Mise à plat de graphes de flot de contrôle et exécution symbolique

Eloi Vanderbeken
eloi.vanderbeken@oppida.fr

Oppida

Résumé Dans cet article nous présentons une approche permettant le débrouillage d'un graphe de flot de contrôle aplati, ainsi qu'une implémentation de cette méthode. Les résultats que nous avons obtenus dans le cadre de l'étude du mécanisme de signature RSA boite blanche mise en place par Apple dans iOS ; son système d'exploitation mobile.

1 Introduction

La technique de mise à plat du graphe de flot de contrôle [1] (ou CFG flattening) dans le contexte du brouillage de code a été décrite par Wang *et al.* en 2000 [6]. Elle consiste à brouiller le graphe de flot de contrôle (ou Control Flow Graph, noté CFG dans la suite de l'article) en remplaçant les différentes instructions de branches liant les blocs de bases par un unique gestionnaire. Ce gestionnaire est chargé de rediriger l'exécution du code à l'aide de variables identifiant le prochain bloc de base à exécuter, initialisées par les blocs de base précédemment exécutés. Le CFG ne donne alors plus d'information sur l'ordre d'exécution des blocs comme illustré dans la figure 1 or de nombreux algorithmes de débrouillage de code nécessitent cette information.

Sous certaines conditions, nous proposons une approche permettant la récupération du CFG d'une fonction brouillée à l'aide de la technique de mise à plat du CFG. L'article est organisé comme suit. Dans la section 2 nous détaillerons les approches existantes ou envisageables ainsi que leurs limitations. Dans la section 3 nous introduirons les notions de *BasicBlock* et de *Trace* puis nous montrerons comment utiliser ces notions pour reconstruire le CFG original d'une fonction. Dans la section 4 nous décrirons les différents choix d'implémentation qui ont été faits ainsi que les résultats obtenus sur le cas concret de la protection mise en place par Apple dans son système de gestion de droit FairPlay. Enfin dans la

1. Pour le plaisir des yeux du lecteur, l'auteur a fait le choix, lorsque cela était possible, de n'utiliser que des termes validés par la Commission générale de terminologie et de néologie

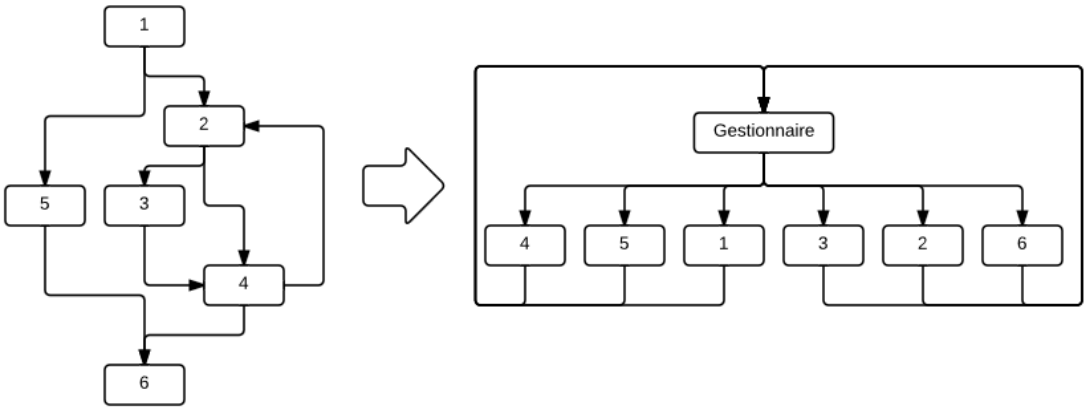

Figure 1. CFG avant et après l'opération de mise à plat

section 5, nous discuterons des évolutions possibles des méthodes de récupération et de brouillage.

2 Approches possibles

Mis à part un chapitre de thèse [4], il n'existe pas à la connaissance de l'auteur de description dans la littérature d'une méthode spécifique à la reconstruction d'un CFG mis à plat. La solution la plus simple, et celle qui est généralement retenue face à ce type de brouillage, est de ne pas tenter de reconstruire le CFG original. En effet, une analyse dynamique du code donne directement l'ordre d'exécution des instructions. Il est alors possible de retrouver la logique du programme en utilisant des méthodes génériques comme le suivi du flot de données (enregistrement des modifications et affectations de variables [2]) ou le suivi du flot d'exécution.

Si ces approches conviennent pour l'analyse de fonctions simples ou ayant certaines propriétés (comme une quantité de branches conditionnelles ou de dépendances inter-variables peu élevée, par exemple dans le cas d'un algorithme de chiffrement symétrique) elles deviennent inefficaces face à des algorithmes complexes. En effet, la masse de données et l'absence d'abstraction due au manque d'information sur la structure de la fonction rendent leur exploitation complexe.

La reconstruction du CFG à partir d'une trace d'exécution n'est pas non plus triviale, d'une part parce que rien n'assure qu'une ou plusieurs exécutions données aient couvert l'ensemble du CFG et d'autre part parce que les conditions de branche ne sont pas directement accessibles (du fait du remplacement des branches conditionnelles par des affectations

conditionnelles, de l'utilisation de prédicats opaques, du brouillage du code etc.).

Les méthodes de rétro-ingénierie classiques sont bien sûr applicables :

- analyse manuelle du code ;
- identification de motifs ;
- automatisation de la reconstruction à l'aide des motifs ;
- itération jusqu'à l'obtention du CFG.

Néanmoins, cette approche peut s'avérer complexe et l'analyse d'une implémentation donnée de mise à plat de graphe de flot de contrôle ne pourra être directement réutilisée pour une autre implémentation.

