Obfuscation de code Python : amélioration des techniques existantes

Serge Guelton et Ninon Eyrolles
sguelton@quarkslab.com
neyrolles@quarkslab.com

Quarkslab

Résumé Le langage Python a connu un essor certain ces dix dernières années. Pour faciliter le déploiement ou rendre plus difficile l'accès au code source, des « packeurs » d'application ont vu le jour. Ces outils ne se contentent pas d'archiver le code source de l'application accompagné d'un interpréteur Python dans un unique fichier, ils appliquent aussi plusieurs transformations au code et/ou à l'interpréteur pour rendre plus difficile le travail de rétro-ingénierie.

Cet article porte sur l'étude des mesures d'obfuscation existantes dans le contexte du langage Python et les contre-mesures qui ont été adoptées. Il propose également plusieurs techniques d'obfuscation reposant sur la modification conjointe du code Python et de l'interpréteur, rendant les stratégies de décompilation reposant sur un seul des aspects inefficaces.

Introduction

Le client du logiciel DropBox est écrit en Python et a été obfusqué en utilisant plusieurs techniques d'obfuscation. [4] dresse un éventail assez précis des techniques mises en œuvre :

1. compilation du code client en un binaire embarquant une version modifiée de l'interpréteur lié en statique et le *bytecode* client sous forme de tableau d'octets ;
2. modification du `MAGIC_NUMBER`, un champ de 4 octets présent dans l'entête de tout fichier Python pré-compilé déterminant la version de l'interpréteur Python nécessaire pour interpréter le *bytecode* ;
3. chiffrement du *bytecode*, qui est déchiffré à la volée par l'interpréteur modifié ;
4. suppression de certaines fonctionnalités de l'interpréteur, comme la possibilité d'accéder au champ d'une fonction contenant son *bytecode* ou de sérialiser du *bytecode* ;
5. modification de la table des *opcode*s et de leur correspondance dans l'interpréteur embarqué avec l'application.

Les techniques présentées dans [4] permettent de passer outre ces protections. Le point 1 n'empêche pas d'accéder au *bytecode* à partir du moment où l'on a pu localiser, par analyse du binaire de l'interpréteur, l'adresse de la fonction chargée de décoder le *bytecode*. Le point 2 ne requiert qu'une modification de l'en-tête. Au lieu de rechercher la clef de chiffrement embarquée dans l'application et utilisée pour chiffrer le *bytecode*, les auteurs ont injecté (via `LD_PRELOAD`) un appel à des fonctions de la bibliothèque Python standard pour accéder au *bytecode* déchiffré. La modification des *opcode*s a été cassée en tirant parti des résultats de `dropboxdec` [1] qui réutilise la technique de comparaison de binaire présentée dans [5].

Il apparaît donc que trois facteurs ont rendu possible l'accès au code final de DropBox :

1. la relative facilité avec laquelle les auteurs ont pu exécuter du code dans le processus `dropbox` et la disponibilité de certaines fonctions de l'API C Python [7] comme `PyRun_SimpleString` qui permet d'exécuter n'importe quelle chaîne de caractères contenant du code Python ;
2. la présence d'un invariant (les modules standards Python) utilisé pour reconstruire la table des *opcodes* ;
3. la possibilité de retrouver le code Python d'origine à partir du *bytecode* en utilisant un décompilateur comme `uncompyle2` [2].

Tout en implémentant les techniques déjà utilisées, l'objectif de cet article est d'ajouter plusieurs niveaux d'obfuscation supplémentaires applicables automatiquement pour ralentir l'attaquant. L'article est structuré de la façon suivante : la section 1 présente des modifications de l'interpréteur Python qui rendent plus difficile l'exécution de code utile par l'attaquant. La section 2 explore les possibilités de modification des modules standards Python afin de supprimer l'invariant qui a permis de retrouver les *opcodes* d'origine. La section 3 présente plusieurs techniques propres au langage Python qui permettent de rendre difficile l'obtention du code source à partir du code compilé, puis dans le cas où il est obtenu, de rendre sa compréhension plus ardue.

