ANNÉE 2017

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : informatique

Ecole doctorale Mathématiques et STIC

présentée par

Pierre Lestringant

préparée à l'Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA) UMR 6074 en partenariat avec Amossys

---

Identification d'Algorithmes Cryptographiques dans du Code Natif

Thèse soutenue à Rennes le 12 décembre 2017

devant le jury composé de :

Jean-Yves MARION
Professeur, U. Lorraine - rapporteur

Pascal JUNOD
Chercheur, Snap Inc. - rapporteur

Sandrine BLAZY
Professeur, U. Rennes 1 - examinatrice

Vincent NICOMETTE
Professeur, INSA Toulouse - examinateur

Marion VIDEAU
Maître de conférences, U. Lorraine en détachement auprès de Quarkslab - examinatrice

Colas LE GUERNIC
Ingénieur de recherche, DGA - examinateur

Frédéric GUIHÉRY
Responsable R&D, Amossys - examinateur

Pierre-Alain FOUQUE
Professeur, U. Rennes 1 - directeur de thèse

Contents

Résumé en Français **4**
1 Identification des Primitives . . . 4
2 Identification des Modes Opératoires . . . 6
3 Évaluation Expérimentale . . . 7
4 Organisation du Manuscrit . . . 8
5 Implémentation . . . 8
6 Liste des Publications . . . 8

Introduction **9**
1 Reverse Engineering . . . 9
2 Cryptographic Notions . . . 10
3 Motivation . . . 11
4 Solution Overview . . . 12
5 Layout of the Thesis . . . 14

1 Related Work **16**
1.1 Basic Techniques . . . 16
1.2 Cryptographic Parameters Retrieval . . . 18
1.3 Binary Code Comparison . . . 19
1.4 Primitive Identification . . . 20

2 Data Flow Graph **33**
2.1 DFG Definition . . . 33
2.2 Sequence of Dynamic Instructions . . . 37
2.3 Construction . . . 40
2.4 Built-in Mechanisms . . . 43

3 Normalization **46**
3.1 Formal Presentation . . . 46
3.2 Practical Aspects . . . 51
3.3 Common Subexpression Elimination . . . 52
3.4 Constant Simplification . . . 54
3.5 Constant Expression Detection . . . 57
3.6 Memory Access Simplification . . . 59
3.7 Memory Coalescing . . . 66
3.8 Commutative and Associative Operation Normalization . . . 70
3.9 Affine Expression Simplification . . . 72
3.10 Operation Size Expansion . . . 73
3.11 Miscellaneous Rewrite Rules . . . 75
3.12 Conclusion . . . 75

4 Subgraph Isomorphism for Primitive Identification **77**
4.1 Signature . . . 77
4.2 Signature Detection . . . 81
4.3 Experimental Evaluation . . . 88

5 Dynamic Slicing for Mode of Operation Identification **105**
5.1 Primitive Identification 105
5.2 Slicing Definition 108
5.3 Adjustments Based on Semantics 112
5.4 Slice Construction 115
5.5 Experimental Evaluation 117

6 Detailed Use Cases **124**
6.1 Automatic Test Vectors Verification 124
6.2 Complete Analysis of an Instant Messaging Application 128

A Cryptographic Algorithms Background **133**
A.1 Primitives 133
A.2 Modes of Operation 136

Résumé en Français

Durant ma thèse, j'ai travaillé à la conception de méthodes automatisées permettant l'identification d'algorithmes cryptographiques dans des programmes compilés en langage machine. Ce besoin bien spécifique trouve une partie de son origine dans le domaine de l'évaluation logicielle. L'utilisation d'algorithmes cryptographiques se fait dans le but d'obtenir des fonctions de sécurité telles que la confidentialité, l'intégrité et l'authenticité pour la cryptographie symétrique ou des fonctions plus diverses comme la signature numérique, l'établissement de secrets partagés ou le chiffrement à clé publique pour la cryptographie asymétrique. Cependant le choix des algorithmes, leur implémentation et leur utilisation au sein d'un programme informatique sont autant de points sensibles pouvant remettre en cause la robustesse de ces fonctions de sécurité. Il est donc primordial dans le cadre de l'évaluation logicielle d'analyser les implémentations cryptographiques afin de s'assurer de leur bon niveau de sécurité. Si dans bien des cas il est possible et plus commode de conduire cette analyse à partir du code source, il n'en demeure pas moins important de pouvoir également opérer à partir du code machine. En effet le code source n'est ni toujours disponible (évaluation pour le compte d'une tierce partie ou avec des informations limitées dans le but de simuler un attaquant réel) ni toujours fiable (biais délibéré ou non entre le code source et le code machine, dû par exemple aux optimisations du compilateur [5]).

L'idée n'est pas ici d'automatiser un type d'analyse particulier (par exemple: vérifier l'absence de corrélation entre temps d'exécution et paramètres secrets pour empêcher les attaques par canaux temporels), mais d'automatiser l'identification des algorithmes cryptographiques, première étape nécessaire à toute analyse plus approfondie. À ce titre, les résultats obtenus ne permettront pas directement dans bien des cas de juger du bon niveau de sécurité des mécanismes cryptographiques, mais serviront de socle à l'évaluateur pour débuter son analyse. Pour ce travail, je me suis limité à la cryptographie symétrique, proposant deux méthodes: une pour l'identification des primitives cryptographiques et l'autre pour l'identification des modes opératoires. Note: ces deux méthodes n'ayant pas été conçues pour l'analyse de code obfusqué, elles n'offrent aucune garantie de bon fonctionnement dans ce domaine.

1 Identification des Primitives

Il est possible, à l'aide d'heuristiques simples (instructions particulières, constantes particulières, taille des blocs de base, fréquence d'exécution, position dans l'arbre d'appels), d'identifier efficacement et de façon sûre un petit nombre de parties du code comme étant de possibles implémentations cryptographiques [33, 81]. La méthode proposée pour l'identification des primitives a donc été conçue pour s'exécuter non sur l'ensemble du programme, mais sur des portions de taille restreinte pouvant être sélectionnées à l'aide de ces heuristiques. Elle utilise des signatures. Une approche par signatures est particulièrement adaptée dans le cas des primitives puisque le nombre de primitives fréquemment utilisées en pratique est faible et que leurs implémentations sont peu sujettes aux variations.

1.1 Graphe de Flot de Données

Le code machine est représenté par une structure de données appelée DFG (pour *Data Flow Graph*). Bien qu'initialement conçue de façon indépendante, cette structure de données se rapproche des *Term Graphs* [60] et des *jungles* [39]. Elle permet de représenter une ou plusieurs expressions composées de symboles d'opération et de symboles de variable sous forme de multigraphe orienté

acyclique. Un sommet v possède une étiquette notée $labV(v)$ qui est soit un symbole d'opération, soit un symbole de variable. Chaque sommet v représente une expression, notée $term(v)$, qui se définit récursivement de la façon suivante:

$$term(v) = \begin{cases} labV(v) & \text{si } v \text{ est étiqueté avec un symbole de variable} \\ labV(v)(term(v_1), ..., term(v_n)) & \text{si } v \text{ est étiqueté avec un symbole d'opération} \end{cases}$$

Dans le second cas, $v_1, ..., v_n$ désignent les prédécesseurs directs de v ordonnés d'après les étiquettes des arêtes les reliant respectivement à v. L'organisation sous forme de graphe a l'avantage par rapport à une organisation plus naturelle sous forme d'arbre, de permettre le partage des sous-expressions communes réduisant fortement les besoins en mémoire et les temps de traitement.

La portion du programme à analyser est vue comme une séquence d'instructions. Cette hypothèse simplificatrice s'explique par le fait que les primitives symétriques contiennent peu d'instructions conditionnelles et que celles-ci ne dépendent généralement pas des arguments d'entrée et de sortie. Par conséquent, le chemin d'exécution ne varie pas d'une exécution à l'autre à l'intérieur du code cryptographique qui peut donc être assimilé à une séquence d'instructions.

Les DFGs sont construits par compositions successives. Chaque instruction de la séquence à analyser est convertie vers un petit DFG. Ceux-ci sont ensuite composés pour obtenir le DFG final représentant la totalité de la séquence. Ce mode de construction se révèle particulièrement utile par la suite puisque même une fois modifiés il sera toujours possible de composer les DFGs (technique utilisée lors de l'agrandissement de la fenêtre d'analyse par exemple). Un effort particulier a été apporté à la traduction des instructions vectorielles (couramment utilisées par certaines implémentation cryptographique).

L'utilisation de DFGs pour représenter le code machine permet de répondre à un certain nombre de besoins: réécriture d'expressions tout en préservant une certaine notion de sémantique appelée similarité observable, comparaison d'expressions et recherche de sous-expressions à travers la notion d'isomorphisme de graphes, *slicing*, et même visualisation par un opérateur d'une partie du code. Cette représentation sera également utilisée pour l'identification des modes opératoires.

1.2 Normalisation

Le but de la phase de normalisation est de supprimer au maximum les différences qui peuvent exister entre plusieurs implémentations d'une même primitive afin de pouvoir détecter avec un petit nombre de signatures une grande variété d'implémentations. Durant la phase de normalisation, un ensemble de règles de réécriture est appliqué itérativement au DFG à normaliser jusqu'à l'obtention d'un point fixe appelé forme normale. Une version réduite de la phase de normalisation est utilisée pour l'identification des modes opératoires.

Deux DFGs sont similaires de façon observable si pour toutes affectations identiques de leurs variables d'entrée on obtient le même ensemble de valeurs pour leurs sommets de sortie. On démontre que si chaque règle de réécriture préserve localement la similarité observable alors un DFG et sa forme normale sont similaires de façon observable. Bien que cette propriété seule ne permette pas de démontrer la validité de la méthode d'identification des primitives, elle n'en demeure pas moins capitale pour obtenir une solution fiable en pratique.

Par facilité d'implémentation mais aussi de présentation, les règles de réécriture sont regroupées en familles appelées mécanismes de normalisation. Onze mécanismes de normalisation sont utilisés. (1) *Constant folding*: diverses simplifications numériques, possibles dès lors qu'une opération possède un ou plusieurs opérandes constants. (2) Détection d'expressions constantes: utilisation de masques d'influence pour identifier les expressions ne pouvant prendre qu'une seule valeur et remplacement de ces expressions par les constantes adéquates. (3) Suppression des sous-expressions communes: fusion de plusieurs opérations partageant les mêmes opérandes. (4) Simplification des accès mémoire: parmi les opérations utilisées dans les DFGs, deux permettent d'accéder à la mémoire: `load` (lecture) et `store` (écriture). Les motifs suivants sont simplifiés dans la séquence des accès effectués à une même adresse mémoire: $(\texttt{load}_i, \texttt{load}_{i+1})$, $(\texttt{store}_i, \texttt{load}_{i+1})$ et $(\texttt{store}_i, \texttt{store}_{i+1})$. La comparaison des adresses mémoire est un point particulièrement sensible. Pour l'identification des primitives, les adresses sont comparées statiquement de façon

sûre. Pour l'identification des modes opératoires, on utilise en revanche des valeurs d'adresse concrètes obtenues pour une exécution donnée. (5) Distribution des constantes: développement des expressions mettant en jeu des constantes réparties sur plusieurs opérations distributives. (6) Élargissement des opérations: chaque sommet d'un DFG possède un attribut de taille. Les opérations de petite taille sont élargies jusqu'à l'obtention d'une taille canonique. (7) *Memory coalescing*: regroupement des accès mémoire de petite taille effectués à des adresses contiguës. (8) Simplification des expressions affines: règles de réécriture dédiées à la simplification d'expressions complexes composées exclusivement d'additions, de soustractions et de multiplication par des constantes. (9) Fusion des constantes: regroupement de deux opérations commutatives et associatives possédant toutes deux au moins un opérande constant. Remarque: ce mécanisme de normalisation diffère du suivant. Son objectif est de provoquer de nouvelles situations propices aux simplifications numériques et non de normaliser des ensembles d'opérations commutatives et associatives. (10) Normalisation des opérations commutatives et associatives: même transformation que pour le précédent mécanisme de normalisation, mais déclenchée par des conditions différentes (absence de successeur, appartenance à une même fonction). (11) Divers: règles de réécriture n'appartenant à aucune des catégories précédemment citées, effectuant principalement des remplacements entre des opérations équivalentes.

1.3 Détection de Signatures par Isomorphisme de Sous-Graphe

Grâce à la phase de normalisation un grand nombre d'implémentations d'une même primitive converge vers une même forme normale. Cette forme normale constitue une signature permettant d'identifier la primitive en question. Les signatures sont ici de grande taille: elles ne sont pas limitées à quelques opérations distinctives, mais couvrent la majeure partie des algorithmes auxquels elles se rapportent. Cela permet d'obtenir des informations détaillées sur la totalité de l'algorithme et notamment de localiser les paramètres d'entrée et de sortie ce qui se révélera indispensable par la suite pour l'identification des modes opératoires.

Afin d'augmenter la robustesse de la méthode et notamment de faire face à d'inévitables défaillances de la phase de normalisation, on a recours à la composition de signatures. Grâce à ce mécanisme des parties disjointes, d'une primitive peuvent être identifiées séparément par de petites signatures spécifiques. Les informations ainsi obtenues sont alors rassemblées par une signature principale couvrant la totalité de la primitive. Deux niveaux sont décrits ici, mais la solution se généralise à un nombre quelconque de compositions.