2.1 Exécution symbolique

L'exécution symbolique permet de surapproximer l'ensemble des traces du programme en manipulant des valeurs symboliques plutôt que concrètes comme illustré dans la figure 2.

Label	Code	Contexte après exécution de l'instruction	Prédicats
$l1$	`int foo(int arg){`	$PC = l2, arg = SymArg$	$\emptyset$
$l2$	`  if (arg)`	$PC = If(Arg, l3, l4), arg = Arg$	$\emptyset$
$l3$	`  return 1;`	$PC = retAddr, arg = Arg$	$If(Arg, l3, l4) = l3 \Leftrightarrow Arg \neq 0$
$l4$	`  return 2;`	$PC = retAddr, arg = Arg$	$If(Arg, l3, l4) = l4 \Leftrightarrow Arg = 0$
$l5$	`}`		

Figure 2. Illustration de l'utilisation de l'exécution symbolique pour la détermination des destinations et conditions de branches

L'exécution symbolique ne permet cependant pas à elle seule de reconstruire le CFG original d'une fonction. En effet, si les destinations et conditions de branches sont retrouvées, il reste nécessaire de différencier le code chargé de la redirection du flot d'exécution du code *utile*. De plus la complexité de l'exécution symbolique explose rapidement en fonction du nombre de branches ; une méthode d'abstraction devient nécessaire pour pouvoir étudier l'ensemble des exécutions possibles de la fonction à analyser.

3 BasicBlock et Traces

Pour reconstruire le CFG d'une fonction, celle-ci va être exécutée de manière symbolique à partir d'un contexte initial fixé. Quand un

gestionnaire est rencontré, on fait une sur-approximation la plus petite possible des blocs pouvant être atteints. Si un bloc est atteignable et a déjà été exécuté, il est réexécuté en fusionnant les différents contextes d'exécution initiaux. Le déroulement de cet algorithme est illustré dans la figure 3.

Pour effectuer cette reconstruction, nous introduisons les notions de *Trace* et de *BasicBlock*, inspirées de Pin [5]. Dans la suite de cette section nous définissons plus formellement l'algorithme de reconstruction du CFG ainsi que les différents objets et fonctions utilisés.

> Un contexte $ctx \in CTX$ est un dictionnaire de variables. Un contexte contient toujours la variable spéciale *PC* correspondant au pointeur d'instruction. La valeur d'une variable *var* contenue dans un contexte *ctx* est notée $ctx[var]$. Cette valeur est soit concrète (c'est à dire qu'elle a une valeur numérique) $ctx[var] \in CCT$, soit symbolique $ctx[var] \in SYM$. Une valeur symbolique étant une expression de symboles et de valeurs concrètes. On note VAL l'union de l'ensemble des valeurs symboliques et concrètes.

En pratique, un contexte contiendra les valeurs des registres ($ctx[R0]$, $ctx[SP]$ etc.), des drapeaux ($ctx[ZF]$, $ctx[NF]$ etc.) et de la mémoire ($ctx[mem(0x402000)]$) à un instant donné de l'exécution. Les termes inconnus des valeurs symboliques correspondent aux arguments inconnus ou aux variables non initialisées d'une fonction par exemple.

Au début d'une fonction les variables sont fixées, l'exécution symbolique permet d'exprimer l'état du programme, son contexte, en fonction de ces variables et ce à chaque point de la fonction.

> On note $sym_{(var,adr)}$ le symbole représentant la valeur de la variable *var* à l'adresse *adr*.

Par exemple, la valeur de `R0` au label `l_label` est notée $sym_{(R0,l\_label)}$.

> Si deux contextes ctx_1 et ctx_2 sont tels que $ctx_1[PC] = ctx_2[PC] = adr \in CCT$, leur union — notée $ctx_u = ctx_1 \cup ctx_2$ — est donnée par la formule suivante :
> $$ctx_u[var] = \begin{cases} ctx_1[var] & \text{si } ctx_1[var] = ctx_2[var] \\ sym_{(var,adr)} & \text{sinon} \end{cases}.$$

> Si deux contextes ctx_1 et ctx_2 sont tels que $ctx_1[PC] = ctx_2[PC] = adr \in CCT$, on dit alors que ctx_1 est inclus dans ctx_2 — noté $ctx_1 \subset ctx_2$ — si $\forall var \in ctx_1, ctx_1[var] = ctx_2[var]$ ou $ctx_2[var] = sym_{(var,adr)}$.

> La fonction $concret : VAL \rightarrow CCT^n$ est la fonction permettant de lister l'ensemble des valeurs concrètes possibles d'une valeur.