1 Restriction des fonctions disponibles pour l'attaquant

Le point de départ de l'attaque de DropBox est la possibilité d'appeler depuis le processus attaqué les fonctions de l'interpréteur qui permettent de

1. `https://github.com/rumpeltux/dropboxdec`

passer plusieurs mécanismes, comme le chiffrement du *bytecode*. Il est donc naturel de s'intéresser aux moyens pour rendre cette étape plus difficile.

1.1 Suppression du chargement dynamique

Les *packers* Python existants comme `cx_freeze` reposent sur une fonctionnalité disponible en Python qui permet de « geler » un module, ce qui consiste d'une part en le stockage du *bytecode* de ce module dans un tableau d'octets chargé au démarrage de l'interpréteur, et d'autre part en une plus grande priorité dans le mécanisme d'importation de module.

Le *packing* d'une application consiste grossièrement en l'agglomération dans un binaire de l'interpréteur Python à travers la `libpython`, de tous les modules Python requis, en les déclarant gelés, et de tous les modules natifs requis. Comme Python charge les modules natifs dynamiquement, le binaire généré est vulnérable à des attaques basées sur `LD_PRELOAD` qui permettent ensuite à l'attaquant d'exécuter n'importe quel code Python dans le processus.

De nombreux modules standards Python sont des modules natifs et sont donc chargés dynamiquement, ce qui rend a priori difficile la suppression de tous les chargements dynamiques. Cependant, en prenant la main sur la chaîne de compilation utilisée pour construire l'interpréteur, il est possible de compiler statiquement la plupart de ces modules qui sont alors intégrés à la `libpython`. On supprime ainsi toute liaison dynamique. On notera cependant que certaines bibliothèques, notamment la `libpthread`, supportent mal la liaison statique et qu'il est alors nécessaire de modifier de façon significative le schéma de construction de la `libpython` pour supprimer toute liaison dynamique.

Cette approche s'étend plus difficilement aux modules natifs tierces, puisqu'il faut avoir accès à leur code source pour les recompiler en statique et les intégrer à l'interpréteur. Le code de nombreux modules populaires est cependant disponible, ce qui rend la tâche théoriquement possible.

1.2 Suppression de fonctionnalités de l'interpréteur

Plusieurs aspects du langage Python peuvent faciliter la tâche d'un attaquant. On peut par exemple citer l'introspection qui permet d'accéder au *bytecode* d'une fonction de manière dynamique (attribut `co_code`), l'accès au module `marshal` pour (dé)sérialiser tout code objet (fonctions `read_object` et `write_object`), évaluation de code sous forme de chaîne de caractères dans un environnement de son choix (fonction `PyRun_SimpleString`).

Ces fonctions font partie intégrante de l'API C de Python, elles sont utilisées par l'interpréteur pour implémenter certains modules. On ne peut donc pas les supprimer inconditionnellement. Comme l'interpréteur est embarqué dans le *packer*, on peut par contre forcer l'*inlining* de ces fonctions pour supprimer le symbole, voire essayer, par analyse statique, de déterminer si une fonction est utilisée. Par exemple la fonction `read_object`, utilisée pour désérialiser un code objet, est tout le temps utilisée, ne serait-ce qu'au chargement d'un module. En revanche, son pendant `write_object` n'est pas forcément utile. De même le champ `co_code` n'est utilisé que par certains modules qui font de l'introspection sur le *bytecode*, comme `module_finder` ou `dis`, modules qu'il n'est pas forcément souhaitable d'embarquer.

2 Obfuscation des modules Python standards

Pour empêcher l'attaquant de tirer parti du *bytecode* une fois qu'il y a accès, l'obfuscateur utilisé dans DropBox modifie la table des *opcodes* en effectuant une permutation des valeurs d'origine. Pour retrouver la table initiale, l'outil [5] part de l'hypothèse que les modules de la bibliothèque standard n'ont pas été modifiés pour apparier l'ancien *bytecode* et le nouveau. Cette stratégie est caduque si le code source des modules standards a été modifié avant d'être embarqué.

Modifier le code des modules de la bibliothèque standard demande un soin particulier. En effet, Python utilise une **liaison tardive**, ce qui rend impossible d'associer statiquement une valeur à chaque identifiant. Il est néanmoins possible de le transformer de manière **aléatoire**, en modifiant le flot de contrôle — il est indépendant des identifiants — ou en modifiant les opérations sur les données, mais uniquement quand on a pu trouver, localement, l'ensemble des valeurs possibles pour un identifiant en ce point.