Tout comme lors de la phase de normalisation, la forme normale est obtenue par compositions successives de règles de réécriture, la détection de la signature principale repose sur la composition de signatures partielles. La ressemblance pourrait être confondante, si ce n'est que les règles de réécritures opèrent par substitution alors que l'on choisit de rechercher les signatures composées en effectuant des superpositions. En procédant à une superposition au lieu d'une substitution, on écarte au prix d'une complexité accrue le délicat problème de la convergence (un DFG peut avoir plusieurs formes normales), laissant toute liberté à l'utilisateur pour la création des signatures.

La recherche des signatures se fait grâce à l'algorithme d'Ullmann [72] qui permet d'énumérer les isomorphismes de sous-graphe. Celui-ci fait l'objet de quelques ajustements afin de permettre la recherche de signatures composées par superposition ainsi que mentionné précédemment.

2 Identification des Modes Opératoires

L'identification des modes opératoires s'inscrit dans le prolongement de l'identification des primitives. En effet, la méthode proposée requiert en entrée des informations détaillées sur la localisation des primitives et de leur paramètres d'entrée et de sortie. Il est donc nécessaire d'identifier d'abord les primitives pour pouvoir ensuite identifier les modes opératoires. Le constat qui avait motivé l'usage des signatures pour l'identification des primitives (peu d'algorithmes utilisés en pratique et une forte stabilité de leur implémentations) ne semble pas se reproduire pour les modes opératoires. En revanche, bon nombre de modes opératoires sont relativement peu complexes. Cela n'étant plus strictement nécessaire, l'idée est alors de ne plus avoir recours à une méthode automatisée pour la reconnaissance des motifs distinctifs, mais de laisser cette tâche à l'utilisateur pour bénéficier d'une plus grande flexibilité. L'essentiel de la méthode se résume alors à extraire d'un DFG une

slice (terminologie empruntée au domaine du *program slicing* auquel notre méthode s'apparente), c'est-à-dire une représentation synthétique des principaux transferts de données intervenant entre les différentes exécutions des primitives. Cette *slice* est retournée à l'utilisateur pour une interprétation manuelle.

2.1 Définition d'une Slice

A partir d'un DFG contenant plusieurs exécutions de primitives, une slice se définit comme le plus petit sous-graphe préservant les distances entre les paramètres d'entrée et de sortie de ces primitives. De façon informelle, une *slice* est complète si elle inclut suffisamment d'éléments pour permettre l'identification du mode opératoire et elle est lisible si elle n'inclut que des éléments strictement nécessaires à cette identification. Si, intuitivement, il est évident qu'une *slice* doit être complète, il est également important qu'elle soit lisible: malgré un motif distinctif simple si, la recherche doit s'effectuer dans un DFG de grande taille, elle risque de s'avérer longue et fastidieuse pour l'utilisateur. La définition d'une *slice* donnée précédemment permet d'atteindre un bon compromis entre ces deux notions qui en pratique sont difficilement conciliables.

Cette définition repose exclusivement sur des aspects syntaxiques. Cela a l'avantage d'imposer des contraintes fortes sur la forme, garantissant une bonne lisibilité. Cependant certains aspects sémantiques doivent être impérativement pris en compte conduisant à quelques ajustements. Le premier concerne les opérations d'accès à la mémoire. Lorsqu'une opération de lecture accède un emplacement mémoire déjà précédemment accédé par une autre opération du DFG, son résultat dépend de cette seconde opération. Par défaut cette dépendance n'est pas représentée par une arête dans les DFGs. Pour y remédier nous avons recours au mécanisme de normalisation intitulé simplification des accès mémoire, en utilisant des valeurs concrètes pour comparer les adresses. Le second ajustement utilise des masques d'influence afin de filtrer les chemins ne représentant pas une relation d'influence effective entre deux opérations d'un DFG.

2.2 Extraction d'une Slice

L'identification des modes opératoires nécessite souvent une fenêtre d'analyse plus large que celle(s) ayant pu être utilisée(s) pour l'identification des primitives. Une première étape consiste donc à construire le DFG de cette fenêtre d'analyse élargie en utilisant notamment la composition de DFGs afin de pouvoir importer directement les résultats ayant été obtenus lors de l'identification des primitives.

Le DFG subit alors une étape de normalisation qualifiée de légère en comparaison à la phase de normalisation utilisée pour l'identification des primitives. L'objectif est d'une part de forcer la représentation de certains aspects sémantiques au niveau de la syntaxe (ajustement relatif aux accès mémoire précédemment mentionné) et d'autre part de réaliser un certain nombre de simplifications à moindre coût qui ne seront que bénéfiques pour la lisibilité des *slices*.

Á notre connaissance, il n'existe pas d'algorithme polynomial permettant de calculer le plus petit sous-graphe préservant les distances pour un ensemble de sommets avec un facteur d'approximation qui serait pertinent pour notre problème (graphes peu denses). A défaut, nous décomposons le problème entre d'une part la recherche des plus courts chemins pour chaque paire de sommets concernés, et d'autre part la résolution d'un problème de couverture minimale. Le premier point ne pose pas de difficulté en pratique malgré un nombre de plus courts chemins pouvant être théoriquement exponentiel par rapport au nombre de sommets. Le second point qui revient à sélectionner k ensembles parmi n ensembles tel que leur union soit minimale, est résolu de façon approchée et sans garantie aucune par un algorithme glouton.

3 Évaluation Expérimentale

Pour évaluer ces deux méthodes d'identification, nous avons eu recours à un ensemble de courts programmes de test. Cet ensemble couvre différents algorithmes (AES, MD5, RC4, SHA1 et XTEA pour les primitives et CBC, CTR et HMAC pour les modes opératoires), différentes implémentations dont une grande partie est issue de bibliothèques cryptographiques renommées et

différentes conditions de compilation (compilateurs et options de compilation). Enfin, pour mettre en pratique l'ensemble de la démarche, elle a été appliquée à des programmes réels.

4 Organisation du Manuscrit

Un premier chapitre introductif permet de présenter le contexte, les motivations et propose un bref aperçu des deux méthodes d'identification. Le chapitre 1 est dédié à l'état de l'art. Il évoque les techniques pouvant être facilement utilisées en pratique: recherche de constantes et identification de bibliothèques liées statiquement, ainsi que des problèmes proches: récupération de paramètres cryptographiques (clé de chiffrement et message en clair) et comparaison de codes binaires. Mais il s'agit surtout de revenir en détails sur les deux méthodes concurrentes pour l'identification des primitives, à savoir: la recherche de relations particulières entre valeurs d'entrée et de sortie et la détection de l'effet d'avalanche dans le graphe de flot de données. Des résultats expérimentaux sont fournis afin de pouvoir comparer ces deux méthodes à celle que nous proposons. Le graphe de flot de données est présenté au chapitre 2. L'étape de normalisation est abordée au chapitre 3. Le chapitre 4 conclut la présentation de la méthode d'identification des primitives, en détaillant la recherche de signatures dans le graphe de flot de données et en proposant des résultats expérimentaux. Le méthode d'identification des modes opératoires est décrite au chapitre 5. Enfin, le chapitre 6 propose deux cas d'utilisation concrets qui permettent de préciser la mise en application des méthodes précédemment décrites dans une contexte opérationnel réel. Pour le lecteur qui ne serait pas familier avec les algorithmes cryptographiques abordés tout au long de ce document, un rappel est disponible dans l'annexe A.

5 Implémentation

Une implémentation complète des deux méthodes d'identification ainsi que la totalité des cas de test ayant servis à l'obtention des résultats expérimentaux sont disponibles à l'adresse suivante: `https://github.com/plestrin/bacs`. Remarque: cette implémentation ne permet l'analyse que de programmes compilés pour l'architecture IA-32.

6 Liste des Publications

- Pierre Lestringant, Frédéric Guihéry and Pierre-Alain Fouque. Assisted Identification of Mode of Operation in Binary Code with Dynamic Data Flow Slicing. In *Applied Cryptography and Network Security - ACNS 2016*.
- Pierre Lestringant, Frédéric Guihéry and Pierre-Alain Fouque. Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. In *ACM Symposium on Information, Computer and Communications Security - ASIA CCS 2015*.

Introduction

1 Reverse Engineering

Reverse engineering is the process of extracting knowledge from a technological object to understand how it was built or how it works. This practice exists in many technological fields such as chemistry, electronics or biology. This work deals exclusively with reverse engineering of computer programs, and more particularly, of programs compiled into machine code.

There are various reasons for reverse engineering programs compiled into machine code (also called binary programs). For instance, one may need to fix a bug in a legacy system the source code of which no longer exists, or one may want to bypass a restriction in a program without the consent of its creator (to create unauthorized duplicates or to access restricted content) or, as a last example, one may need to build a system which interfaces with another system the specifications of which are not available. The reverse engineering techniques presented in this document, because they target cryptographic algorithms, are mainly motivated by security concerns. These techniques can be used either to find security flaws in binary programs or, on the contrary, to verify that none exists. Yet, because they contribute to the global understanding of binary programs containing cryptographic algorithms, they have an interest in many other situations, including those mentioned previously.

1.1 Black Box and White Box

A program is analysed in black box if the analyst has no access to its machine code. In this situation, the only way to obtain knowledge on the program is to observe its executions. The analysis is either passive if the analyst is only able to monitor inputs and outputs of executions of the program, or active if the analyst is also able to submit chosen inputs to the program. [68] is an example of passive black box reverse engineering of cryptographic algorithms. By looking at an RSA public key, the authors are able to classify with a high accuracy the RSA implementation which generated that key.

By contrast, a program is analysed in white box if the analyst has access to its machine code. In this work we only consider the white box situation.

1.2 Obfuscation

An obfuscation technique is a program transformation which does not alter the program functionalities but makes the reverse engineering task more difficult. There are as many obfuscation possibilities as there are reverse engineering techniques. For instance, obfuscation may target static or dynamic analysis and manual or automated inspection. To the best of our knowledge, there is no obfuscation technique able to defeat a wide range of reverse engineering attempts with only a minor performance penalty. To achieve a large and efficient protection, one needs to use simultaneously multiple weak obfuscation techniques. The quality of the protection depends on the number of obfuscation techniques as well as on their rarity. If an obfuscation technique is broadly used, specific tools will be developed to mitigate its impact on reverse engineering procedures. By combining several obfuscation techniques together and by introducing variations, one will have better chances to obtain a protection that will require specific reverse engineering effort to remove.

Obfuscated code is not addressed by this work. It means that the methods presented in this document were not devised to deal with obfuscated code specifically. But it does not mean that

they are totally worthless to analyse obfuscated programs. If it is possible to undo the obfuscation transformations or if they do not directly affect parts of a program execution which contains cryptographic algorithms, it will be possible to use our methods nonetheless. For instance, a basic but common obfuscation strategy which consists in using packing plus various anti-debug and anti-virtualization tricks remains perfectly compatible with our identification methods. In fact, these methods rely on execution traces, thus they are unaffected by whatever packing method is used. And anti-debug and anti-virtualization tricks which may prevent us from collecting execution traces, can be removed manually before executing our identification method.

2 Cryptographic Notions

2.1 Symmetric and Asymmetric Cryptography

In symmetric cryptography a secret is shared by the communicating parties. This secret is used to fulfil two security notions: confidentiality and authenticity. Confidentiality ensures that an encrypted message does not leak any information about its initial content to a party with no knowledge of the shared secret. Authenticity ensures that only parties in possession of the shared secret can write messages. A third security notion, called integrity, falls within the realm of symmetric cryptography even though it does not require a shared secret. Integrity ensures that a message is not altered during its storage or transmission.

By opposition, all other cases where communicating parties do not possess a shared secret, belong to the field of asymmetric cryptography. Asymmetric cryptography relies on strong mathematical constructions to achieve richer and more diverse security notions. The most well known usage of asymmetric cryptography is perhaps public-key encryption. In public-key encryption anyone can encrypt messages using a public key but only those who possess the private key secretly attached to the public key are able to decrypt messages. Other commonly used asymmetric schemes are: key exchange and digital signature algorithms.

From a practical point of view, symmetric algorithms are simple, fast and they are used to process large amount of data. Meanwhile, asymmetric algorithms are complex, slow and they are only used at key points of cryptographic protocols to provide security properties which are impossible to fulfil otherwise.

This work is limited to the identification of symmetric algorithms. There are two main reasons to explain this choice.

- First, programs are unlikely to contains their own implementation of asymmetric schemes. Instead they probably use one of the few available cryptographic libraries. From our experience, it takes a software developer at most a couple of days to implement a symmetric scheme from scratch, while it would probably take a cryptographer more than a month to do the same for an asymmetric scheme. As an example, we did a survey of open source SSL libraries and it showed that a large number of them relies on the same library to implement big integers. Hence, in our opinion, the problem of identifing asymmetric schemes can be solved in many cases by looking for external libraries (refer to Section 1.1.1).

- Second, because asymmetric schemes are significantly more complex than symmetric ones, the analysis of their binary implementations is also more challenging. In our opinion it was more reasonable to start with symmetric cryptography and unfortunately we ran out of time before starting anything serious regarding asymmetric cryptography. Note that the methods we devised cannot be easily adapted to work with asymmetric schemes. First, the additional complexity diminishes the chance of success and second these methods rely on specificities of the cryptographic objects they target, specificities which are unlikely to be shared by asymmetric schemes.

2.2 Primitives and Modes of Operation

Symmetric schemes are composed of two abstract types of constructions: primitives and modes of operation. Primitives are the most fundamental constructions: they do not rely on any other. Yet

primitives on their own are limited and a second layer of cryptographic constructions is necessary to obtain security notions such as confidentiality and authenticity. For instance a block cipher, which is a particular type of primitive, can only encrypt or decrypt small messages of fixed length. To obtain confidentiality for messages of arbitrary length, one needs a mode of operation to specify how to repeatedly execute the block cipher. For reasons which are explained in the following sections, the distinction between primitives and modes of operation is essential for this work.