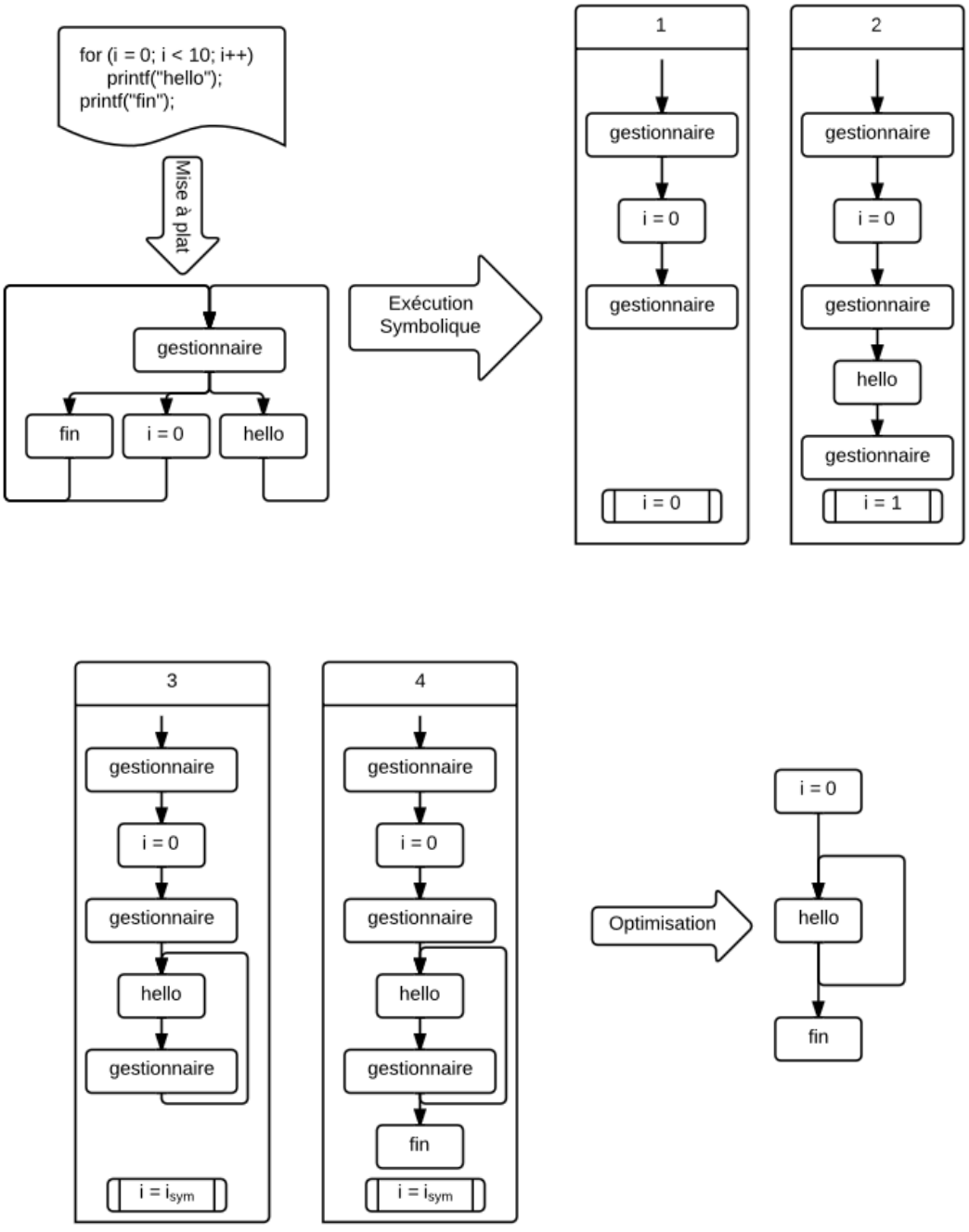

Figure 3. Illustration de la reconstruction du CFG d'une fonction

Par exemple $concret(a\&3)$ retourne le quadruplet $(0, 1, 2, 3)$ et $concret(3)$ retourne le singleton (3).

La fonction $simplifie : \begin{vmatrix} CTX, ASSERT \to VAL \\ ctx, assert \mapsto simplified\_ctx \end{vmatrix}$ est la fonction permettant de simplifier les valeurs du contexte ctx à partir de l'assertion $assert$.

Par exemple, $simplifie(x\&2, x = 1) \mapsto 0$ et $simplifie(If(x > 1, y, z), If((t \neq 0)\&(x > 6), 4 + z, 0) = 5) \mapsto y$.

Une instruction $ins \in INS$ est un couple $(adr, exec) \in (CCT, EXEC : CTX \to CTX)$ où adr correspond à l'adresse de l'instruction et $exec$ la fonction décrivant son exécution.

Par exemple, l'instruction `EOR R1, R0, R0` (R1 = R0 R̂0) située à l'adresse 0x401000 sera représentée par le couple $(0x401000, f_{eor} : ctx_{in} \mapsto ctx_{out})$ où

$$ctx_{out}[v] = \begin{cases} 0 & \text{si } v = \texttt{R1} \\ 0x401004 & \text{si } v = PC \\ ctx_{in}[v] & \text{sinon} \end{cases}.$$

Dans cet article, nous ne considérons pas le cas des codes auto-modifiants (c'est à dire le cas où deux instructions différentes peuvent être situées à une même adresse).

On définit un sous-ensemble d'instructions $GEST\_INS \subset INS$ correspondant à l'ensemble des instructions terminant les gestionnaires utilisés pour la mise à plat du CFG.

Cet ensemble correspond typiquement à l'ensemble des branches indirectes comme `JMP [EAX*4+XXXX]` (x86) ou `ADD PC, PC, R0` (ARM). Il n'est pas nécessaire que ces instructions soient uniquement présentes en fin de gestionnaire.

Le cas où le gestionnaire utilise une structure du type *cascade de if* peut être traité en créant une *méta-instruction* regroupant l'ensemble des instructions de la cascade.

La fonction $desas : CCT \to INS$ est la fonction de désassemblage. Elle permet de faire le lien entre adresse et instruction.

Un *BasicBlock* $bbl \in BBL$ est une structure contenant une liste d'instructions — $bbl.insts$ — et un dictionnaire associant aux successeurs du *BasicBlock* les conditions de transition — $bbl[succs] = cond$ —. Ces successeurs sont soit des *BasicBlocks* soit des *Traces* identifiées par leur adresse.

L'algorithme 1 donne la construction d'un *BasicBlock*.

Une *Trace* $trace \in TRC$ est une structure contenant une liste de *BasicBlocks* — $trace.bbls$ — et un contexte initial — $trace.ctx$ —.

Algorithm 1 Algorithme de construction d'un *BasicBlock*.