2.1 Transformation de l'arbre de syntaxe abstraite

La figure 1 illustre le comportement de la chaîne de compilation source-à-source mise en œuvre. Pour appliquer des passes d'obfuscation sur du code source Python, on utilise un arbre de syntaxe abstraite, ou *Abstract Syntax Tree* (AST). Cette structure est adaptée pour appliquer des transformations source-à-source puisqu'il est facile de regénérer du code Python à partir d'un AST donné. Le fichier source de chaque module est transformé en un AST par la fonction `parse` disponible dans le module standard `ast`. Cet arbre est ensuite transformé en un arbre équivalent, puis un *pretty printer*

génère le code source Python correspondant. Il est également possible de chaîner les transformations de cette manière.

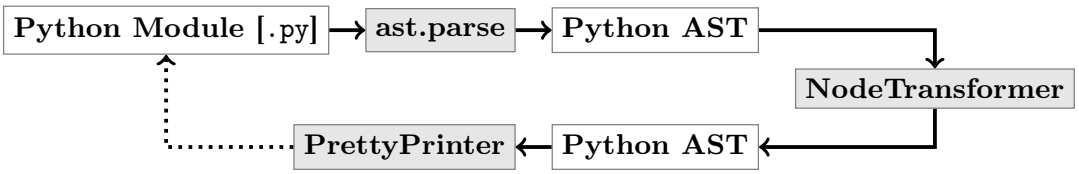

Figure 1. Flot de compilation source-à-source pour la transformation de module Python.

Les transformations sont effectuées au niveau de l'AST en modifiant les nœuds existants ou en les remplaçant par de nouveaux nœuds. Il est ainsi possible d'effectuer automatiquement diverses modifications sur du code source Python tout en préservant la sémantique du programme original, comme par exemple dans [1]. Nous avons effectué trois types de modifications de l'AST : des modifications du flot de contrôle (boucles, branchements...), du flot de données (modification des constantes, transformation des expressions arithmétiques...) ainsi que des identifiants (renommage des fonctions, des variables...)

2.2 Modification du flot de contrôle

Les modifications du flot de contrôle consistent à changer les chemins pris lors de l'exécution du programme. Visuellement, on cherche à transformer le graphe de flot de contrôle (*Control Flow Graph*) du programme. Pour cela, il existe de nombreuses solutions possibles, allant de simples modifications sur les boucles ou les conditions, à des techniques plus complexes comme l'expansion de fonctions (*inlining*) ou l'aplatissement de code (*code flattening*) [6].

Dans le contexte de nos travaux, des modifications simples peuvent être suffisantes puisqu'il s'agit de modifier les modules standards Python afin d'empêcher la correspondance automatique des *opcodes*. On privilégie alors des transformations sur un seul nœud de l'AST.

Transformation des boucles `for` Une obfuscation classique des boucles `for` empruntée aux techniques d'optimisation est le déroulage de boucle. En effet, cette transformation permet de cacher un motif important et d'augmenter facilement la taille du code obfusqué. Néanmoins, cette obfuscation nécessite certains pré-requis pour être appliquée. En effet, il est

courant que l'itération se fasse sur le résultat d'un appel à la fonction `range` du module `__builtin__` ; or l'identifiant `range` n'est pas forcément associé à cette fonction : il peut avoir reçu une autre valeur. À défaut d'une analyse de *points-to* précise, l'obfuscateur a besoin d'une indication du développeur pour s'assurer que le déroulage peut se faire suivant le résultat standard de la fonction `range`. Comme pour d'autres techniques d'obfuscation abordées dans la suite de cet article, on utilise un décorateur de fonctions pour indiquer que les identifiants des fonctions de `__builtin__` n'ont pas été réaffectés.

Dans le cas où l'on itère sur une collection figée, e.g. `[1, 2, 1]`, on peut réaliser systématiquement le déroulage de boucle, en prenant néanmoins garde à évaluer le contenu du tableau dans le même contexte que le code d'origine.