3 Motivation

3.1 Security Concerns about Cryptography in Software

Design and implementation of cryptographic schemes in software is a difficult and error prone task especially for non-cryptographers. We give below a brief overview of common mistakes which affect the design and implementation of cryptographic schemes and can compromise the very security notions they were initially supposed to achieve. This summary is mostly focused on symmetric cryptography.

- The choice of algorithms may be problematic. Algorithms for which practical attacks have been published, must no longer be used. For instance practical collisions were found for MD5 [79] and SHA1 [67], but yet these hash functions are still used in many applications. Furthermore algorithms which have not been thoroughly analysed by cryptographers should be considered as insecure. This includes any software developers' attempt to devise their own cryptographic schemes. For instance in the XBox security system, Microsoft's engineers used successively RC4 and TEA to authenticate the bootloader [66]. But none of these two authentication schemes were able to prevent practical attacks from altering the boot sequence.

- Algorithm implementations may be problematic. Poorly implemented algorithms may leak information about secret data. The cache timing attack against table implementations of AES [71] and the recent attack based on acoustic leakage against RSA [30] are good example of practical side-channel attacks. Modes of operation are also subtle to securely implement. It is well-known that padding schemes used in modes of operation are highly sensitive. For instance bad paddings have led to devastating attacks on many IETF standards [73].

- Finally input parameters may be problematic. Here by input parameters we refer to the key and to the Initialisation Vector (IV). For schemes with keys of variable length, keys must be sufficiently long. Keys must be unpredictable (hard coded keys for instance only provide a false sense of security). IVs must be compliant with the requirements of the mode of operation. For instance predictable IVs in CBC mode have led to practical attacks against SSL and TLS [63].

The conclusion to be drawn is that cryptographic schemes used in software need to be analysed to determine the actual level of security they provide. The objective is, for an attacker, to find security flaws in order to exploit them and, for a concerned user, to check that he does not put himself at risk using a particular piece of software.

A first step to analyse the security of cryptographic schemes used in software is to identify and locate their main algorithms. Identity information can be used to flag deprecated algorithms and location information can be use to start an in-depth analysis of their implementation. The goal of this work is to propose automated solutions to identify and locate both primitives and modes of operation in binary programs.

3.2 Native Code Analysis

Obviously, it is easier to verify the security of cryptographic schemes by examining their source code or high level descriptions. Yet the ability to perform this verification directly on binary programs remains of great importance for several reasons.

First, source code and documentation are not always available. This is the case for an attacker which tries to find security flaws in a closed source program without the consent of its creator. This is also the case for a security analyst which tries to assess the security of a program on behalf of a third party, or with only limited information in order to simulate a real attacker.

Second, source code and documentation are not always to be trusted. One does not execute documentation nor source code. Only the security of binary programs really matters. There can be unintended deviations from source code and documentation. For instance compilers may introduce security breaches [9]. In this regard, we found during this work that in one of the AES implementation of the Crypto++ library, compiler optimizations remove the protection designed to avoid cache timing attacks [5]. Besides, one may suspect deliberate deviations. In fact, without the compiler toolchain, it may be difficult to determine with accuracy if a binary program truly derives from a given source code. In the context of software evaluation, compiler toolchains are not always provided by software development companies nor is the totality of source code.

3.3 Additional Reasons to Identify Cryptographic Algorithms

As explained previously, the main reason to reverse engineer cryptographic schemes is to test their security. But the ability to rapidly identify cryptographic algorithms has other advantages. In fuzz testing for instance, if the targeted application performs integrity checks on its input, malformed inputs will systematically be rejected. With the ability to automatically identify checksums and hash functions, one can devise a fuzzing system able to bypass integrity checks [75]. As another example, one may be interested in identifying cryptographic schemes not for the schemes themselves but for the data they manipulate. For instance, dumping decrypted data may be useful to remove Digital Right Management (DRM) protections [74] or to analyse communication of malware [15]. Finally, as implicitly pointed out by this work, detecting symmetric schemes is relatively easy compared to other types of algorithms. It might be a good idea to start the reverse engineering of a binary program by looking for cryptographic algorithms, especially if it can be done at a minimal cost. Knowledge retrieved from this first step may be providential to understand how a program processes its input / output.

4 Solution Overview

To identify cryptographic schemes we use a bottom-up approach. We devised two identification methods: one for primitives and one for modes of operation. The primitive identification method is executed first and its results are passed to the mode of operation identification method. Both methods rely on the same data structure to represent assembly code. This data structure, called a Data Flow Graph (DFG) has several advantages. It can be used to rewrite code, that is to say, modify code expressions without breaking their original semantics. It can also be used to search code for known expressions. And finally, it can be used to a certain extent by human analysts to visualize piece of code. DFGs are constructed from sequences of instructions. In practice, to obtain sequences of instructions, programs are executed in a monitored environment and the executed instructions are recorded in execution traces. A flowchart which summarizes both identification methods is given in Figure 1.

4.1 Primitive Identification

We use signatures to identify primitives. A signature is a DFG which represents one or several distinctive expressions. Signatures are compared to the DFG representing the program to analyse. To minimize the number of signatures, the DFG representing the program to analyse is first normalized. The goal of the normalization process is to remove some of the small variations which may exist between different machine-code implementations of the same algorithm.

The signature approach is particularly effective to identify primitives for two reasons. First, the number of primitives which are frequently used in software is relatively small. Among the possible reasons to explain this observation is the existence of well established standards such as AES and SHA. Second, there are not many variations among implementations of these primitives. The existence of reference implementations is a possible explanation. Moreover, these implementations

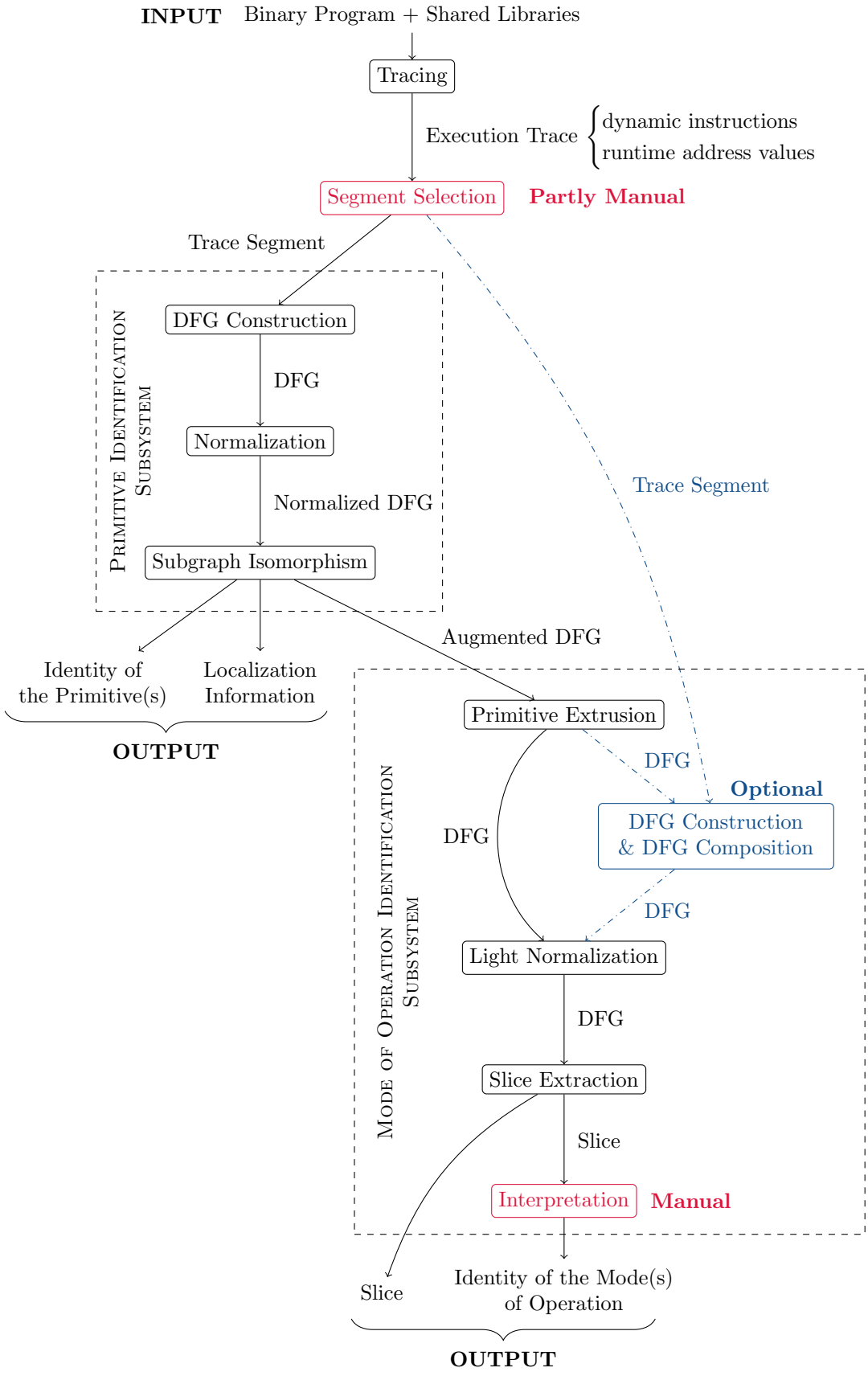

Figure 1: Solution overview.

have carefully been studied in term of both performance and security, leaving not much room for creative software engineering. As a consequence, the number of signatures should remain relatively small. Note that the signature approach is extremely accurate and provides a level of details which outperforms any other identification methods. By creating large signatures, one can even automatically locate the input and output parameters of primitives.

In a nutshell, our primitive identification method is a three step process. First, starting from an execution trace segment, we create its DFG representation. Second, we normalize this DFG using rewrite rules. And third, we search the normalized DFG for signature instances using a subgraph isomorphism algorithm. These three steps are represented on the top left of Figure 1.

Claim: It is possible to detect a wide range of primitive instances using only a few signatures.

4.2 Mode of Operation Identification

We do not use signatures to identify modes of operation. Instead, we produce a synthetic representation of the main data transfers occurring between primitive executions. This synthetic representation, called a slice, is extracted from the DFG representing the program to analyse. To extract a slice, we need to know the location of each primitive execution. This knowledge is obtained from the primitive identification method which must be executed prior to the mode of operation identification method. Slices are manually interpreted by users.

Manual interpretation of slices is possible on two conditions. First, modes of operation must be relatively simple. Otherwise it will be difficult for a human analyst to recognize their complex data flow patterns. Second, slices must only contain what is strictly necessary for mode of operation identification. Otherwise it will be difficult for a human analyst to search slices for distinctive data flow patterns.

Manual interpretation has two advantages over automated pattern matching techniques. First, manual interpretation is more flexible than automated pattern matching techniques. Second, returning a synthetic representation seems a good solution to bridge the gap between automated processing and manual analysis. Automated pattern matching techniques often produce fully processed results which may be hard to seize by a human analyst should he have to pursue the analysis manually.

We chose automated signature detection to identify primitives because it requires to find few but complex data flow patterns. On the contrary, we chose automated synthesis and manual interpretation to identify modes of operation because it requires to find numerous[1] but simple data flow patterns.

To conclude, our mode of operation identification method is only partly automated since slices must be interpreted manually. The method is divided into three steps. First, we delete parts of DFGs which correspond to primitives executions. DFGs are obtained from the primitive identification method. In case these DFGs do not cover sufficiently large trace segments, it is possible to extend their coverage thanks to DFG composition. Second DFGs go through a light normalization process the objective of which is to prepare DFGs for slice extraction. And finally slices are extracted. The mode of operation identification method is depicted on the bottom right of Figure 1.

Claim: It is possible to manually identify most modes of operation based on a synthetic representation of the data transfers occurring between primitive executions.

5 Layout of the Thesis

Related works are presented in Chapter 1. This chapter contains detailed descriptions of existing solutions to identify cryptographic algorithms as well as experimental results which can be used for comparative purpose. DFGs are presented in Chapter 2. Chapter 3 is dedicated to the normalization mechanisms, used for the most part by the primitive identification method. Signature detection is discussed in Chapter 4. This chapter contains experimental results for the primitive

[1] Compared to primitives, implementations of modes of operation are more diverse and there is a greater number of commonly used modes of operation.

identification method. The mode of operation identification method is described in Chapter 5. This chapter also contains experimental results for the mode of operation identification method. Finally in Chapter 6 we conclude this work with two practical examples. In Appendix A, we give a short description of the main cryptographic algorithms mentioned in this document.

Chapter 1

Related Work

Identification of cryptographic algorithms in binary programs has already been studied. Practical solutions exist in the industry (Section 1.1) and more elaborate solutions have been proposed in the academic world (Section 1.4). We also mention in this chapter two closely related fields of research which are: automatic cryptographic parameters retrieval (Section 1.2) and binary code comparison (Section 1.3).

Contents

1.1 Basic Techniques 16
1.1.1 Statically Linked Libraries 16
1.1.2 Specific Constants 17
1.2 Cryptographic Parameters Retrieval 18
1.2.1 Plaintext Extraction from Running Programs 18
1.2.2 Key Extraction from Memory Dumps 18
1.3 Binary Code Comparison 19
1.3.1 Examples 19
1.3.2 Relevance to Cryptographic Algorithm Identification 20
1.4 Primitive Identification 20
1.4.1 Input/Output Relation 21
1.4.2 Avalanche Effect 28

1.1 Basic Techniques

In this section we present two solutions that are commonly used in practice to reverse engineer binary programs which contain cryptographic code. The first one deals with the identification of statically linked libraries. The second one is related with the identification of specific constants.