Require: $f \in FUN$, $trace \in TRC$, $ctx_{bbl} \in CTX$, $ctx_{bbl}[PC] \in CCT$, $pere \in BBL$, $cond \in VAL$.
▷ f est la fonction à laquelle appartient le *BasicBlock* à créer.
▷ *trace* est la *Trace* contenant le *BasicBlock* à créer.
▷ ctx_{bbl} est le contexte initial du *BasicBlock* et doit contenir la valeur symbolique spéciale *adrret* correspondant à l'adresse de retour de la fonction.
▷ Le *BasicBlock pere* et la condition *cond* éventuels identifient le *BasicBlock* père du *BasicBlock* à construire ainsi que la condition de branchement du père vers le nouveau *BasicBlock*.
function *CreeBbl*(f, *trace*, ctx_{bbl}, *pere*, *cond*)
  $bbl.insts \leftarrow []$
  $fils \leftarrow []$
  ▷ Exécution symbolique du code jusqu'à atteindre la fin d'un gestionnaire ou une condition de branche symbolique ou la fin de la fonction.
  repeat
    $ins \leftarrow desas(ctx_{bbl}[PC])$
    $bbl.insts.append(ins)$
    $ctx_{bbl} \leftarrow ins.exec((ctx_{bbl})$
  until $ins \in GEST\_INS$ or $ctx_{bbl}[PC] \in SYM$ or $ctx_{bbl}[PC] = adrret$
  ▷ Si la fin de la fonction est atteinte, le *BasicBlock* n'a pas de successeur.
  if $ctx_{bbl}[PC] \neq adrret$ **then**
  ▷ Calcul des adresses des successeurs du *BasicBlock* et des conditions de branches éventuelles. C'est l'étape qui permettera la reconstruction du CFG original.
    for $adr \in concret(ctx_{bbl}[PC])$ **do**
      $ctx_{adr} \leftarrow simplifie(ctx_{bbl}, ctx_{bbl}[PC] = adr)$
      $cond_{adr} \leftarrow ctx_{bbl}[PC] = adr$
      if $ins \in GEST\_INS$ **then**
      ▷ Si la dernière instruction d'un *BasicBlock* est une instruction de fin de gestionnaire alors ses successeurs sont des *Traces*.
        $f.traces.append(ctx_{adr})$
        $bbl[adr] \leftarrow cond_{adr}$
      else
      ▷ Sinon ce sont des *BasicBlocks* qui doivent être parcourus.
        $fils.append((ctx_{adr}, bbl, cond_{adr}))$
      end if
    end for
  end if
  ▷ Ajout du *BasicBlock* nouvellement créé aux successeurs de son pere éventuel.
  if $pere \neq \emptyset$ **then**
    $pere[bbl] = cond$
  end if
  ▷ Ajout du *BasicBlock* à la *Trace*
  $trace.bbls.append(bbl)$
  ▷ retourne la liste des *BasicBlock* fils à explorer. **return** (bbl, $fils$)
end function

L'algorithme 2 décrit la construction d'une *Trace*.

> Le découpage d'une fonction en *Traces* à partir d'un contexte initial est une structure $f \in FUN$ contenant le contexte initial — $f.ctx$ et un dictionnaire associant une *Trace* à une adresse — $f[adr] = trace$ —.

L'algorithme 3 permet ce découpage.

Si l'algorithme se termine et si l'exécution symbolique est correcte, alors le découpage de la fonction permet de créer un graphe orienté dont les nœuds sont les *BasicBlocks* des *Traces* et dont les arcs sont donnés par les successeurs des *BasicBlocks*. Ce graphe est alors équivalent à celui de la fonction protégée (avec éventuellement des nœuds non atteignables). Un exemple de représentation graphique du graphe obtenu à partir de ce découpage est donné dans la figure 4.

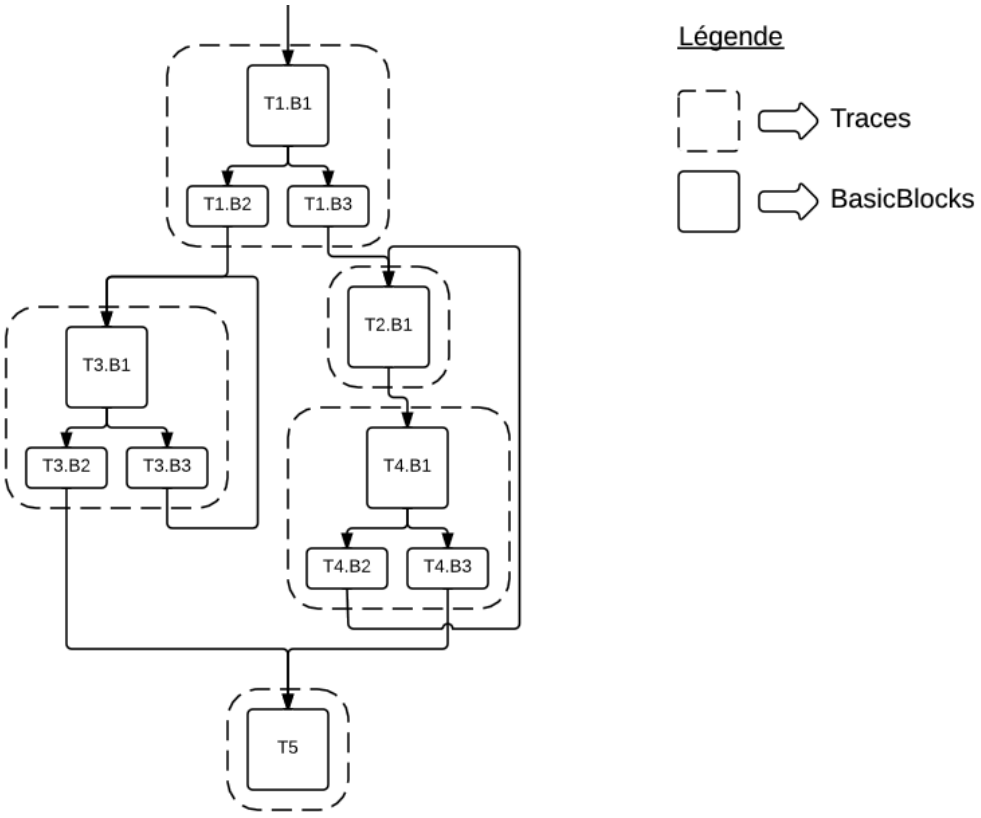

Figure 4. Illustration du découpage d'une fonction en *Traces* et *BasicBlocks*.

