivalents.

— Les deux `BRK` sont ici car ils correspondent à l'opcode `0x00`. Nous n'avons pas pu manipuler les octets aux adresses `0x96` à `0x99` inclus et les zéros nous ont sauvé. Ils ont le bon goût d'augmenter le *program counter* de deux et anecdotiquement de déclencher les interruptions logicielles qui n'ont pas d'effet ici.

— `INX` incrémente le registre X, cela va être très utile car l'opération suivante dépend de X pour la position de l'écriture de la valeur de l'accumulateur en RAM.

— `STA,X $7d` est une opération un peu plus complexe, elle stocke la valeur de l'accumulateur à l'adresse `0x7d` + la valeur contenue dans le registre X. On comprend ici que le bon déroulement dépend complètement de la valeur initiale contenue dans X, il s'avère qu'elle est plutôt déterministe à ce stade de l'exécution du jeu est qu'elle est à `0x19`. D'ailleurs l'operand `0x7d` est précisément choisi par rapport à cet offset pour tomber au bon endroit pour la première écriture.

— `BNE` 0091 est un branchement conditionnel qui permet de continuer l'exécution de notre boucle tant que la valeur contenue dans X est différente de 0. En réalité nous verrons que ça n'arrivera jamais et qu'on se servira uniquement de ce branchement pour boucler.

À partir de cette étape, les boutons pressés sur le contrôleur vont être cruciaux car ils vont permettre d'écrire du code injecté, octet par octet. Nous aborderons dans la partie suivante les problématiques que cela implique et comment nous avons fait pour écrire du code, l'assembler et le convertir en entrées contrôleurs de NES. Néanmoins, en faisant les calculs, en considérant qu'il y'a `0x19` dans X au tout début de l'exécution, on constate que le prochain octet allant être écrit par l'opération `STA,X $7d` est à l'adresse `0x1a` + `0x7d` = `0x97`. Ce premier octet est l'octet juste après un `BRK` donc il n'a pas d'importance, on peut y écrire n'importe quoi, comme `0x00` par exemple. Cependant on va vouloir réécrire ce qui se trouve à l'adresse `0x0098` pour y écrire deux octets `0xF0`, l'opcode de `BEQ` et `0x05` qui est le nombre d'octets à sauter pour le branchement conditionnel. En effet `BEQ` s'adresse en relatif, avec un operand en complément à deux sur 1 octet, permettant de faire de `pc` − 128 à `pc` + 127. Cette nouvelle ligne va transformer notre boucle comme ceci :

```
$0091: JSR $96E5 ; subroutine that waits until frame ends;
↪  NMI updates controllers & $15 counter
$0094: LDA $F7   ; load controller 1 state
$0096: BRK       ; do nothing
$0098: BEQ $009f ; if nothing pressed on controller 1, jump
↪  to $009f
$009a: INX       ; increment X register
$009b: STA,X $7d ; store controller data to next spot in
↪  RAM
$009d: BNE $0091 ; if X isn't zero, loop to $0091
```

`BEQ $009f` permet alors de sortir de notre boucle sitôt qu'aucun bouton n'est appuyé sur le contrôleur. Voilà une première contrainte concernant le contenu du code que nous allons injecter : il ne doit pas contenir d'octet nul sans quoi il signe la fin de l'écriture programme et donc son exécution.

On comprend donc qu'à ce stade, l'injection du point de vue du joueur est terminée. Nous avons injecté le code qui va nous permettre, via les entrées utilisateurs du contrôleur 1, d'écrire du code arbitraire ou du moins quasiment, car nous allons voir qu'il y a encore quelques problématiques.

3.6 Le second code injecté : la charge

L'enjeu est donc d'écrire un programme, de l'assembler et de traduire les octets résultants en des entrées de contrôleur de NES. Il est d'abord important de comprendre comment sont traduites les entrées utilisateur dans la mémoire de la console. Un pilote est chargé d'aller lire une zone mémoire (par exemple `0x4016` pour le contrôleur 1) selon le protocole établi pour comprendre sur quelles touches l'utilisateur appuie lors d'une *frame*. Le pilote de Super Mario Bros 3 a choisi de stocker la valeur résultante à l'adresse `0xF7`.

La manette de la NES possède 8 boutons (voir figure 7), ce qui est pratique pour coder l'information sur un octet. Et le codage de l'information est décrit par le tableau de la figure 6 [23]. Ce qui peut donner par exemple `0b10000001` si le joueur est en train de sauter en courant vers la droite dans Super Mario Bros 3.

bit	7	6	5	4	3	2	1	0
bouton	A	B	Select	Start	Up	Down	Left	Right

FIGURE 6 – Correspondance bouton-bit pour la manette de NES.