1.1.1 Statically Linked Libraries

Many programs that use cryptography do not contain their own implementation of cryptographic algorithms. Instead, they use external cryptographic libraries. When it is the case, the task of the analyst may be considerably facilitated. If the cryptographic library is documented and trustworthy, the analyst will be able to answer most questions by simply looking at the functions of the library which are executed by the program. Unfortunately, for portability reasons, these libraries are often statically linked. When it happens, there is no way at first glance to distinguish the code of the program from the code of the library. A basic idea is then to compare the program with every known cryptographic library to see if they share common parts. This simple strategy can benefit from the following improvements.

- The space required to store all versions of all libraries produced for all platforms is relatively important. To reduce storage consumption, we can only store for each function a hash value instead of its full content.

- Some addresses are modified at link time and at load time. We must identify these addresses and skip them while comparing the content of two functions.
- Several functions may be perfectly similar except that they themselves call different sub-functions. To distinguish them, we must characterize similar functions according to the identity of the sub-function(s) they call. As a consequence, several passes are now necessary to completely identify some functions.

This solution was implemented at a large scale in IDA [2] (that is to say not only for cryptographic libraries). IDA comes with a database of signatures covering the most commonly used libraries, compilers and platforms. When a function is identified, it is automatically renamed. It is possible to create new signatures for libraries or compilation environments which are not covered by the initial database. A similar feature also exists in Radare2 [61] under the name of *zignature* but very little documentation is available on the subject.

To conclude, identification of statically linked libraries is not specific to cryptographic libraries. It is probably the first technique to try while starting to reverse engineer a piece of software. In its most basic form, it is precise and extremely efficient. But to get positive results, one needs to possess a signature covering the exact library version compiled with the exact compiler and compiler options for the exact architecture. More flexible techniques to compare two pieces of binary code are presented in Section 1.3.

1.1.2 Specific Constants

Symmetric cryptographic algorithms often contain specific constants. These constants range from a single value of a few bits to large lookup tables of several kilobytes. Because it is unlikely to find them in different algorithms, they can be used to identify cryptographic code. This is a widely used technique and it has been implemented in several publicly available tools, such as Findcrypt2 (IDA plugin) [34], KANAL (PEiD plugin), or H&C Detector, to name but a few. Apart from Findcrypt2 which has the advantage to be integrated within an interactive disassembler, we do not recommend to use any of them. As illustrated in [33], most of them are incomplete, target specific architecture or operating system and suffer from poorly usable interfaces. It should be noted that it is not necessary to disassemble and even to parse the executable file format in order to search for cryptographic constants.

As far as we know all tools based on constant identification solely rely on static analysis. However we should bear in mind that constant identification can also be performed using dynamic analysis. This is particularly useful in the case of packed programs, or for constants which are dynamically computed. We developed a proof of concept based on PIN [52] which instruments a binary program to monitor its memory access and to detect dynamically computed cryptographic constants. We run it on a simple program (that just encrypts few blocks of data using a table implementation of AES) packed with ASProtect [7], and on 7-Zip [34] (a well known compression tool that has cryptographic capabilities as part of the specifications of the different archive formats). In 7-Zip, the AES lookup tables are computed dynamically. As expected for both programs a static search failed to find the lookup tables, but our tool was able to detect them.

Constant identification is a very effective first step, but it is insufficient to precisely and completely uncover cryptographic primitives and modes of operation. We exhibit some of its limitations with the following example, that uses an AES table implementation as a target. Given a binary program, let us assume an AES substitution box has been detected by one of the previously listed tools. The precise location of the AES encryption/decryption routines still needs to be investigated. In fact multiple parts of the program can access the substitution box, such as: the AES key schedule (either for encryption or decryption) or a 4 kilobytes lookup tables generation routine. Moreover the parameters have not been identified. And last but not least, the detected algorithm could be another cryptographic primitive that uses the AES substitution box such as the Pelican Message Authentication Code (MAC) function [22], the Fugue hash function [36] or the LEX stream cipher [13]. Even though this last point is very unlikely in practice, this example is far from being only theoretical. For instance, a constant search on EasyLock [27] (a commercial solution to encrypt files on USB flash drives) reveals an AES substitution box. This substitution box is

accessed by two different functions but none of them is the encryption or the decryption routine, albeit AES is actually used to encrypt and decrypt files.

1.2 Cryptographic Parameters Retrieval

1.2.1 Plaintext Extraction from Running Programs

Let us consider a program which decrypts its input or encrypts its output. The line of work presented here provides solutions to automatically retrieve plaintext data from the memory of the program either before its encryption or after its decryption. This problem deserves to be mentioned here, since it requires to detect when and where the encryption / decryption algorithm takes place and to identify the output parameter (for a decryption) or one of the input parameters (for an encryption). The following works are all focused on the decryption problem.

Lutz [26] was the first to address this problem in the context of malware analysis. A Dynamic Binary Instrumentation (DBI) framework is used to perform dynamic data tainting (the taint source is the encrypted data) and to collect an execution trace containing the list of executed basic blocks and the list of tainted memory values. A dynamic Control Flow Graph (CFG) is first constructed out of the execution trace and this CFG is then exploited to recover loops (here a loop is defined as a back edge in the CFG). The decryption algorithm is supposed to be a loop which uses integer arithmetic and XOR operations and which decreases the entropy of tainted memory. Reformat [76] is a tool designed to automate protocol reverse engineering when messages are encrypted. It also uses data tainting to track encrypted data in memory, but solely relies on the ratio of arithmetic and bitwise instructions to detect the turning point between encrypted and decrypted data. Wang *et al.* [74] proposed an automatic mechanism to remove DRM protection of several popular streaming services. Their approach is driven by real-time constraints. Program executions are split according to the loop abstraction (here a loop is defined as a repetition of a sequence of basic blocks). For each loop, input and output memory buffers are reconstructed using relatively complex rules. Finally the decryption algorithm is characterized by a randomness decrease between and input and an output buffer. According to this work randomness (evaluated using the Chi-Square randomness test) should be preferred over entropy to distinguish encrypted data from compressed data.

To conclude, these three approaches seem relatively similar: they all rely on heavy dynamic instrumentation, use approximatively the same set of heuristics (loops, arithmetic and bitwise instructions and entropy / randomness) and do not require any knowledge on the decryption algorithm. But, apart from very basic location information, they provide almost no information about the cryptographic algorithm(s) themselves.

1.2.2 Key Extraction from Memory Dumps

A second line of work aims at extracting cryptographic keys from memory dumps. It addresses the following situation. One needs to obtain the key used by a running program to perform cryptographic operations but the only possible way to interact with the program is to dump its memory once. This situation is particularly important in the field of digital forensics. An investigator may have a physical access to a running system but also may lack credential to perform privileged actions on it. A possibility then, is to read either the physical memory or the storage device on which swap or hibernate files may be located. The techniques presented here, unlike those mentioned in the previous paragraph, depends on algorithms. They are also, for the most part, fast and accurate.

Symmetric Keys. Many block ciphers use a key schedule algorithm to expand a key into several round keys. For performance reasons, round keys are usually computed once for several executions of the block cipher. Due to their size and to their number, round keys are usually stored in a memory buffer during the different executions of the block cipher. As a consequence, a memory dump collected during an encryption or a decryption will contain the round keys. If it is possible to reverse part of the key schedule and obtain the key from a subset of round keys, then Algorithm 1 can be used to extract the key from any memory dump containing the round keys. The efficiency

of the method greatly depends on the difficulty to the reverse the key schedule: it can be extremely simple such as in AES (the key is equal to first round key(s)) or computationally intensive such as in Twofish.

Algorithm 1 Key Extraction

Notations: ks is the key schedule algorithm, $reverse\_ks$ is the reverse key schedule, s_r is the size of the round key buffer in bytes, m is a memory dump formalized as a sequence of bytes and $m[i:j]$ denotes the subsequence of m starting at offset i and ending at offset $j-1$.

for all offset i in $[0, lenght(m) - s_r]$ **do**
  $key \leftarrow reverse\_ks(m[i : i + s_r])$ ▷ we suppose that $m[i : i + s_r]$ is the round key buffer
  if $ks(key)$ is equal to $m[i : i + s_r]$ **then**
    return key
  end if
end for

This technique was first used in [35] for AES, DES and also to extract tweak values used in LRW and XTS. It was later extended to Serpent and Twofish [53]. Even though it may be of limited practical interest, we noticed that this technique can also be applied to SHA1, SHA-256 and SHA-512 in order to recover the last message block that was processed.

Asymmetric Keys. The format of RSA public and private keys is standardized as part of PKCS #1 [3]. RSA keys are defined as ASN.1 objects and they are encoded according to the DER encoding for storage and exchange. A possibility to efficiently detect RSA keys in a memory dump is to search for memory segments which satisfy the DER syntax and define the right type of ASN.1 object [35].

1.3 Binary Code Comparison

A second closely related area is binary code comparison. It is a more general problem but some of the techniques used might be of some interest for cryptographic algorithm identification. Binary code comparison is a vast field of research. It is itself related with malware analysis (the objective of which is to determine if a binary program is benign or malicious and in this latter case to determine if it belongs to a previously known family of malware) and source code comparison. We do not pretend to give here a comprehensive summary of all research works on binary code comparison. Instead we only present three different solutions to give a brief but limited overview of the kind of techniques which may be used in this field. We conclude this section by explaining why, in our opinion, dedicated solutions to identify cryptographic algorithms tends to produce better results than binary code comparison.

1.3.1 Examples

Sæbjørnsen *et al.* [65] proposed a solution to detect code clones in binary programs. At a high level their solution follows a classical approach. Programs are first divided into several pieces. A piece is either a function as in [43, 28] or simply a sliding window over the disassembled instructions as it is the case here. Each piece of code is characterized by a feature vector. To determine whether two code pieces are alike, their feature vector are compared. Binary code comparison methods usually differ in the composition of feature vectors and in the way feature vectors are compared. In the solution proposed by Sæbjørnsen *et al.*, a piece of code is characterized by bags of mnemonics, types of operands [1] and various combination of mnemonics and operand types. Bags are converted to feature vectors by counting the number of occurrences of each of their elements. And feature vectors are compared using the ℓ_1 distance.

[1] The type of an operand is either memory, register or immediate value.

In Rendezvous [43], features vectors are made of mnemonics n-grams, CFG subgraphs, and constants. A text search engine is used to compare feature vectors. To this end feature vectors are first converted to string expressions.

Instead of relying exclusively on static characteristics to compare pieces of code one can also take their dynamic behaviour into account. For instance in BLEX [28], two pieces of code are deemed similar if they exhibit similar behaviour while executed under the same controlled randomized environment. Feature vectors in BLEX are made of runtime values read to or written from memory, calls to dynamically linked libraries and system calls made during the execution. Features are set of values. To compare two feature vectors, the authors of BLEX compute a weighted arithmetic mean of the Jaccard index[2] of each of their features.

1.3.2 Relevance to Cryptographic Algorithm Identification

To identify the algorithm implemented in a piece of code, one can compare this piece of code with reference implementations of known algorithms. It is possible in order to increase the *recall*[3] to compare not only with one reference implementation per algorithm but with several.

Success of binary code comparison methods depends on the capacity of selected features to abstract characteristics of algorithms from implementation noise. This hazardous extraction has to be done twice: once per piece of binary code. It seems more reliable, in order to identify algorithms, to start with an abstract description of the algorithm rather than with a particular implementation. In the context of binary clone detection, the analyst is not necessarily aware of which algorithms are duplicated. Whereas in the context of algorithm identification, we can suppose that the analyst has access to an abstract description of the algorithms he wants to identify. In this context, it is suboptimal to reject an abstract description of the algorithm in favour of a reference implementation which will have to undergo a phase of abstraction anyway. This is the first argument against the usage of binary code comparison for algorithm identification.

In the case of cryptographic algorithms it seems interesting to devise specific identification method. In fact, symmetric cryptographic algorithms share common characteristics. For instance, they have very simple CFG, they do not call any sub-function (only true for primitives) or make any system call, they contain large amount of bitwise and arithmetic instructions and their operands are memory buffers. By taking these characteristics into account a dedicated identification method will have a better efficiency and will produce more accurate results. For instance it is ineffective to classify cryptographic primitives according to their CFG, the sub-functions they call or their count of XOR instructions. Yet these characteristics are extremely interesting to devise efficient front end filters to rapidly detect possible cryptographic algorithms in large programs.

To conclude, it is possible to use binary code comparison to identify algorithms, but it is more effective to devise an identification method which takes as input abstract description of algorithms. And creating dedicated solutions to identify cryptographic algorithms is interesting since they have pronounced characteristics.

1.4 Primitive Identification

In this section we present two existing techniques to identify cryptographic primitives. The first one is based on the unique relationship that exists between the input and output values of a cryptographic primitive. To some extent this technique may be considered as a specialized version of the dynamic approach used in BLEX [28]. The second technique relies on the avalanche effect of cryptographic functions.

[2]The Jaccard index measures the similarity of two finite sets. It is defined as the size of the intersection of the two sets divided by the size of their union.

[3]The recall is equal to the fraction of relevant instances returned by the identification method, over all relevant instances. It is used along with the *precision* to measure the relevance of an identification method. The precision is equal to the fraction of relevant instances among all instances returned by the identification method.

1.4.1 Input/Output Relation

Input/Output Relation (IOR) is a dynamic technique. It relies on the following principle: given a cryptographic primitive $f : \mathcal{I} \rightarrow \mathcal{O}$, the relation between an input value and the corresponding output value identifies f with an overwhelming probability. In other words, if during an execution, a sequence of instructions reads a value $i \in \mathcal{I}$ and writes a value $o \in \mathcal{O}$ such that $f(i) = o$ then we can conclude that this sequence implements f.