Les problèmes pratiques qui se posent sont maintenant de faire une exécution symbolique *correcte* et de minimiser les nœuds non atteignables.

Algorithm 2 Algorithme de construction d'un *BasicBlock*.

Require: $ctx_{trace} \in CTX$, $finFUN$.
▷ f est la fonction à laquelle appartient la *Trace* à créer.
function CreeTrc(ctx_{trace}, f)
$trace.ctx \leftarrow ctx_{trace}$
$trace.bbls \leftarrow []$
▷ La liste des *BasicBlocks* à explorer contient les triplets (contexte à partir duquel créer le bbl, père éventuel, condition de branche sur le bloc à partir du père).
$bbls \leftarrow [(ctx_{trace}, \emptyset, \emptyset)]$
repeat
ctx_{bbl},père, $cond \leftarrow bbls.pop()$
$bbls = bbls || CreeBbl(f, trace, ctx_{bbl}$,père, $cond)$
until $bbls = []$
▷ Ajout ou mise à jour de la *Trace* dans la fonction.
$f[adr_{trace}] = trace$
end function

Algorithm 3 Algorithme de création d'une fonction.

Require: $ctx \in CTX$, $ctx[PC] \in CCT$
▷ Le contexte doit contenir la valeur symbolique spéciale *adrret* correspondant à l'adresse de retour de la fonction.
$f.ctx \leftarrow ctx$
$f.traces \leftarrow [ctx]$
repeat
$ctx_{trace} \leftarrow f.traces.pop()$
$adr_{trace} \leftarrow ctx_{trace}[PC]$
▷ Si l'adresse ne correspond pas à une *Trace* existante ou que le nouveau contexte n'est pas inclus dans celui de l'ancienne *Trace*, une nouvelle *Trace* est créée.
if $adr_{trace} \notin f$ or $ctx_{trace} \subset f[adr_{trace}].ctx$ **then**
▷ Si une *Trace* existait déjà, on fusionne son contexte initial avec le nouveau contexte.
if $adr_{trace} \in f$ **then**
$ctx_{trace} \leftarrow ctx_{trace} \cup f[adr_{trace}].ctx$
end if
$CreeTrc(ctx_{trace}, f)$
end if
until $f.traces = []$ **return** f

4 Implémentation pratique

Dans cette section nous présentons l'application concrète de l'algorithme 1 sur des fonctions de iOS. Ces fonctions sont une implémentation de signature RSA en boite blanche utilisée par les différents périphériques mobiles Apple (iPad, iPhone etc.) afin de *prouver* que ce sont bien des périphériques Apple. Ces fonctions sont protégées à l'aide de la mise à plat du CFG, probablement via la modification de LLVM. Un exemple de CFG d'une de ces fonctions et sa version reconstruite est donné dans la figure 5

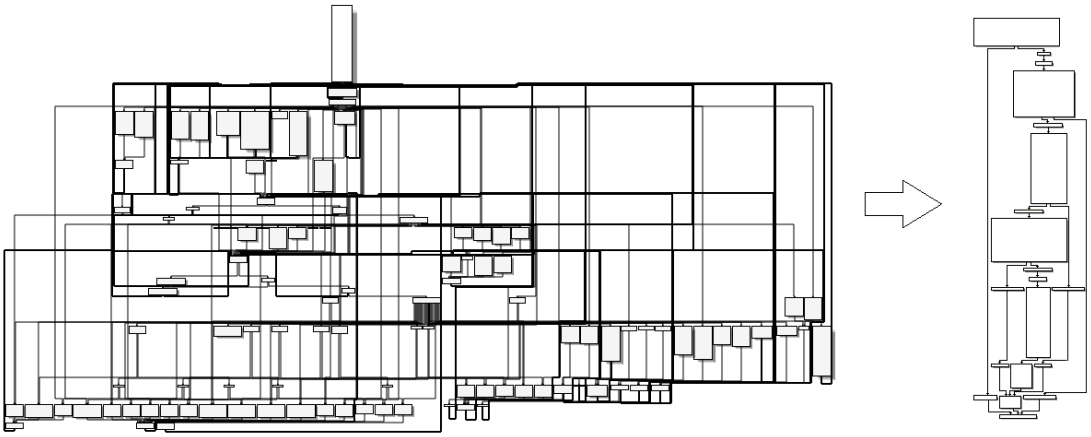

Figure 5. Exemple du CFG d'une fonction avant et après reconstruction

Plusieurs particularités de cette mise à plat sont à noter :

- les identifiants des blocs de bases sont masqués à l'aide de constantes (voir figure 6) ;
- il existe plusieurs gestionnaires pour une fonction donnée (voir figure 7) ;
- certaines variables utilisées pour le calcul de l'adresse du bloc de base suivant sont initialisées dans les blocs précédemment exécutés ;
- certaines variables utilisées pour le calcul de l'adresse du bloc de base suivant sont initialisées à l'aide d'arguments passés à la fonction ;
- de nombreux blocs inatteignables sont ajoutés au CFG.