Si le déroulage n'est pas possible, les boucles `for` sont transformées en boucle `while` suivant le modèle de la figure 2. Comme `iter` peut avoir été redéfini, on importe la fonction `iter` du module `__builtin__` en utilisant un identifiant unique que nous avons noté ici `my_iter`. Cela suppose néanmoins que la valeur pointée par l'identifiant `__builtin__` n'ait pas été modifiée.

```
# original loop
for expr0 in expr1:
    work()
```

⟶

```
from __builtin__ import
    iter as my_iter

# obfuscated loop
var = my_iter(expr1)
while 1:
    try:
        expr0 = var.next()
    except StopIteration:
        break
    work()
```

Figure 2. Modèle de transformation d'une boucle `for` en boucle `while`.

Transformation des instructions `if` et `while` Cette obfuscation vise à rendre la distinction des branchements conditionnels et des boucles `while` plus complexe. Pour cela, on transforme chaque instruction `if` en `while` en utilisant un prédicat opaque, suivant le modèle de la figure 3. Il suffit ensuite de transformer les instructions `while` sur le même modèle, mais avec un prédicat opaque toujours vrai. Pour créer les prédicats opaques, on utilise les expressions *Mixed Boolean-Arithmetic* présentées section 2.3. Cela nous permet de transformer une simple affectation en une équation beaucoup plus difficile à comprendre ou à simplifier. Il est également possible d'utiliser les prédicats opaques de manière plus classique, c'est-à-

dire en ajoutant des branchements conditionnels dont une branche n'est jamais prise : on place alors dans celle-ci du code dit « mort ». Le plus simple paraît de copier aléatoirement une partie du code existant et de l'obfusquer pour qu'elle soit plus difficilement reconnaissable.

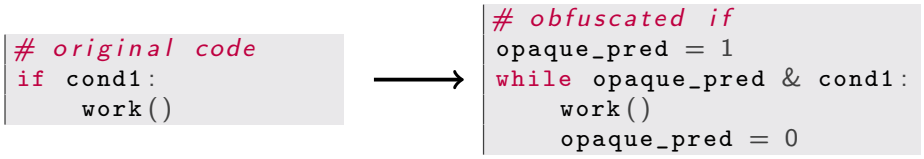

Figure 3. Modèle d'obfuscation d'un branchement conditionnel.

Obfuscation des listes en compréhension Les listes en compréhension (*list comprehension*) permettent la création d'une liste de manière simple, en utilisant des boucles `for` et des instructions `if` dans la déclaration de la liste. Pour obfusquer ces éléments, on crée une nouvelle fonction pour chaque boucle `for` qui calcule la liste nécessaire. Cette technique est donc fortement inspirée de l'*outlining* de fonction, c'est-à-dire du fait de remplacer une portion de code par un appel à une fonction qui contient cette partie du code. Le modèle de cette obfuscation est détaillé dans la figure 4. Ce modèle n'est pas exhaustif : en effet, dans le cadre de listes en compréhension imbriquées, il peut être nécessaire de passer certaines variables en arguement des fonctions créées (voir la figure 5 pour un exemple).

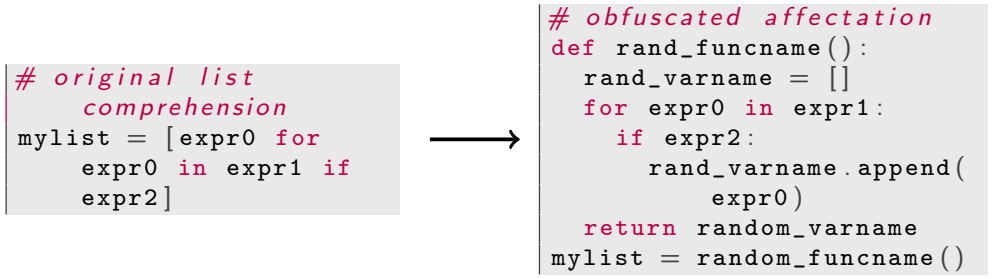

Figure 4. Modèle d'obfuscation d'une liste en compréhension.