In its most basic form, this technique can be implemented using a debugger. Breakpoints are set manually before and after a given function. After hitting the breakpoints the parameters of the function are dumped and tested against known cryptographic primitive(s). However, due to the large number of functions to test and to the difficulty to manually find the right parameters, it makes sense to automate this technique to perform a more systematic search.

IOR on the Full Execution Trace.

Gröbert *et al.* [33] were the first to use IOR to automatically identify cryptographic primitives. Their main hypothesis is that the cryptographic parameters, at some point of the program execution, will be read from memory and written to memory. From an implementation perspective, the targeted program is executed in a DBI environment. The value and the address of every memory access are recorded in an execution trace. Because cryptographic parameters are generally larger than the word size of the architecture, they are accessed using several operations. To reconstruct possible cryptographic parameters, memory operands are regrouped according to the following rules:

- parameters are stored in continuous memory locations ;
- parameters are accessed by operations of equal size ;
- parameters are accessed either in ascending or in descending order ;
- operations used to access the same parameter are close to each other.

The first rule is always satisfied. The three others are used, when several accesses are made at the same address, to decide how to organize them: in one, or in several parameters. A detailed pseudo code of their parameter reconstruction algorithm is given in [32]. Although the authors published their implementation, we were unable to get it to work properly. None of our AES synthetic samples (refer to Section 4.3.1) was detected. It may be due to a misunderstanding from our side or to their implementation that might not be sufficiently mature.

IOR on Function Traces.

Zhao *et al.* [81, 80] also used IOR to automatically identify cryptographic primitives. But instead of running IOR on the whole execution trace, IOR is only executed on preselected trace segments. In their work, a trace segment corresponds to the execution of a function. Trace segments are selected based on three simple heuristics: they must contain specific constants (refer to Section 1.1.2), they must contain at least one XOR instruction and they must not call any sub-functions. The latter criterion comes from the fact that cryptographic primitives are usually implemented as a single function that does not rely on any sub-functions.

For each memory address that is accessed within a trace segment, the first value which is read (if there is no previous write) and the last value which is written are recorded. This way, two partial memory snapshots are reconstructed. The first one corresponds to the memory state before the execution of the trace segment. It contains only the memory locations that are read in the trace segment. The second one corresponds to the memory state after the execution of the trace segment. It contains only the memory locations that are written in the trace segment. A cryptographic parameter can be any continuous memory location of the right size that is either in the first snapshot (for an input parameter) or in the second snapshot (for an output parameter).

Experimental Results. We have implemented our own version of the identification scheme proposed by Zhao *et al.* This implementation is limited to the IOR part. We did not implement their segment selection heuristics. Therefore, in the following experiments, trace segments were

selected manually. The results we obtained on the set of synthetic samples are given in Table 1.1. The set of synthetic samples is presented in Section 4.3.1. Despite a few corner cases which are detailed as follows, primitive identification was successful for most synthetic samples.

The synthetic sample based on the AES implementation of Crypto++ was not correctly identified for a key of 192-bit and 256-bit. It comes from the fact that the endianness of the four first round keys is different from the endianness of the others. The reference implementation that was used to verify the IOR is not compatible with this peculiar round keys format. The synthetic sample based on the AES implementation of OpenSSL was not correctly identified. The function that implements the AES primitive is written in assembly language and does not use standardized calling conventions. Parameters are read from XMM registers. However, if the trace segment includes the wrapper function that reads the parameter from the memory and writes them in the XMM registers, the identification is successful. For every synthetic samples based on our own implementation of MD5, the identification failed. In our own implementation of MD5 the compression function is not implemented as a separate function but directly in the code of the hash function. Because we verified IOR using only a reference implementation of the compression function, we did not find any match. None of the synthetic samples based on RC4 was correctly identified. The message that is being encrypted by the synthetic samples is not sufficiently large to access the whole permutation. Thus, the initial permutation state, which is an input parameter for RC4, cannot be reconstructed. The SHA1 implementation given in RFC 3174 compiled with MSVC `-O2` was only partially identified. The compression function is inlined and we had to extract trace segments corresponding to the caller functions. One of these caller functions also pads the last message block. Consequently, only the unpadded message block was considered to be an input parameter and the last execution of the compression function was not identified. Finally, synthetic samples based on the XTEA implementation of Botan and TomCrypt were not correctly identified. It is due to specific round keys format. To conclude, we observed the following limitations:

- specific calling conventions in which parameters are passed to the callee through registers ;
- specific parameters format ;
- small parameters (for instance the two 8-bit index pointers of RC4). It increases the number of candidates and it makes a systematic search more difficult. It should be noted than in the results presented in [81] none of the RC4 test scenarios was correctly detected ;
- the totality of parameters has to be accessed in the recorded execution ;
- specific Application Programming Interface (API) such as in our own implementation of MD5.

However, the method described by Zhao *et al.* is simple and gives satisfying results in most of the cases. Because the analysis is limited to trace segments, the sets of input and output values can be determined precisely. Moreover, extracting candidates at the function granularity increases the chance of finding the parameters written in memory in a sensible format. In our opinion, this method has the best performance over complexity ratio.

IOR on Loop Traces.

More recently, Calvet *et al.* [16] proposed a complete and detailed tool named Aligot based on IOR. According to its authors the main contribution of Aligot is to deal with obfuscated programs. In this context, the segment selection method proposed by Zhao *et al.* is no longer applicable. In fact, constants can be easily hidden or even modified, junk code can be inserted to disturb heuristics based on mnemonics and function boundaries can be hard to detect (API obfuscation).

Instead, Aligot relies on loop abstraction to extract trace segments. Symmetric cryptographic primitives usually execute a round function multiple times. Therefore, a repetition of a portion of the code indicates a possible implementation of a cryptographic primitive. Notice that loops of primitives should not be mistaken with loops of modes of operation. To resist obfuscation, Aligot uses its own loop definition which is a slight modification of the definition proposed by Kobayashi [44]. According to Kobayashi, a loop is defined as a repetition of a sequence of dynamic instructions. The authors of Aligot claim that this dynamic loop definition is well suited to deal with obfuscated code. In fact, unlike the usual loop definition used in static analysis (a loop is defined as an edge of the CFG that goes from a vertex to one of its dominant), their definition

Table 1.1: Results obtained on the set of synthetic samples with our own implementation of the identification method described by Zhao *et al.*

Primitive	Source	Compiler	Opt.	Result
AES-128 AES-192 AES-256	Gladman V_0	*	*	ok
	Gladman V_1	*	*	ok
	Gladman V_2	*	*	ok
	Gladman V_3	*	*	ok
	Botan	-	-	ok
	Crypto++	-	-	AES-128: ok, AES-192,256: nok
	Nettle	-	-	ok
	OpenSSL	-	-	nok
	TomCrypt	-	-	ok
MD5 Compress	Own Source	*	*	nok
	Botan	-	-	ok
	Crypto++	-	-	ok
	Nettle	-	-	ok
	OpenSSL	-	-	ok
	TomCrypt	-	-	ok
RC4	*	*	*	nok
SHA1 Compress	RFC 3174	$GCC_{4.9.2}$	*	ok
		$Clang_{3.5.0}$	*	ok
		$MSVC_{18.0}$	`-O0`	ok
			`-O2`	∼ 1 execution out of 2 was detected
	Botan	-	-	ok
	Crypto++	-	-	ok
	Nettle	-	-	ok
	OpenSSL	-	-	ok
	TomCrypt	-	-	ok
XTEA	Wikipedia	*	*	ok
	Botan	-	-	nok
	Crypto++	-	-	ok
	TomCrypt	-	-	nok

does not consider control flow flattening[4] as a loop. Moreover, the authors claim that their loop definition is also able to detect unrolled loops.

Unrolled Loops. Loop unrolling is a common technique used by software developers and compilers to reduce the number of instructions that are required to control loops, to minimize branch penalties and to increase instruction level parallelism. Unrolled loops are common in cryptographic implementations as performance is a major concern. For instance, in most implementations of MD5 and SHA1, the Feistel network is fully unrolled. But in our opinion and according to our experiments, the loop definition used by Aligot is unable to cope with unrolled loops. This statement is illustrated by the example of Figure 1.1. This figure is an assembly code sample taken from and unrolled implementation of MD5. It contains the 17th and 18th step. The same boolean function is used by these two steps but nevertheless they were compiled to different sequences of instructions. Loop unrolling is not the final stage of the compilation process. Loops may even be unrolled directly in the source code as it was the case for the example of Figure 1.1. Thus, further optimizations passes will continue to modify the code after it. For instance, in Figure 1.1 a common subexpression has been eliminated between the 16th and 18th step. Instruction scheduling and instruction selection are different in the 17th and 18th step. And finally, several early memory

[4]Control flow flattening is an obfuscation technique that modifies the CFG to reduce its readability [45]. All transitions between basic blocks are regrouped into a single switch statement called a dispatcher. An additional variable is used by the dispatcher to determine which basic block should be executed next. The CFG is flattened in the sense that all basic blocks are at the same level. Their initial hierarchy is now hidden in the dispatcher.

loads mix the code of the different steps. To conclude, if loops are unrolled in the source code or during compilation, there is no guarantee that the different repetitions of the loop body will result in the exact same sequence of instructions. In practice it is almost never the case for the first and the last iteration due to side effects.

Parameter Reconstruction. But the most serious problem with the method described by Calvet *et al.* comes from parameter reconstruction. For trace segments based on functions (Zhao *et al.*), parameter reconstruction is relatively easy. Parameters are stored in memory and their format is intelligible (at least by the software developer). But for trace segments based on loops, none of that is true any more. Parts of parameters may have been loaded into registers prior to the loop. They may also have been copied into separate local variables the organization of which on the stack does not abide by any particular rule. To address this problem, Calvet *et al.* proposed to include in the execution trace, the values that are read from registers and written to registers. Unfortunately the set of rules that is given to reconstruct parameters from the list of memory and register values, is incorrect and incomplete. It includes two rules based on spatial proximity in both code and memory. Spatial proximity in code, means that an instruction in a loop body manipulates always the same parameter during all its executions. Spatial proximity in memory, means that non-adjacent memory locations cannot belong to the same parameter. As previously mentioned, a parameter may have been copied into several local variables before the loop execution. If these local variables are declared as separate variables and not as an array, they might be mixed with others local variables on the stack. In that case, the second rule will miss this parameter. This scenario is illustrated by the following code snippet taken from the AES source code of Nettle [48]. The plaintext buffer `src` is read in little-endian mode (`LE_READ_UINT32`) and then it is XORed with the first set of round keys. The result which is one of the input parameters of the loop, is stored in four separate 32-bit variables.

```
uint32_t w0, w1, w2, w3;

/* [...] */

w0 = LE_READ_UINT32(src)      ^ keys[0];
w1 = LE_READ_UINT32(src + 4)  ^ keys[1];
w2 = LE_READ_UINT32(src + 8)  ^ keys[2];
w3 = LE_READ_UINT32(src + 12) ^ keys[3];

for (i = 1; i < rounds; i++){
        /* [...] */
}
```

Moreover, nothing was proposed to regroup register values together and with memory values. Consequently, if we assume that a parameter may be scattered throughout memory and registers, we must resort to a brute force approach to find it. Given n memory or register values, the number of parameters resulting from every combination and permutation of k values, is $\binom{n}{k}k!$ This is not tractable in practice. Note that this problem only affects input parameters. In fact output parameters can be reconstructed a posteriori. Once input parameters are recovered, the corresponding output parameters can be computed using a reference implementation. Finally, one simply needs to determine if every part of the output parameters is within the list of values that are written to memory and registers.