De plus la fonction brouillée étant une boite blanche RSA, les approches par suivi de données sont inefficaces, les calculs effectués étant trop complexes. Nous avons donc implémenté la méthode décrite dans

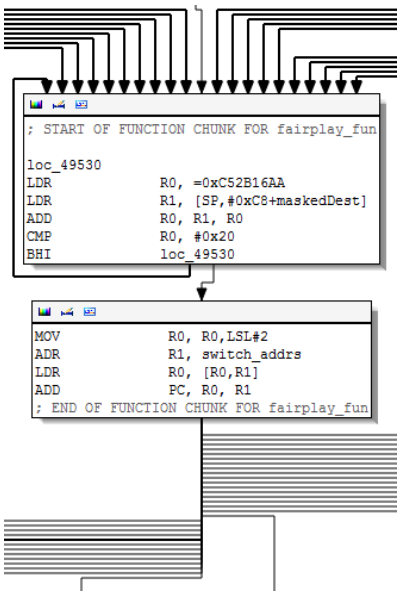

FIGURE 6. Code chargé du débrouillage de l'identifiant de bloc

la section 3 sous la forme d'un plugin Python pour IDA Pro. La suite de cette section décrit les différents choix d'implémentation qui ont été faits.

4.1 Désassemblage

Le désassemblage des instructions est fourni directement par IDA Pro sous la forme d'une structure commune à toutes les architectures dont de nombreux champs sont non documentés. Il a donc fallu retrouver les différents champs par rétro-ingénierie (ainsi les conditions d'exécution des instructions ARM sont enregistrées dans le champ `segpref` de la structure `insn_t`, normalement utilisé pour stocker le segment d'une instruction x86). Chaque instruction désassemblée est enregistrée sous la forme d'un objet Python générique, permettant la réutilisation du code pour différentes architectures.

4.2 Identification des instructions de fin de gestionnaire

La détermination de l'ensemble des instructions terminant les gestionnaires — *INS_GEST* — est un prérequis à l'utilisation de la méthode décrite dans la section 3. Dans le cas de FairPlay, cet ensemble est simplement l'ensemble des branches indirectes, c'est-à-dire les instructions modifiant `PC` dont au moins un des arguments est un registre différent de `PC`. Par exemple `ADD PC, PC, R0` est une instruction de fin de gestionnaire.

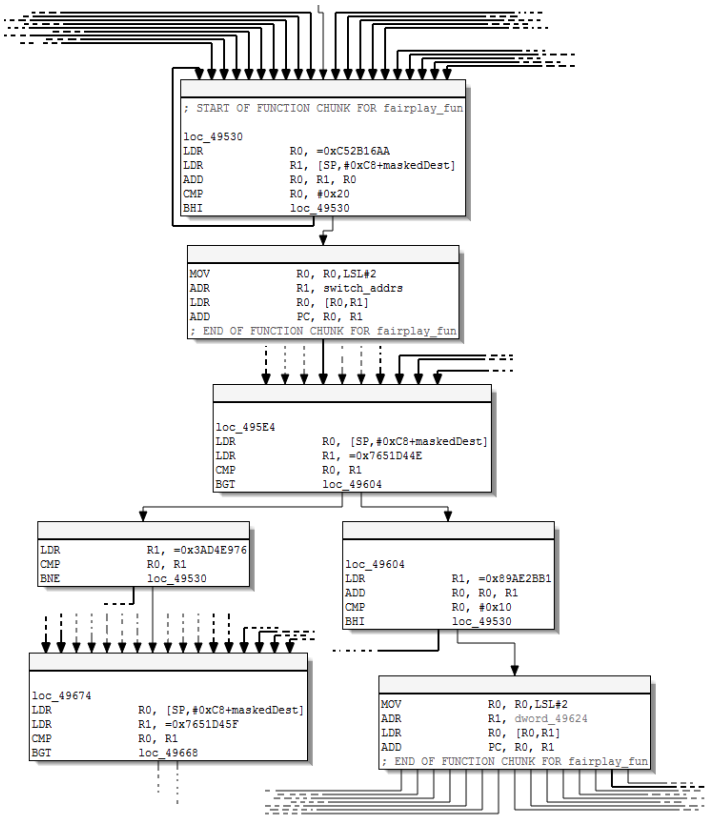

FIGURE 7. Une partie de l'ensemble des gestionnaires différents d'une fonction

4.3 Exécution symbolique

L'exécution symbolique est faite en décrivant la sémantique des instructions en Python. Cette sémantique permet de décrire les effets de bords de l'instruction sur un contexte passé en paramètre. Ce contexte peut être symbolique ou concret, permettant donc aussi bien l'exécution symbolique que l'émulation. Un exemple de description de la sémantique d'une instruction est donné dans la figure 8.

4.4 Contexte

Le contexte est la partie centrale de l'exécution symbolique. Il est en charge de la gestion des différentes variables et des différentes opérations sur les valeurs de ces variables. Ces variables pouvant être des registres, des drapeaux ou des cases mémoire.