De manière générale, on peut imaginer toute une série de transformations qui remplacent une structure de contrôle ou une certaine suite d'instructions par une autre moins lisible pour l'utilisateur. D'autres techniques plus poussées peuvent être également utilisées pour renforcer l'obfuscation, mais nous avons pris le parti de rester sur des transformations relativement simples car elles s'adaptaient à nos besoins. Si les transformations

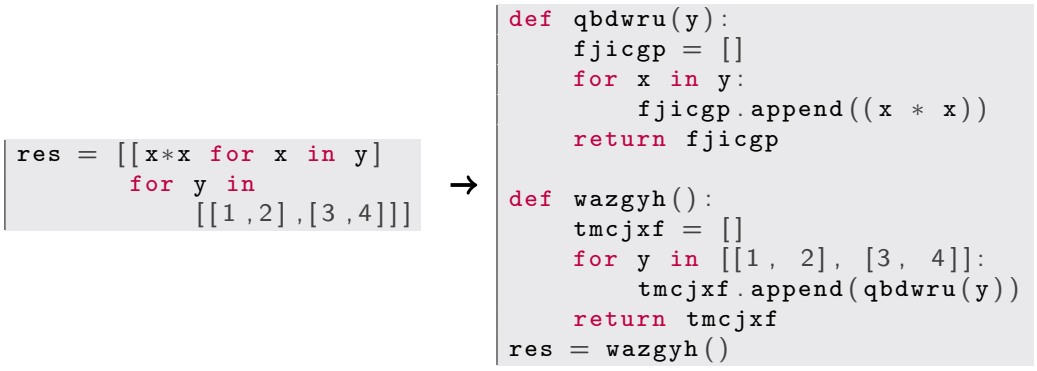

Figure 5. Exemple d'obfuscation de listes en compréhension imbriquées.

présentées dans cette section ne sont pas particulièrement complexes, elles ont l'avantage de modifier significativement le *bytecode*, et donc de rendre plus difficile la comparaison de module destinée à retrouver la permutation d'*opcodes*.

Les transformations présentées ci-dessus ne sont pas triviales, principalement à cause de certaines particularités de Python. En effet, si l'on considère la transformation des boucles `for` ou des *list comprehension*, on retrouve le même problème lié à la liaison dynamique. Comme vu précédemment, l'utilisation de la fonction `range` pose certains problèmes, mais également d'autres fonctions comme `iter`. Le côté dynamique de Python pose des limites aux obfuscations, limites que l'on ne rencontre pas dans d'autres langages.

2.3 Modification du flot de données

Les modifications du flot de données (*data flow*) visent à masquer les données manipulées par le programme, par exemple en appliquant des transformations ou en insérant un grand nombre de données inutiles (*junk code*). Une première obfuscation simple est de ne pas laisser les littéraux visibles clairement par l'utilisateur. Comme le *packer* a le contrôle de l'interpréteur, on peut s'autoriser à chiffrer les chaînes de caractères dans le code source, et à modifier l'interpréteur de façon à ce qu'il déchiffre ces constantes à la volée.

Une autre obfuscation est la transformation des expressions arithmétiques en expressions arithmétiques-booléennes (*Mixed Boolean-Arithmetic equations*), telle que décrite par [9]. En effet, les méthodes de simplification actuelles sont peu efficaces lorsqu'une expression mélange arithmétique classique et arithmétique booléenne. Une table d'expressions équivalentes

est disponible dans [8] et illustrée par les équations 1, 2 et 3. Des travaux en cours étudient la possibilité de substituer un générateur à cette table.

$$(x + y) = (x \oplus y) + 2 \times (x \wedge y) \tag{1}$$

$$(x \vee y) = (x \wedge (\neg y)) + y \tag{2}$$

$$(x \oplus y) = (x \vee y) - (x \wedge y) \tag{3}$$

Cette technique peut également être utilisée pour obfusquer les affectations. Par exemple, pour obfusquer `x = 36`, on choisit un nombre aléatoire r, et on introduit $d = 36 - r$ si $r < 36$ (resp. $d = r - 36$ sinon). Il suffit alors d'appliquer la transformation MBA (éventuellement de manière récursive) sur $d + r = 36 - r + r = 36$ (resp. $r - d = r - r + 36 = 36$). On peut voir un exemple d'une affectation obfusquée dans la figure 6

```
x = 36  →  x = ((((2 * ((-816744550) | 816744552)) -
                 ((-816744550) ^ 816744552)) *
                (((3783141896 ^ 3921565134) -
                  (2 * ((~3783141896) & 3921565134))) |
                 ((4009184523 & (~3870761249)) -
                  ((~4009184523) & 3870761249)))) -
               (((2105675179 & (~2244098417)) -
                 ((~2105675179) & 2244098417)) ^
                ((3657555079 + (~3519131805)) + 1)))
```