Experimental Results. The authors of Aligot published their implementation. However, it suffers from very poor performances. For none of our AES synthetic samples (refer to Section 4.3.1) we obtained a result during the first hour of computation. We did not continue the execution after this time limit. We have implemented our own version of Aligot. Parameter reconstruction is done as follows. First memory values of equal size which form a continuous memory segment are regrouped. These memory segments are then combined with the register values using a brute force approach. To limit the combinatorial complexity, register values are not inserted inside

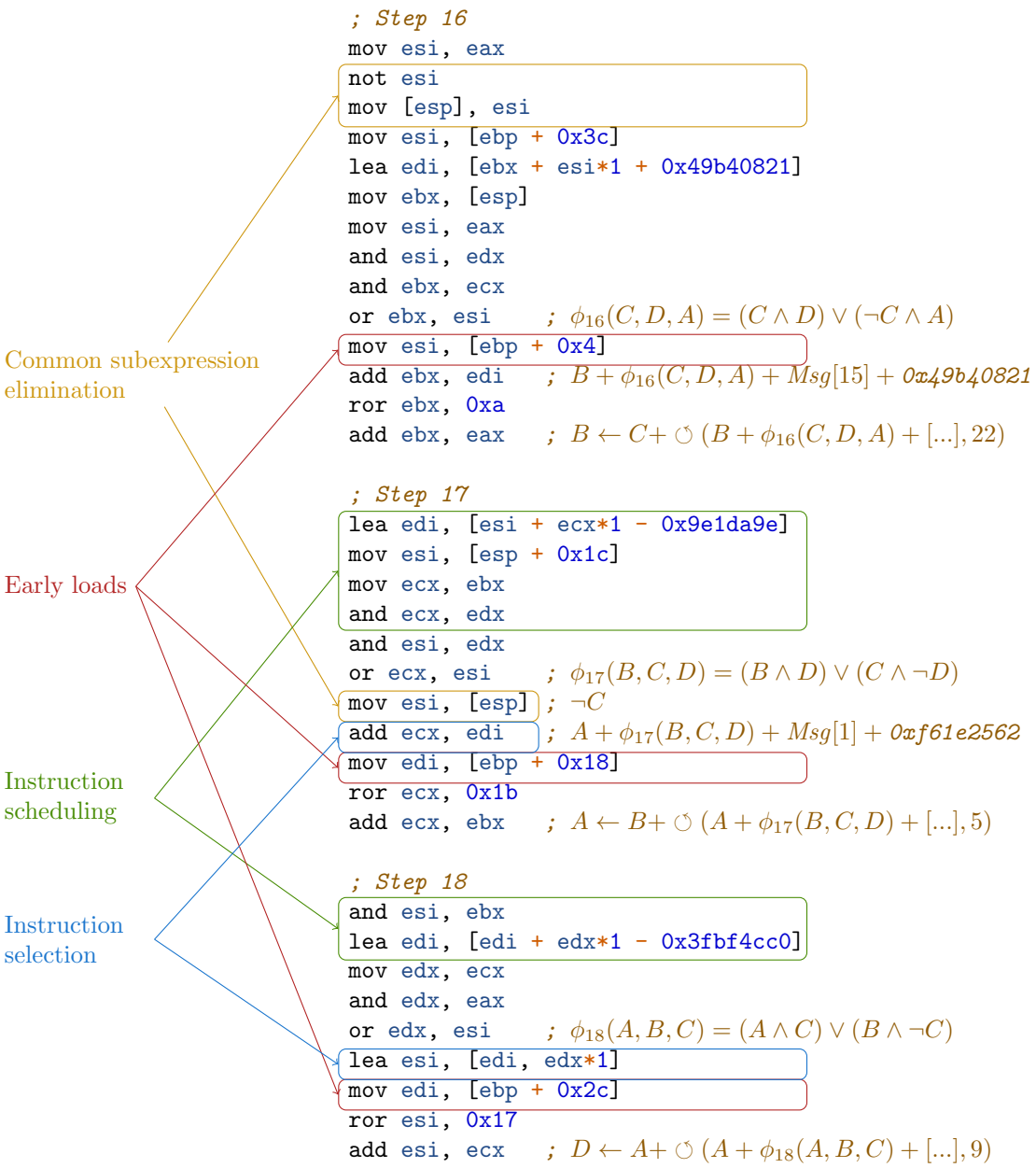

Figure 1.1: Assembly code sample taken from an unrolled implementation of MD5. This sample contains the 16th, 17th and 18th step. Even though the 17th and 18th step realize the same computation, the compiler returned different sequences of instructions for each of them. Consequently, they are not detected as a loop body by Aligot. Some of the mechanisms of the compiler that were responsible for the differences between the two sequences are highlighted in the figure.

memory segments. We only consider the cases where they are appended at the beginning or at the end of memory segments. We do not combine memory segments together neither. Finally, a parameter can be any frame of the right size inside these groups of aggregated values. Although our implementation is not an exact replica of Aligot (the rules are a little bit different), it should have essentially the same limitations.

The results we obtained on the set of synthetic samples are given in Table 1.2. Some of the synthetic samples were removed because they contain MMX and SSE instructions which are not supported by our implementation. The RC4 and MD5 synthetic samples were not explicitly tested and are only mentioned for completeness. As previously stated, the message that is being encrypted by the RC4 synthetic samples is not sufficiently large to access the entire permutation. Thus, there is no way the permutation can be reconstructed. In every MD5 synthetic samples, the loop of the Feistel network is unrolled directly in the source code. Thus, it is very unlikely that the loop can be detected[5]. Note that for MD5 and SHA1, we could have used the loop over the message blocks to identify either the compression function (one execution of the loop body) or directly the full hash function. But, since this loop is more related with the mode of operation than with the primitive itself (which we consider to be the compression function in our experiments) we did not do it. For short messages the loop over the message blocks will contain a single execution of the loop body. Depending on how the message chunks are passed to the hash function API, the loop over the messages blocks may not even be detected by Aligot.

Because trace segments are extracted based on the loop abstraction, we must limit reference implementations which are used to verify IOR to the part of primitives which are executed inside a loop. The set of reference implementations, also called verifiers, that we used is as follows:

AES There is no *MixColumns* in the last round of AES. Because of this difference, the last round is not executed as part of the main loop. To deal with the different keys sizes, we used a 9, 11 and 13 round verifier. A classical AES optimization consists in manually unrolling the loop by a factor of two (an iteration computes two rounds). Therefore, we also used a 8, 10 and 12 round verifier.

SHA1 We used a verifier for the key schedule (noted ks) and a verifier per groups of step functions. There are four groups of step functions. The verifiers are respectively noted $h0, h1, h2$ and $h3$. For a perfect detection, we must find match for the five verifiers.

XTEA We used a single verifier that covers the 32 cycles and the key schedule.

Table 1.2 only mentions the name of the verifiers that were found. The loop of the primitive is fully unrolled in the following synthetic samples: Gladman V_0, SHA1 Botan, SHA1 Crypto++ and SHA1 Nettle. We can see that for all these samples no verifier was found. For several synthetic samples (Gladman V_1 for instance) the one hour time limit was reached before any result was returned. It may be due to a large number of loops that need to be processed. This is the case for the synthetic samples based on Botan. Because this library performs a lot of processing during its initialisation, many more loops are detected in its execution trace than for any other synthetic samples (more than 1800 for AES Botan compared to 35 for AES OpenSSL for instance). But it is also due to the high combinatorial complexity of parameter reconstruction combined with the large number of verifiers (six only for AES, but that number must be doubled if we also want to detect big endian implementations such as in TomCrypt). For the synthetic samples based on AES and SHA1 less than half of the expected verifiers were actually found. These results are significantly worse than those obtained with the method of Zhao *et al.* presented in Table 1.1[6]. For the synthetic samples based on XTEA, the detection rate is higher. This is not surprising because the size of the message block is small (64 bits). Fewer operands need to be combined to reconstruct the input message block.

[5]Unrolled loops detection is illustrated with other tests such as Gladman V_0 and SHA1 synthetic samples.

[6]However, we should bear in mind while comparing theses two sets of results, that for the Table 1.1 the trace segments were manually selected, whereas in Table 1.2 we rely on the loop detection to select trace segments.

Table 1.2: Results obtained on the set of synthetic samples with our own implementation of the identification method described by Calvet *et al.* Execution times were measured on a i5-3320M processor.

Primitive	Source	Compiler	Opt.	Result	Time
AES-128 AES-192 AES-256	Gladman V_0	$GCC_{4.9.2}$	-O0	∅	1 min 39 s
			-O1	∅	> 1 h
			-O2	∅	> 1 h
			-O3	∅	> 1 h
		$Clang_{3.5.0}$	-O0	∅	54 min 23 s
		$MSVC_{18.0}$	-O0	∅	> 1 h
			-O2	∅	> 1 h
	Gladman V_1	$GCC_{4.9.2}$	-O0	8 rounds	1 min 36 s
			-O1	∅	> 1 h
			-O2	∅	> 1 h
			-O3	∅	> 1 h
		$Clang_{3.5.0}$	-O0	∅	54 min 39 s
		$MSVC_{18.0}$	-O0	∅	> 1 h
			-O2	∅	> 1 h
	Gladman V_2	$GCC_{4.9.2}$	-O0	9, 11, 13 rounds	1 min 39 s
			-O1	9, 11, 13 rounds	1 min 30 s
			-O2	9, 11, 13 rounds	1 min 29 s
			-O3	9, 11, 13 rounds	1 min 29 s
		$Clang_{3.5.0}$	-O0	∅	52 min 9 s
		$MSVC_{18.0}$	-O0	∅	> 1 h
			-O2	∅	> 1 h
	Gladman V_3	$GCC_{4.9.2}$	-O0	∅	3 min 20 s
			-O1	∅	2 min 28 s
			-O2	∅	2 min 17 s
			-O3	∅	2 min 19 s
		$Clang_{3.5.0}$	-O0	9 rounds	2 min 13 s
		$MSVC_{18.0}$	-O0	9 rounds	5 min 18 s
			-O2	∅	19 min 35 s
	Botan	-	-	∅	> 1 h
	Nettle	-	-	∅	1 min 47 s
	TomCrypt	-	-	∅	7 min 33 s
MD5	*	*	*	∅	-
RC4	*	*	*	∅	-
SHA1	RFC 3174	$GCC_{4.9.2}$	-O0	ks	12 s
			-O1	∅	> 1 h
			-O2	h0, h1, h2, h3	1 min 32 s
			-O3	h0, h1, h2, h3	5 min 32 s
		$Clang_{3.5.0}$	-O0	ks	2 s
			-O1	ks, b0	8 min 1 s
			-O2	ks, b0	7 min 58 s
			-O3	ks, b0	7 min 53 s
		$MSVC_{18.0}$	-O0	ks	5 min 26 s
			-O2	∅	> 1 h
	Botan	-	-	∅	> 1 h
	Crypto++	-	-	∅	55 min 16 s
	Nettle	-	-	∅	33 min 42 s
	TomCrypt	-	-	b0	14 min 1 s
XTEA	Wikipedia	$GCC_{4.9.2}$	-O0	∅	41.2 s
			-O1	32 cycles	45.2 s
			-O2	32 cycles	45.6 s
			-O3	32 cycles	45.2 s

Primitive	Source	Compiler	Opt.	Result	Time
XTEA	Wikipedia	$\text{Clang}_{3.5.0}$	-O0	∅	0.4 s
			-O1	32 cycles	1.6 s
			-O2	32 cycles	1.6 s
			-O3	32 cycles	1.6 s
		$\text{MSVC}_{18.0}$	-O0	∅	6 min 20 s
			-O2	32 cycles	6 min 20 s
	Botan	-	-	∅	> 1 h
	Crypto++	-	-	32 cycles	11.8 s
	TomCrypt	-	-	∅	29.2 s

Conclusion

To summarize, we have seen three different methods based on Input/Output Relation. The first method was proposed by Gröbert *et al.* The execution trace is analysed as a whole. Because data stored at a given address may change during the execution of the program, memory buffers must be reconstructed based on spatial proximity in memory and also temporal proximity in the execution trace. The implementation published by the authors gave no positive result on our set of synthetic samples. The second method was proposed by Zhao *et al.* Trace segments are extracted based on the function abstraction. Parameter reconstruction relies on two memory snapshots: one before and one after the execution of the trace segment. The evaluation conducted on our own implementation, gave good results. The third method was proposed by Calvet *et al.* Trace segments are extracted based on the loop abstraction. Parameters must be reconstructed from memory and register values. The evaluation conducted on our own implementation gave poor results for both loop detection in case of unrolled loops and parameter reconstruction. We do not deny that for obfuscated programs, the method presented by Calvet *et al.* may outperform the others. Still, for regular programs we would rather recommend the method of Zhao *et al.*

1.4.2 Avalanche Effect

A transformation from n bits to m bits satisfies the strict avalanche criterion if for any randomly chosen input and any $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$ when the i^{th} input bit is flipped there is a 50% probability that the j^{th} output bit will also be flipped [77]. Intuitively it means that each output bit depends on all the input bits. From a cryptographic point of view, it is a highly desirable property. We assume that it is always satisfied by cryptographic algorithms but almost never satisfied by non-cryptographic algorithms. Thus, to identify cryptographic primitives, one could search for sets of variables which verify the avalanche property. The main advantage of this method is that it does not require any a priori knowledge of the primitive. As long as the primitive satisfies the avalanche property, it can be detected. But as such, this method is not capable of revealing the identity of the primitive. It can only return the location of the primitive and of its parameters.

The traditional way to implement this method is to use memory tainting. Data which is supposed to be processed by a cryptographic primitive is tainted. A different taint value is associated to each byte (or bit, depending on the granularity). If at some point during the program execution there is a set of variables which are tainted with all the different taint values, these variables might be the output of a cryptographic primitive. Note that this condition is necessary but not sufficient to satisfy the strict avalanche criterion. Such an implementation was used in TaintScope [75] to detect hash functions and checksums, and later in CipherXRay [47] to identify cryptographic primitives and modes of operation. In the remainder of this section, we detail the solution proposed by CipherXRay and give some experimental results.

Two additional assumptions are made in CipherXRay. First, cryptographic parameters are contained at some point of the program execution in continuous memory buffers. Thus, one does not need to check the avalanche property for any set of intermediate variables but only for memory buffers. Second, cryptographic primitives are implemented in their own dedicated functions. Thus one can limit the search to the input and output of functions. These assumptions are very similar to those made by Zhao *et al.* in Section 1.4.1. Consequently, for known primitives, CipherXRay is

going to have at least the same limitations as the method of Zhao *et al.* (results presented in Table 1.1). CipherXRay only returns the largest memory buffers which satisfy the avalanche property. CipherXRay does not consider implicit dependencies. Even though we did not test it, the authors of CipherXRay claim that their solution is also able to detect modes of operation and asymmetric cryptography.

Experimental Results. Since the authors did not release their implementation, we implemented our own version of CipherXRay. Instead of performing live memory tainting, using Valgrind for instance as suggested by the authors, we perform the complete analysis based on execution traces. We did it to reuse part of the framework that was developed for our own primitive identification solution. Starting from an execution trace segment containing a sequence of dynamic instructions and runtime address values, we build the corresponding DFG (DFGs are detailed in the following chapter). Then we search for any two sets of vertices such that:

- the first set is made of memory read operations which access a continuous memory buffer ;
- the second set is made of memory write operations which also access a continuous memory buffer and which are reachable from all the vertices of the first set ;
- they are not strictly included in sets which also satisfy the previous two points. Thereby we only consider local maximums.