Les variables contenues dans un contexte sont considérés comme des vecteurs de 32bits. Ces vecteurs de 32bits sont manipulés à l'aide de Z3 [3] qui est chargé de la simplification des équations et de la concrétisation des valeurs symboliques.

```
class Eor(ArmIns) :
    __metaclass__ = MetaIns

    def _emu(self, context) :
        # les conditions d'execution sont traitees dans la fonction
            ArmIns.emu qui appelle _emu selon les besoins.
        if len(self.ops) != 3 :
            raise EmulationException("%s is not supported yet"%self)
        dest = self.ops[0]
        src1 = self.ops[1]
        src2 = self.ops[2]
        v = context[src1] ^ context[src2]
        context[dest] = v
        if self.affect_flags :
            context[NF] = v.N
            context[ZF] = v.Z
```

Figure 8. Exemple de la description de la sémantique de l'instruction EOR

Les cases mémoires sont considérées comme des mots de 32bits dont l'adresse est alignée sur 32bits. Cette restriction n'est cependant pas bloquante, des accès non alignés ou de taille différente de 32bits pouvant être modélisés comme des concaténations d'accès éventuellement masqués et/ou décalés. Par exemple `WORD ptr[0x401003] = (DWORD ptr[0x401000] & 0xFF) | (DWORD ptr[0x401004] » 24)`.

La pile et la mémoire globale sont dissociées. Pour atteindre une variable dans la pile, il faut que l'adresse déréférencée soit indexée par le symbole spécial `SP`. Cette approximation ne nuit pas en pratique à la précision de l'exécution, le code compilé n'ayant pas connaissance *a priori* de la valeur du pointeur de pile.

4.5 Déréférencement de pointeur et appels de sous-fonctions

La gestion du déréférencement des pointeurs symboliques et des appels de sous-fonctions est une partie sensible de l'exécution symbolique. En effet, une sous-approximation des valeurs possibles d'un pointeur ou des effets de bords d'une fonction peut entrainer la non exploration de l'ensemble du CFG (exemple illustré dans la figure 9) tandis qu'une sur-approximation peut entrainer l'exploration de code non atteignable et peut même aller jusqu'à bloquer le déroulement de l'algorithme 3 dans une boucle infinie (exemple illustré dans la figure 10).

La solution retenue a été de considérer que l'ensemble des variables impliquées dans le calcul des identifiants des blocs de base ont des valeurs constantes de haute entropie et que les fonctions respectent les

```
int foo(int arg)
{
    bool b = false;

    bar(&b, arg); // met la valeur de b a true ou false suivant la
        valeur de arg
    if (b)
    {
        /* non explore si on sous-approxime la fonction bar en ignorant
            ses effets de bord sur b */
    }
}
```

Figure 9. Exemple de non exploration d'une partie du CFG d'une fonction en cas de sous-approximation des valeurs possibles d'un pointeur

conventions d'appels ARM (les registres de R0 à R7 sont potentiellement écrasés, la valeur des auters registres n'est pas modifiée par la fonction).

Les algorithmes 4, 5 et 6 permettent le déréférencement et l'exécution symbolique de fonctions inconnues.

4.6 Optimisation du CFG

Une fois l'algorithme 3 déroulé, le CFG obtenu contient de nombreuses instructions qui sont maintenant inutiles parce que dédiées à la redirection du flux d'exécution. Afin de les supprimer, le code des *Basic-Block* est transformé en pseudo-code, puis optimisé. La transformation en pseudo-code est faite en exécutant symboliquement les traces tout en enregistrant les modifications du contexte sous forme d'affectations. Le pseudo-code est optimisé en supprimant les affectations inutiles (affectant des variables non utilisées dans la suite du code) et des blocs vides. Le résultat obtenu est illustré dans la figure 11.

5 Discussion

L'algorithme proposé dans la section 2 se montre efficace pour la reconstruction de CFG brouillés. Cependant, son fonctionnement dépend de plusieurs choix d'implémentation non triviaux et n'est efficace que dans certains cas spécifiques.

Il est en effet nécessaire de pouvoir différencier les variables ajoutées lors de la mise à plat du CFG des variables initialement présentes afin de pouvoir explorer toutes les branches du CFG mais aussi de ne pas sur-approximer les destinations possibles de branches indirectes.

Algorithm 4 Algorithme de déréférencement en lecture d'une variable.

Require: $adr \in VAL, ctx \in CTX$
  function $deref\_lecture(adr, ctx)$
    $vals_{adr} \leftarrow concret(adr)$
    ▷ Si le nombre de valeurs possibles de l'adresse est supérieur à un certain seuil, un nouveau symbole est créé et retourné
    if $card(vals_{adr}) > MAX\_DEREF$ **then**
      return $ctx.nouveau\_sym()$
    end if
    ▷ Sinon la valeur déréférencée val_{ret} est créée
    $val_{ret} \leftarrow \emptyset$
    for $val_{adr} \in vals_{adr}$ **do**
      $val_{ret} \leftarrow If(adr = val_{adr}, ctx.deref(val_{adr}), val_{ret})$
    end for
    return val_{ret}
  end function

Algorithm 5 Algorithme de déréférencement en écriture d'une variable.

Require: $adr \in VAL, val \in VAL, ctx \in CTX$
  function $deref\_ecriture(adr, val, ctx)$
    $vals_{adr} \leftarrow concret(adr)$
    ▷ Si le nombre de valeurs possibles de l'adresse est supérieur à un certain seuil, l'ensemble des valeurs non constantes ou dont l'entropie est faible dans la pile sont considérées comme atteignables
    if $card(vals_{adr}) > MAX\_DEREF$ **then**
      for $var_{adr} \in ctx.pile$ **do**
        $var_{val} \leftarrow ctx.deref(var_{adr})$
        if $var_{val} \in SYM$ ou $entropie(var_{val}) < MIN\_ENTROPIE$ **then**
          $ctx.set\_deref(var_{adr}, If(adr = var_{adr}, val, var_{val}))$
        end if
      end for
      return ctx
    end if
    ▷ Sinon les valeurs atteignables sont mises à jour
    for $var_{adr} \in vals_{adr}$ **do**
      $var_{val} \leftarrow ctx.deref(var_{adr})$
      $ctx.set\_deref(var_{adr}, If(adr = var_{adr}, val, var_{val}))$
    end for
    return ctx
  end function

```
(void (*)(int)) func_table[] = [foobar1, foobar2];

int foo(int* arg1, int arg2)
{
    int a = 1;

    *arg = arg2; // arg ne pointant pas sur a

    /* si on sur-approxime la valeur de arg en considerant que arg peut
        pointer sur a */
    /* on considere alors que l'ensemble des adresses adr = func_table[
        XXX] sont */
    /* atteignables quelque soit l'entier XXX */
    func_table[a](arg);
}
```

Figure 10. Exemple de sur-approximation de la valeur d'un pointeur entrainant l'exploration d'une grande quantité de code non atteignable.