Figure 6. Modèle d'obfuscation d'une affectation.

Une amélioration intéressante de ce type d'obfuscation est d'étendre le calcul de l'équation à plusieurs lignes de codes (qui ne sont pas forcément adjacentes) en utilisant des variables intermédiaires.

2.4 Obfuscation des symboles

Une technique assez simple mais efficace est de remplacer les noms de fonctions ou de variables par des chaînes de caractères aléatoires. En effet, ces noms donnent souvent des informations précieuses sur l'utilité d'une variable ou sur la finalité d'une fonction. Il paraît donc important de remplacer ces informations par des chaînes de caractères aléatoires. Or, comme vu précédemment, cela n'est pas simple à mettre en place en Python : renommer les fonctions peut modifier fortement le comportement d'un programme. Par exemple, changer le nom de la fonction `__init__` d'une classe modifiera la sémantique du code. Concernant les variables, on peut trouver également des exemples de code (voir listing 2.4) qui

empêchent un renommage automatique des variables en utilisant l'AST. Il existe de nombreux cas de ce genre en Python, il est donc nécessaire de bénéficier d'une information de la part de l'utilisateur : on utilisera donc un décorateur pour indiquer si le nom d'une fonction ou de ses variables peuvent être obfusqués.

```
exec('a=1')
print a
```

Figure 7. Exemple de code résistant à la transformation de symboles.

2.5 Transformation en programmation fonctionnelle

Bien que le langage Python soit de haut niveau, la pratique [3] montre par exemple qu'il est tout à fait possible d'utiliser un style de programmation fonctionnel en abusant des fonctions d'ordre supérieur. En s'inspirant de la sémantique dénotationnelle, un modèle mathématique utilisé pour la formalisation et la preuve de programme, nous avons implémenté une technique d'obfuscation qui transforme une définition de fonction en une affectation de lambda fonction. Cette lambda fonction utilise elle-même la composition d'autres lambda fonctions pour modéliser le flot de contrôle. Cette transformation est succinctement illustrée dans la figure 8. Une information critique pour la compréhension du code est de savoir que le paramètre formel `_` représente l'état mémoire local courant, la clef `'$'` stocke la valeur de retour de la fonction, et l'état mémoire local est initialisé à `{'$': None}`

```
def foo():
    while 1:
        pass
```

→

```
foo =
 (lambda :
    (lambda _:
      (lambda f, _: f(f, _))
         ((lambda __, _: ((lambda _: __(
             __, _))((lambda _: _)(_))
             if 1 else _)),
          _)
    )({'$': None})['$'])
```

Figure 8. Exemple d'obfuscation en utilisant un style fonctionnel.

3 Prévention de la décompilation

Si l'attaquant parvient à accéder au *bytecode* en clair, il lui suffit généralement d'utiliser un décompilateur comme `uncompyle2` [2] ou `pycdc`.

Ces outils supposent que le *bytecode* qui leur est fourni a été généré par l'interpréteur standard et va s'exécuter sur ce même interpréteur. On peut tirer parti de ces suppositions pour induire ces outils en erreur. Les modifications décrites section 2 peuvent également servir à rendre plus difficile la compréhension du source une fois le *bytecode* décompilé.

3.1 Insertion d'opcode inutiles

Le *bytecode* Python est interprété en utilisant un mécanisme de pile. Parmi les *opcodes* disponibles, on note la présence d'instructions de manipulation de la pile, e.g. `ROT_TWO`, `ROT_THREE` ou `POP_TOP` qui effectuent respectivement une permutation circulaire des deux éléments du dessus de la pile, des trois éléments du dessus de la pile ou qui supprime l'élément en haut de la pile. Combinées avec une instruction ajoutant une valeur au dessus de la pile, elles permettent de modifier le *bytecode* sans modifier les valeurs calculées. La figure 9 illustre une telle séquence pour un opérateur binaire quelconque.