By searching for paths in DFGs, we dot not precisely track dependencies at the bit or byte granularity. Instead for each operation, that is to say for each vertex, we consider that every output bit depends on all input bits regardless of the exact computation performed by the operation. Thanks to runtime address values it is easy check whether or not a set of memory operations accesses a continuous memory buffer. To accurately track data flow as variables are written and read from memory our initial idea was to reuse some form of memory access simplification based on runtime address values such as presented in Section 5.3.1. But it was inappropriate. In fact, according to memory access simplification, if two read operations access the same memory location, the result of the second read is replaced by the result of the first read. And as a consequence, the dependency between the address of the second read operation and its result is deleted. This problem happens for instance for the AES lookup tables. To solve this issue we adopted a more traditional memory tainting approach for memory operations:

- a memory write is tainted if either its address operand or its value operand is tainted ;
- a memory read is tainted if its address operand is tainted or if the last memory write occurring at the same address is tainted.

The results we obtained on the set of synthetic samples are given in Table 1.4. We ran the analysis on the exact same trace segments as in Table 1.1 (IOR on function traces). To remove some obvious false positives and to return a primitive name, we filter sets of memory buffers subject to the avalanche property based on their size. For instance, given a primitive $\mathcal{P}$ with two input parameters of size s_1 and s_2 and one output parameter of size s_3, we consider that $\mathcal{P}$ was detected if we found two couples of memory buffers subject to the avalanche property (x_1, y_1) and (x_2, y_2) such that $y_1 = y_2$, the size of x_1 is equal to s_1, the size of x_2 is equal to s_2 and the size of y_1 is equal to s_3. For each primitive involved in the set of synthetic samples, its number of parameters as well as their respective size are given in Table 1.3. To determine whether a detection is a false positive or not, we run a countercheck using IOR. The number of detections that were successfully verified using IOR is given on the right of oblique strokes in Table 1.4. A first observation is that many primitives were missed because parameters returned by our implementation of CipherXRay are larger than the real parameters. If a real input parameter is stored next to a variable which also has an influence on every part of the output then it is impossible to separate the real parameter from the other variable. It happens for instance with our own implementation of MD5, with the MD5 implementation of Botan and with the SHA1 implementation of RFC 3174. A second observation is that there are many false positives. This is not utterly surprising. For instance, if we consider AES, which is the primitive with the highest rate of false positive, every time four adjacent locations are accessed in a lookup table it creates a false positive. In fact, it forms a 128-bit memory buffer, which verifies the avalanche property with the output and therefore can

Table 1.3: Bits affected by the avalanche property for primitives involved in the set of synthetic samples.

Primitive	Prototype
AES-128 (key schedule not included)	$\{0,1\}^{128} \times \{0,1\}^{1280} \rightarrow \{0,1\}^{128}$
AES-192 (key schedule not included)	$\{0,1\}^{128} \times \{0,1\}^{1536} \rightarrow \{0,1\}^{128}$
AES-256 (key schedule not included)	$\{0,1\}^{128} \times \{0,1\}^{1792} \rightarrow \{0,1\}^{128}$
MD5 (compression function)	$\{0,1\}^{128} \times \{0,1\}^{512} \rightarrow \{0,1\}^{128}$
SHA1 (compression function)	$\{0,1\}^{160} \times \{0,1\}^{512} \rightarrow \{0,1\}^{160}$
XTEA (key schedule included)	$\{0,1\}^{64} \times \{0,1\}^{128} \rightarrow \{0,1\}^{64}$

be mistaken with the message block. It happens for AES-192 which in all synthetic samples has exactly two more false positives than AES-128 and AES-256. Note that only the implementation changes from one synthetic samples to another, not the value of the parameters.

As a conclusion, the avalanche property provides a generic criterion to detect cryptographic primitives. Compared to IOR, it does not require any a priori knowledge of primitives. But for known primitives, it is less accurate than IOR regarding both the false positive and the false negative ratio. It should be noted that detection methods based on the avalanche property need only runtime address values whereas IOR also needs runtime memory values. We did not evaluate the avalanche criterion to detect modes of operation as described in CipherXRay. It may help to reduce the number of false positives. For instance, the key is always the same for every execution of a block cipher within a mode of operation and message blocks are likely to be adjacent in memory. But it implies that one has to create signatures for modes of operation which consist in influence patterns. And as such, it contradicts the simplicity and the genericity of the original idea.

Table 1.4: Results obtained on the set of synthetic samples with our own implementation of CipherXRay. For each primitive which was detected, two numbers are given. They are separated by an oblique stroke. On the left is the number of solutions returned by our own implementation of CipherXRay, and on the right is the number of these solutions which also satisfy the IOR criterion. Depending on whether these numbers are below, equal or above the expected value, they are printed in red, green or orange respectively.

Primitive	Source	Compiler	Opt.	Result
AES-128 AES-192 AES-256 (encryption only)	Gladman V_0	$GCC_{4.9.2}$	-O0	128: 1/1, 192: 3/1, 256: 1/1
			-O1	128: 2/1, 192: 4/1, 256: 2/1, XTEA: 5/0
			-O2	128: 1/1, 192: 3/1, 256: 1/1, XTEA: 84/0
			-O3	128: 1/1, 192: 3/1, 256: 1/1, XTEA: 84/0
		$Clang_{3.5.0}$	-O0	128: 1/1, 192: 3/1, 256: 1/1
			-O1	128: 1/1, 192: 3/1, 256: 1/1
			-O2	128: 1/1, 192: 3/1, 256: 1/1, MD5: 4/0
			-O3	128: 1/1, 192: 3/1, 256: 1/1, MD5: 4/0
		$MSVC_{18.0}$	*	128: 1/1, 192: 3/1, 256: 1/1
	Gladman V_1	$GCC_{4.9.2}$	*	128: 1/1, 192: 3/1, 256: 1/1
		$Clang_{3.5.0}$	-O0	128: 1/1, 192: 3/1, 256: 1/1
			-O1	128: 1/1, 192: 3/1, 256: 1/1, XTEA: 84/0
			-O2	128: 2/1, 192: 4/1, 256: 2/1, XTEA: 136/0
			-O3	128: 2/1, 192: 4/1, 256: 2/1, XTEA: 136/0
		$MSVC_{18.0}$	*	128: 1/1, 192: 3/1, 256: 1/1
	Gladman V_2	$GCC_{4.9.2}$	*	128: 1/1, 192: 3/1, 256: 1/1
		$Clang_{3.5.0}$	-O0	128: 1/1, 192: 3/1, 256: 1/1
			-O1	128: 3/1, 192: 5/1, 256: 3/1
			-O2	128: 2/1, 192: 4/1, 256: 2/1
			-O3	128: 2/1, 192: 4/1, 256: 2/1
		$MSVC_{18.0}$	-O0	128: 1/1, 192: 3/1, 256: 1/1
			-O2	128: 0/0, 192: 3/1, 256: 1/1
	Gladman V_3	$GCC_{4.9.2}$	-O0	128: 4/1, 192: 6/1, 256: 4/1
			-O1	128: 5/1, 192: 7/1, 256: 5/1, XTEA: 2/0
			-O2	128: 4/1, 192: 7/1, 256: 5/1, XTEA: 214/0
			-O3	128: 4/1, 192: 7/1, 256: 5/1, XTEA: 214/0
		$Clang_{3.5.0}$	-O0	128: 4/1, 192: 6/1, 256: 4/1
			-O1	128: 4/1, 192: 6/1, 256: 4/1
			-O2	128: 5/1, 192: 7/1, 256: 5/1
			-O3	128: 5/1, 192: 7/1, 256: 5/1
		$MSVC_{18.0}$	-O0	128: 4/1, 192: 6/1, 256: 4/1
			-O2	128: 0/0, 192: 6/1, 256: 4/1
	Botan	-	-	128: 0/0, 192: 0/0, 256: 0/0
	Crypto++	-	-	128: 6/1, 192: 6/0, 256: 3/0
	Nettle	-	-	128: 0/0, 192: 2/0, 256: 0/0
	OpenSSL	-	-	128: 0/0, 192: 0/0, 256: 0/0
	TomCrypt	-	-	128: 1/1, 192: 3/1, 256: 1/1
MD5	Own Src.	*	*	0/0
	Botan	-	-	0/0
	Crypto++	-	-	2/2
	Nettle	-	-	2/2
	OpenSSL	-	-	2/2
	TomCrypt	-	-	2/2
RC4	*	*	*	not tested
SHA1	RFC 3174	*	*	0/0
	Botan	-	-	1/1
	Crypto++	-	-	2/2
	Nettle	-	-	2/2

Primitive	Source	Compiler	Opt.	Result
SHA1	OpenSSL	-	-	4/2
	TomCrypt	-	-	2/2
XTEA (encryption only)	Wikipedia	$GCC_{4.9.2}$	-O0	3/1
			-O1	1/1
			-O2	1/1
			-O3	1/1
		$Clang_{3.5.0}$	-O0	4/1
			-O1	1/1
			-O2	1/1
			-O3	1/1
		$MSVC_{18.0}$	-O0	0/0
			-O2	0/0
	Botan	-	-	0/0
	Crypto++	-	-	2/1
	TomCrypt	-	-	0/0

Chapter 2

Data Flow Graph

In order to modify, compare and visualize assembly code, we use a graph representation called a DFG. In this chapter, we first give a definition of a DFG. Then, we explain how a DFG representation can be computed out of a sequence of assembly instructions. We finally describe some built-in features which facilitate DFG manipulation.

Contents

2.1 DFG Definition 33
2.1.1 Formal Definition 33
2.1.2 Memory Operations 36
2.1.3 Practical Aspects 36
2.2 Sequence of Dynamic Instructions 37
2.2.1 Straight-Line Code Hypothesis 37
2.2.2 Execution Trace 39
2.2.3 Segment Selection 39
2.3 Construction 40
2.3.1 Mapping x86 Instructions to DFGs 40
2.3.2 SIMD Instructions 40
2.3.3 DFG Composition 41
2.4 Built-in Mechanisms 43
2.4.1 Dead Code Removal 43
2.4.2 Value Range Analysis 44

2.1 DFG Definition

The definition of a DFG given below is heavily inspired on the literature on rewrite systems and on term graphs (in particular [24] and [60]). In fact, we show in Chapter 3 that many normalization mechanisms can be formalized as term rewrite rules and that the set of all normalization mechanisms forms a reduction system. This formal definition is followed by more practical aspects that were introduced to specifically represent x86 assembly code.

2.1.1 Formal Definition

Terms. Let Σ be a set of operation symbols and X be a set of variable symbols such that $\Sigma \cap X = \emptyset$. The arity function defines the number of arguments an operation symbol takes. Operation symbols with arity 0 are called constant. To extend the arity function to $\Sigma \cup X$, we define $\text{arity}(x) = 0$ for all $x \in X$. A term or an expression over Σ and X is a variable symbol, a constant, or a string of the form $f(t_1, ..., t_n)$ where f is an operation symbol of arity n and $t_1, ..., t_n$ are terms. We note $T_{\Sigma,X}$ the set of all terms over Σ and X. The subterms of a term t are t plus, if t is a composite term such as $t = f(t_1, ..., t_n)$, all the subterms of $t_1, ..., t_n$. Terms will be written

with the prefix notation even for symbols that represent basic arithmetic or logical operations. This way, it will be possible to differentiate terms from others mathematical expressions.

We assume there is a set D such that every operation f, the symbol of which is in Σ, is a mapping from D^n to D (with $n = \text{arity}(f)$). We define an assignment as a mapping from X to D. Given an assignment θ, we can evaluate every term of $T_{\Sigma,X}$ to an element of D (composite terms are evaluated by applying operations to the evaluation of their operands). By extension, the value of a term $t \in T_{\Sigma,X}$ under an assignment θ, is called $\theta(t)$ and θ is considered to be a mapping from $T_{\Sigma,X}$ to D.

Terms have a natural ordered tree structure. Leaves are labelled with variables symbols or constants and internal vertices are labelled with operation symbols of strictly positive arity. The indegree of the internal vertices is equal to the arity of their label. However, tree representations can be extremely costly in both time and space: each instance of a subterm has to be stored and processed separately. When working with terms that contain numerous common subterms, it is more efficient to adopt a graph representation where they can be shared.

Directed Multigraphs. A directed multigraph G is a 4-tuple $G = (V_G, E_G, src_G, dst_G)$ where V_G is a set of vertices, E_G is a set of edges, $src_G \colon E_G \to V_G$ assigns to each edge its source vertex and $dst_G \colon E_G \to V_G$ assigns to each edge its destination vertex. Unlike graphs, multigraphs can have parallel edges, that is to say, edges which have the same source and destination vertices. Multigraphs are well suited to represent terms, since a subterm may appear several times in a composite term resulting in parallel edges. In the sequel, directed multigraphs are simply called graphs unless explicitly stated otherwise.

Given an edge e in a graph G, $src_G(e)$ is said to be a direct predecessor of $dst_G(e)$ and conversely $dst_G(e)$ is said to be a direct successor of $src_G(e)$. Given a vertex v in G, $\text{indegree}_G(v)$ and $\text{outdegree}_G(v)$ denote the number of direct predecessors of v and the number of direct successors of v respectively. Given two vertices u and v in G, a path from u to v is a sequence of edges $(e_i)_{1 \leq i \leq n}$ such that $src_G(e_1) = u$, $dst_G(e_n) = v$ and $dst_G(e_i) = src_G(e_{i+1})$ for all $i \in \{1, ..., n-1\}$. If there is a path from u to v, we say that v is reachable from u. The successors of v are the vertices that are reachable from v and the predecessors of v are the vertices from which v is reachable. A graph is acyclic if there is no vertex that is reachable from itself via a non-empty path. Given two graphs G and H, H is a subgraph of G if $V_H \subset V_G$, $E_H \subset E_G$ and $\forall e \in E_H, src_H(e) = src_G(e)$ and $dst_H(e) = dst_G(e)$.