Paradoxalement, le fait que les variables impliquées dans le calcul des identifiants des blocs de base soient obscurcies simplifie cette différenciation. De plus, le fait que la valeur de ces variables soit constante d'une exécution à l'autre d'une même trace fait que l'algorithme d'union de contexte est efficace bien que simpliste.

De nombreuses améliorations peuvent être apportées à l'algorithme de reconstructions comme à son implémentation, par exemple en considérant les sous-fonctions et en utilisant des algorithmes issus de l'interprétation abstraite pour l'union des différents contextes de début de *Trace*. De même, certaines améliorations simples peuvent être apportées aux algorithmes de mise à plat de CFG pour compliquer considérablement la reconstruction en ajoutant des dépendances virtuelles entre variables et en faisant varier artificiellement les variables impliquées dans le calcul des identifiants de bloc comme proposé dans une présentation de 2009 [1].

La mise à plat de flot de contrôle est une technique qui peut sembler à première vue inefficace face à l'analyse dynamique, elle se montre cependant pertinente dans le cas du brouillage de code complexe et demande à être approfondie. L'auteur espère que cet article encouragera le lecteur à ne plus considérer le brouillage de code uniquement sous l'angle de la mutation de code et le poussera à s'intérresser aux méthodes de brouillage de plus haut niveau.

Algorithm 6 Algorithme d'exécution d'une fonction inconnue.

Require: $ctx \in CTX$
  function $exec\_fonct(ctx)$
    ▷ L'ensemble des registres de R0 à R7 est écrasé
    for $R_i \in \{R0, R1, R2, R3, R4, R5, R6, R7\}$ **do**
      $ctx[R_i] \leftarrow ctx.nouveau\_sym()$
    end for
    ▷ L'ensemble des valeurs non constantes ou dont l'entropie est faible dans la pile est écrasé
    for $var_{adr} \in ctx.pile$ **do**
      $var_{val} \leftarrow ctx.deref(var_{adr})$
      if $var_{val} \in SYM$ ou $entropie(var_{val}) < MIN\_ENTROPIE$ **then**
        $ctx.set\_deref(var_{adr}, ctx.nouveau\_sym())$
      end if
    end for
    ▷ PC est mis à jour
    $ctx[PC] \leftarrow ctx[PC] + 4$
    return ctx
  end function

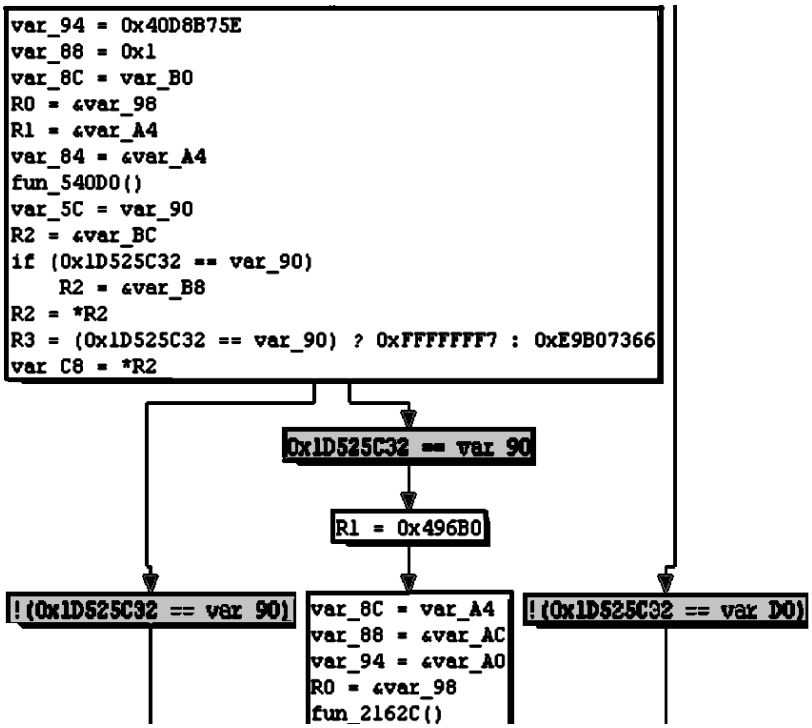

Figure 11. Illustration du code obtenu suite à la transformation des instructions en pseudo-code et à sa simplification

Références

1. Bertrand Anckaert, Mariusz H. Jakubowski, Ramarathnam Venkatesan, and Chit Wei (Nick) Saw. Runtime protection via dataflow flattening, 2009.
2. Jean-Baptiste Bedrune. Digital content protection — how to crack drm and make them more resistant, 2010.
3. Leonardo De Moura and Nikolaj Bjørner. Z3 : an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, 2008.
4. Sébastien JOSSE. *Analyse et détection dynamique de code viraux dans un contexte cryptographique*. PhD thesis, École Polytechnique, 2009.
5. Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel programs with dynamic instrumentation. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 81–92, 2004.
6. Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance : Obstructing static analysis of programs, 2000.