L'intérêt d'une telle transformation réside dans le fait que l'interpréteur standard n'aurait jamais généré une telle séquence. Par exemple l'outil `uncompyle2` termine sur une erreur à la lecture du *bytecode* de la figure 9.

`LOAD_FAST 0`	`LOAD_FAST 1`	`BUILD_MAP`	`ROT_THREE`	`BINARY_ADD`	`ROT_TWO`	`POP_TOP`

Figure 9. Insertion d'*opcodes* inutiles autour d'un opérateur binaire. Les *opcodes* grisés ne participent pas au calcul final.

3.2 Écriture de code auto-modifiant

Une autre opportunité d'obfuscation qui complexifie la tâche du désassembleur est l'écriture de code auto-modifiant. On part du fait que toute fonction Python embarque son propre *bytecode* dans le champ `func_code.co_code`. Cet attribut est en lecture seule au niveau Python, mais c'est une simple chaîne de caractères au niveau C. Il est donc possible de la modifier en écrivant une fonction native exposée au niveau Python. Par exemple, la fonction présentée à la figure 1 appelée juste avant un opérateur binaire transforme cet opérateur en somme. L'obfuscation consiste donc en :

1. transformation du code source pour remplacer un opérateur `+` en un autre opérateur ;
2. ajout d'un appel à la fonction modifiant le *bytecode* avant l'opérateur.

Pour comprendre le comportement de la séquence, on ne peut plus se fier à une seule inspection *a priori* du *bytecode*, il est nécessaire également d'analyser la fonction native appelée. Il est aisé de générer un grand nombre de tels modules binaires, chacun spécialisé pour une situation donnée, ce qui complexifie grandement la tâche de l'attaquant.

```
PyThreadState *tstate = PyThreadState_GET();
if (tstate && tstate->frame) {
    PyFrameObject *frame = tstate->frame;
    char *bytecode = PyString_AsString(frame->f_code->co_code);
    bytecode[frame->f_lasti+10] = BINARY_ADD;
}
```

Listing 1. Fragment de code permettant la conversion d'un opérateur binaire quelconque en addition si appelé avant l'opérateur.

Conclusion

Cet article présente plusieurs techniques permettant d'écrire un *packer* obfusquant pour le langage Python. Plusieurs techniques sont spécifiques au Python, alors que d'autres s'appliquent de manière similaire à d'autres langages interprétés comme Javascript ou Ruby. Une idée maîtresse est de répartir le « secret » entre le *bytecode* de l'application et l'interpréteur spécialisé pour cette application. On peut alors appliquer des techniques d'obfuscation propres à chaque cible après avoir séparé le secret entre elles.

Références

1. Jurriaan Bremer. Python source obfuscation using asts. `http://jbremer.org/python-source-obfuscation-using-asts/`, 2013.
2. Hartmut Goebel. uncompyle2, a Python 2.7 byte-code decompiler. `https://github.com/wibiti/uncompyle2`, 2012.
3. Johnny Healey. How to write obfuscated Python. PyConUS, 2011.
4. Dhiru Kholia and Przemysław Węgrzyn. Looking inside the (drop) box. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, pages 9–9, Berkeley, CA, USA, 2013. USENIX Association.
5. Rich Smith. pyREtic, in memory reverse engineering for obfuscated Python bytecode. In *BlackHat / Defcon security conferences*, 2010.
6. Geoffroy Gueguen Sébastien Josse. Aplatissement de code. In *MISC n°68*, 2013.
7. Guido van Rossum and Fred L. Jr. Drake, editors. *Python/C API Reference Manual*. Python Software Foundation, September 2012.
8. Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
9. Yongxin Zhou, Alec Main, YuanX. Gu, and Harold Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *Information Security Applications*, volume 4867 of *Lecture Notes in Computer Science*, pages 61–75. Springer Berlin Heidelberg, 2007.