Il apparaît alors assez facile de se dire que l'on va pouvoir traduire n'importe quel programme en suite d'entrées utilisateur. Cependant, un détail majeur est à prendre en compte, le pilote de lecture du contrôleur de Super Mario Bros 3 empêche la possibilité d'appuyer en même temps sur haut-bas ou droite-gauche, ce qui est possible sur une manette abîmée [23], on peut le retrouver dans la version désassemblée du jeu [20]. Cela fait passer le sous-ensemble d'octets

FIGURE 7 – Manette de la NES avec ses 8 boutons.

possibles à écrire de 256 à 144. Il ne faut pas oublier non plus qu'il ne faudra pas écrire d'octet nul, sauf pour conclure le programme.

C'est pourquoi, pour cette partie, nous avons décidé d'écrire un script Python qui prend en entrée standard n'importe quel flot d'octets et les traduit en flot d'entrées contrôleur NES avec différents formats. Les formats possibles correspondent aux formats des listes d'entrées contrôleur dans les fichiers *movies* que nous utilisés via les différents émulateurs. Voici la documentation de script :

```
usage: bytesToNES.py [-h] [-d] [-f {bk2,fm2}] [-v] [-b BYTE]

Converts bytes to NES controller strings, see '-d' for
↪  information on the format.

optional arguments:
  -h, --help            show this help message and exit
  -d, --doc             display documentation on the NES
  ↪  controller string format and exit
  -f {bk2,fm2}, --format {bk2,fm2}
                        change the format of output
  -v, --verbose         verbose human readable output, line
  ↪  is red if the controller combinaison is impossible a
  ↪  priori
  -b BYTE, --byte BYTE  input one byte to convert
```

Voici par exemple un programme tout simple de NES, qui déclenche un son via l'*Audio Process Unit* (APU) sur une NES dans un état initialisé :

```
.org $0
lda #$01    ; square 1
sta $4015
lda #$08    ; period low
sta $4002
```

```
lda #$02    ; period high
sta $4003
lda #$bf    ; volume
sta $4000
forever:
  jmp forever
```

Une fois ce programme assemblé (nous avons utilisé l'assembleur de loopy `asm6f` [24] et `NESasm` [25]), on obtient la suite d'octet suivante : `a901 8d15 40a9 088d 0240 a902 8d03 40a9 bf8d 0040 4c14 00`. Le script `bytesToNES.py` permet de transformer cette suite d'octet en cette suite d'entrée utilisateur dans le format natif de la NES :

```
0xa9: A-s-U--R0x08: ----U---0x8d: A---UD-R0x00: --------
0x01: -------R0x8d: A---UD-R0x03: ------LR0x40: -B------
0x8d: A---UD-R0x02: ------L-0x40: -B------0x4c: -B--UD--
0x15: ---S-D-R0x40: -B------0xa9: A-s-U--R0x14: ---S-D--
0x40: -B------0xa9: A-s-U--R0xbf: A-sSUDLR0x00: --------
0xa9: A-s-U--R0x02: ------L-0x8d: A---UD-R
```

Et dans le format de *movie* utilisé par l'émulateur FCEUX, c'est-à-dire le format `fm2` :

```
0xa9: |0|R..U.s.A|........|0x8d: |0|R.DU...A|........||
0x01: |0|R.......|........|0x03: |0|RL......|........||
0x8d: |0|R.DU...A|........|0x40: |0|......B.|........||
0x15: |0|R.D.T...|........|0xa9: |0|R..U.s.A|........||
0x40: |0|......B.|........|0xbf: |0|RLDUTs.A|........||
0xa9: |0|R..U.s.A|........|0x8d: |0|R.DU...A|........||
0x08: |0|...U....|........|0x00: |0|........|........||
0x8d: |0|R.DU...A|........|0x40: |0|......B.|........||
0x02: |0|.L......|........|0x4c: |0|..DU..B.|........||
0x40: |0|......B.|........|0x14: |0|..D.T...|........||
0xa9: |0|R..U.s.A|........|0x00: |0|........|........||
0x02: |0|.L......|........||
```

Les lignes violettes sont les lignes qui ne sont pas « légales » avec un pilote qui effectue des vérifications sur la *directional safety* comme celui de Super Mario Bros 3. On constate donc qu'une première étape pour facilité l'écriture de code est de réécrire premièrement un pilote de contrôleur avec les instructions disponibles pour s'affranchir de la limite [12].

4 Conclusion

Ce projet nous a permis de découvrir une partie de la communauté organisée autour des *speedruns* et plus particulièrement des *tool-assisted speedruns*. Leurs exploits, aux deux sens du terme, sont impressionnants et nécessitent une connaissance du matériel et du fonctionnement du jeu très poussée. Les communautés développent des projets open sources sur des technologies vieilles de plusieurs dizaines d'années par plaisir et pour le partage de la passion de ces jeux qui ont accompagné une partie de leur vie.

D'un point de vue de la sécurité, on constate qu'on peut faire des parallèles entre les *exploits* réalisés dans ces jeux et ce qu'on peut retrouver dans le monde du logiciel. Cependant, il faut garder à l'esprit que le développement de ces jeux était d'abord contraint par les limitations du matériel et que les développeurs tentaient par tous les moyens de réaliser le meilleur jeu possible, tournant le mieux possible sur ces consoles en limitant le plus possible les bugs gênants pour le joueur.

Références

[1] Seth Bling. *SNES Code Injection – Flappy Bird in SMW*. 2016. URL : https://www.youtube.com/watch?v=hB6eY73sLV0 (cf. p. 1, 3).

[2] Kyle Orland. *How a robot got Super Mario 64 and Portal "running" on an SNES*. 2017. URL : https://arstechnica.com/gaming/2017/01/how-a-robot-got-super-mario-64-and-portal-running-on-an-snes/ (cf. p. 1, 3).

[3] Wikipedia. *TASBot*. 2018. URL : https://en.wikipedia.org/wiki/TASBot (cf. p. 1, 3).

[4] Julian N. - Masterjun. *Masterjun's SNES Super Mario World 'glitched' in 01 :39.74*. 2013. URL : http://tasvideos.org/3957S.html (cf. p. 2).

[5] Julian N. - Masterjun. *Masterjun's SNES Super Mario World 'Arbitrary Code Execution' in 02 :25.19*. 2014. URL : http://tasvideos.org/4156S.html (cf. p. 2).

[6] TASVideo.org. *Bizhawk / Lua Functions*. URL : http://tasvideos.org/Bizhawk/LuaFunctions.html (cf. p. 3).

[7] adelikat. *How To make a Tool Assisted Speedrun*. 2012. URL : http://tasvideos.org/TASTutorial.html (cf. p. 3).

[8] Sam Agnew. *How to Write Lua Scripts for Video Games with the BizHawk Emulator*. 2020. URL : https://www.twilio.com/blog/how-to-write-lua-scripts-for-video-games-with-the-bizhawk-emulator (cf. p. 3).

[9] Retro Game Mechanics Explained. *Super Mario Land 2 - Memory Exploration*. 2018. URL : https://www.youtube.com/watch?v=FPzuYWbnln4 (cf. p. 3).

[10] Retro Game Mechanics Explained. *Super Mario Bros. 3 - Wrong Warp*. 2016. URL : https://www.youtube.com/watch?v=fxZuzos7Auk (cf. p. 3).

[11] Lord Tom et Tompa. *Lord Tom and Tompa's NES Super Mario Bros. 3 in 02 :54.98*. 2014. URL : https://videos.org/4288S.html (cf. p. 3).

[12] Lord Tom. *Lord Tom's NES Super Mario Bros. 3 'Total Control' in 08 :16.23*. 2016. URL : http://tasvideos.org/4961S.html (cf. p. 4, 7, 11, 15).

[13] Brad Taylor. *NTSC 2C02 technical reference*. 2004. URL : http://nesdev.com/2C02%5C%20technical%5C%20reference.TXT (cf. p. 6).

[14] Patrick Diskin. *Nintendo Entertainment System Documentation*. 2004. URL : http://www.nesdev.com/NESDoc.pdf (cf. p. 6).

[15] Norbert Landsteiner. *6502 Instruction Set*. 2015. URL : https://www.masswerk.at/6502/6502_instruction_set.html (cf. p. 6).

[16] Wikibooks. *NES Programming*. URL : https://en.wikibooks.org/wiki/NES_Programming (cf. p. 6).

[17] Datacrystal. *Super Mario Bros. 3 :RAM map*. 2018. URL : https://datacrystal.romhacking.net/wiki/Super_Mario_Bros._3:RAM_map (cf. p. 6, 8).

[18] TASVideo.org. *BK2 Format*. URL : http://tasvideos.org/Bizhawk/BK2Format.html (cf. p. 8).

[19] adelikat. *FM2 Movie file format*. 2009. URL : http://fceux.com/web/FM2.html (cf. p. 8).

[20] captainsouthbird. *Disassembly of Super Mario Bros 3*. 2017. URL : https://github.com/captainsouthbird/smb3 (cf. p. 8, 13).

[21] mariowiki. *List of Super Mario Bros. 3 glitches*. URL : https://www.mariowiki.com/List_of_Super_Mario_Bros._3_glitches (cf. p. 9).

[22] Summoning Salt. *World Record Progression : Super Mario Bros 3 any%*. 2017. URL : https://www.youtube.com/watch?v=sllAIF99h3s (cf. p. 11).

[23] NESDev.com. *Controller Reading Code*. 2021. URL : https://wiki.nesdev.com/w/index.php/Controller_reading_code (cf. p. 13).

[24] loopy. *asm6f 6502 assembler*. 2018. URL : https://github.com/freem/asm6f (cf. p. 15).

[25] *NESasm 6502 assembler*. 2016. URL : https://github.com/camsaul/nesasm (cf. p. 15).