Definition 1 (DFG)**.** A Data Flow Graph G is a 6-tuple $(V_G, E_G, src_G, dst_G, labV_G, labE_G)$, where:

- (V_G, E_G, src_G, dst_G) is an acyclic directed multigraph ;
- $labV_G \colon V_G \to \Sigma \cup X$ is a vertex labelling function which maps vertices to variable symbols and operation symbols such that $\text{indegree}_G(v) = \text{arity}(labV_G(v))$ for all $v \in V_G$;
- $labE_G \colon E_G \to \mathbb{N}$ is an edge labelling function which maps edges to positive integers. Two distinct edges $e_1, e_2 \in E_G$ that have the same destination vertex $dst_G(e_1) = dst_G(e_2)$ have distinct labels $labE_G(e_1) \neq labE_G(e_2)$. This ensures that the edge labelling function $labE_G$ defines a total order relation over any set of edges associated to a given destination vertex. To guarantee the uniqueness of the edge labelling function $labE_G$, we also impose that for every edge e: $labE_G(e) < \text{indegree}_G(dst_G(e))$.

The function $term_G \colon V_G \to T_{\Sigma,X}$ maps every vertex of G to a term of $T_{\Sigma,X}$. It is defined as follows:

$$term_G(v) = \begin{cases} labV_G(v), & \text{if } \text{indegree}_G(v) = 0 \\ labV_G(v)(term_G(v_1), ..., term_G(v_n)), & \text{if } \text{indegree}_G(v) = n \end{cases}$$

where $v_1, ..., v_n$ are the direct predecessors of v. These vertices are ordered according to the ordering of the edges $e_1, ..., e_n$ that connects them to v (*i.e.* according to $labE_G \circ src_G$). Note that this recursive definition is correct since DFGs are acyclic.

Many properties can be proven on DFGs using one of the following induction principles. Given a DFG, if a property P holds for all the vertices with no ingoing edge and if P holds for a vertex on condition it holds for all its direct predecessors, then P holds for every vertex. Conversely, if a

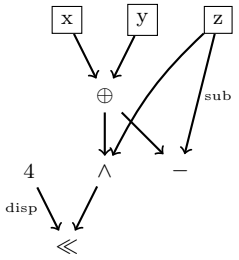

Figure 2.1: A DFG example. We will use the same format to depict DFGs throughout this document. Vertices labelled with a variable symbol have a rectangle shape to distinguish them from those labelled with a operation symbol. For commutative operation symbols, the label of ingoing edges is omitted. For non-commutative operation symbols, a letter/symbol is used to clearly identify the role of each operand.

property P holds for all vertices with no outgoing edge and if P holds for a vertex on condition it P holds for all its direct successors, then P holds for every vertex. These induction principles are correct because DFGs are acyclic.

DFGs are used to represent one or several terms in a concise and efficient way. The DFG given in Figure 2.1 is a possible representation for the following two terms:

$$\begin{cases} \ll (\wedge(z, \oplus(x, y)), 4) \\ -(\oplus(x, y), z) \end{cases}$$

In the literature, DFGs are comparable to term graphs [60] and jungles [39]. Definitions of term graphs usually rely on hypergraphs as underlying structures. Hyperedges in term graphs play the role of vertices in DFGs. They are labelled with variable or operation symbols and they connect a set of operand vertices to a result vertex. Term graphs have a single root vertex that is reachable from any other vertex. As such, they can only be used to represent a single term. Jungles are an extension of term graphs. They can have several root vertices and thus represent several possibly independent terms. Another difference between term graphs and jungles is that, in jungles, the input and output arguments of operations symbols are characterized by a type. Types are represented as vertex labels. For this work, we could have reused the exact formalism and structure of jungles to represent expressions that are computed in x86 assembly code. Unfortunately, we became aware of the literature on this field of research only once most of the work had been done according to our own DFG definition. Despite some formalization details, the two notions (DFG and jungle) are rather similar. In this document we will keep our original DFG definition to more accurately describe what has actually been implemented.

Definition 2 (DFG Morphism)**.** A DFG morphism $f\colon G \to H$ between two DFGs G and H consists of two functions: $f_V\colon V_G \to V_H$ and $f_E\colon E_G \to E_H$ such that:

- $f_V \circ src_G = src_H \circ f_E$ and $f_V \circ dst_G = dst_H \circ f_E$;
- $labV_G(v) = labV_H(f_V(v))$ for all $v \in V_G$ such that $labV_G(v) \notin X$;
- $labE_G(e) = labE_H(f_E(e))$ for all $e \in E_G$ such that $labV_G(dst_G(e))$ is non-commutative ;
- $f_V(u) = f_V(v)$ for all $u, v \in V_G$ such that $labV_G(u) = labV_G(v) \in X$[1].

A DFG morphism f is surjective (respectively injective) if both f_V and f_E are surjective (respectively injective). A morphism is an isomorphism if it is injective and surjective.

A term or more generally a set of terms can be represented by different DFGs. These DFGs differ in the way common subterms are shared. At one extreme, there is a tree representation which has a separate vertex to represent each subterm. And at the other extreme, there is a graph representation, called fully collapsed, which maximizes the sharing of common subterms. More formally, a DFG G is fully collapsed if $term_G$ is injective. The DFG depicted in Figure 2.1 is fully

[1]This last condition can be removed if we assume that for every variable symbol $x \in X$ there is at most one vertex v in G such that $labV_G(v) = x$.

collapsed. Two DFGs representing the same set of terms are said to be bisimilar. Given a DFG G, the bisimilarity class of G contains, up to isomorphism, a single tree element noted ΔG and a single fully collapsed element noted ∇G. There is a surjective morphism from ΔG to G and from G to ∇G.

The DFG definition that has been given so far is abstract. It can be used to represent a wide range of functional expressions. In this work, we will use DFGs to represent the expressions that are computed during the execution of a sequence of assembly instructions. In the remainder of this section, we will describe the specificities that were introduced to adapt this abstract DFG definition for this specific usage.

2.1.2 Memory Operations

In this section we will present the two operations that are used to access the memory. The `load` operation is used to read the value that is stored at a given address and the `store` operation is used to update the value that is stored at a given address. They are defined more formally as follows. A memory state is represented by a mapping from D to itself. The set of all memory states is noted $\mathcal{M}$. `load` and `store` are defined by:

$$\begin{array}{lllll} \texttt{load}: & \mathcal{M} \times D & \rightarrow & D \\ & (m, a) & \mapsto & m(a) \end{array}$$

$$\begin{array}{lllll} \texttt{store}: & \mathcal{M} \times D \times D & \rightarrow & \mathcal{M} \\ & (m, a, v) & \mapsto & m' \text{ where } m'(a) = v \text{ and } m'_{|D\setminus\{a\}} = m_{|D\setminus\{a\}} \end{array}$$

When these two operations are used in DFGs, their memory state operand is omitted. Instead, all the memory operations of a DFG are organized in a sequence. Given such a sequence of memory operations $(op_i)_{1\leq i\leq n}$, it is implicit that the memory state operand of op_j is equal to the memory state returned by the last `store` operation in $(op_i)_{1\leq i<j}$. Therefore in DFGs, `load` operations have a single input operand and `store` operations have two input operands and do not explicitly return any result. Terms rooted by a `store` operation cannot be subterms for any other term. That is to say, vertices labelled with `store` operation symbol do not have any outgoing edges. The size of a memory access is specified by its size attribute (size attributes will be presented in the next section).

Graphically, vertices labelled with a memory operation symbol will have an index to specify their position in the sequence of memory operations. Vertices labelled with `load` are depicted in the same way as vertices labelled with a variable symbol. Intuitively, if a `load` accesses a memory location for the first time, its result is considered to be an input variable. To that extent, assignments also define the values of the initial memory state. Vertices labelled with `store` are depicted in the same way as final vertices (refer to Section 2.4.1 for information on final vertices), that is to say, they have an ellipsis shape. Intuitively, if a `store` accesses a memory location for the last time, it is considered to be an output variable.

2.1.3 Practical Aspects

Operation Symbols

The set Σ of operation symbols with a strictly positive arity that we use to represent sequences of x86 instructions is given in Table 2.1. This set is sufficient for the test scenarios and the use cases presented in this document. But it is clearly far from being sufficiently complete to represent any sequence of x86 instructions. The main design principle is to keep a small number of simple operations to facilitate the normalization phase. Specific instructions such as SSE instructions, should be converted or decomposed into more common operations. Thereby, the influence of instruction selection is reduced. Moreover, fewer operations generally means fewer rewrite rules to normalize them.

Nearly every operation symbol that is used in this work represents not only a single operation but a family of operations. A family of operation regroups operations which basically perform the same transformation but on a different number of operands or on operands of different sizes.

For instance if we take the + operation symbol, it refers to the family of operations of the form $(+_{s,\text{arity}}\colon \{0,1\}^s \times ... \times \{0,1\}^s \to \{0,1\}^s)_{s\in\mathbb{N},\text{arity}>1}$ which compute the sum their operands modulo 2^s. The parameter s which defines the size of operands and the value of the modulus is equal to the size attribute (refer to Section 2.1.3). The arity is defined by the indegree of the vertex. Operations have been regrouped into families to simplify the normalization phase. Many normalization mechanisms are applied to families of operations regardless of the exact characteristics of each of their members. Using the same label to represent every member of a family is an effective way to implement generic normalization mechanisms.

Variable Symbols

There is one variable per x86 register, except for the EIP and the EFLAGS register. For reasons that are detailed in the following section and in the footnote of Table 2.1, modifications affecting these two registers are not transcribed in DFGs. SIMD registers are represented by several variables. The association between registers and variables is important only during DFG construction. It has no influence on the normalization and identification phase.

Size Attribute

Each vertex v of a DFG G has a size attribute, called $\text{size}_G(v)$. The size attribute of a vertex v specifies the exact operation inside the family of operations associated with $labV_G(v)$. It generally defines the size of the operands and the size of the result. Vertices labelled with variable symbols also have a size attribute. All the vertices labelled with the same variable symbol must have the same size attribute. DFGs have the following property: for any edge e, the size of the result returned by $src(e)$ (or the size attribute of $src(e)$ if its label is a variable symbol) is equal to the size of the i^{th} operand of $dst(e)$, where i is the position of e in the sequence of edges which have the same destination vertex than e ordered according to their label. This property is obtained while constructing DFGs by adding explicit size modifier operations such as `movzx`, `part18`, `part28` and `part116` (refer to Section 2.3.3 for an example). This property is preserved during the normalization process. The size attribute may be assimilated to types in jungles. In the rest of this document, we will only consider assignments which map variable symbols to bit strings the size of which is equal to the size attribute.

2.2 Sequence of Dynamic Instructions

2.2.1 Straight-Line Code Hypothesis

Due to performance and security considerations (typically to resist timing attacks), implementations of symmetric cryptographic algorithms tend to avoid conditional branches. For instance, there is no conditional statement in the implementation of the compression function of MD5 given in the RFC [62]. As another example, in the most widespread implementation of AES (tables implementation), the only branching instruction is for the main loop that iterates the round function. Besides, most of the times this loop is partially unrolled. This observation has two consequences.

- Even though lack of conditional branches is a good indication of cryptographic code, control flow information is of no use to precisely identify primitives. In fact, the only control flow patterns that may be encountered in cryptographic code are common (loop structure) and may be subject to variations (loop unrolling). Therefore, modifications of the EIP and the EFLAGS register are not represented in DFGs, since they probably cannot be used to create distinctive patterns.

- The number of branching instructions which are dependent on input data should be extremely small. Thus, execution paths inside cryptographic code are not supposed to change significantly from one execution to another. To take advantage of this observation, we assume that the code to be analysed is a sequence of instructions which are executed from the first to the last. This sequence of instructions corresponds to one possible execution path. It can easily be obtained in practice by recording the instructions that are executed during a

Table 2.1: List of the operation symbols in Σ. The size attribute is noted s.

Symbol	Prototype	Description
`adc`	$(\{0,1\}^s \times \{0,1\}^s \rightarrow \{0,1\}^s)_s$	Modular addition with carry[2].
$+$	$(\{0,1\}^s \times ... \times \{0,1\}^s \rightarrow \{0,1\}^s)_{s,\text{arity}>1}$	Modular addition (the modulus is equal to the size attribute).
$\wedge$	$(\{0,1\}^s \times ... \times \{0,1\}^s \rightarrow \{0,1\}^s)_{s,\text{arity}>1}$	Bitwise AND.
`cmov`	$(\{0,1\}^s \times \{0,1\}^s \rightarrow \{0,1\}^s)_s$	Conditional move[2].
`divq`	$(\{0,1\}^{2s} \times \{0,1\}^s \rightarrow \{0,1\}^s)_s$	Divide the first argument by the second, returns the quotient.
`divr`	$(\{0,1\}^{2s} \times \{0,1\}^s \rightarrow \{0,1\}^s)_s$	Divide the first argument by the second, returns the remainder.
`idivq`	$(\{0,1\}^{2s} \times \{0,1\}^s \rightarrow \{0,1\}^s)_s$	Signed division, returns the quotient.
`idivr`	$(\{0,1\}^{2s} \times \{0,1\}^s \rightarrow \{0,1\}^s)_s$	Signed division, returns the remainder.
`imul`	$(\{0,1\}^s \times ... \times \{0,1\}^s \rightarrow \{0,1\}^s)_{s,\text{arity}>1}$	Signed multiplication.
`movzx`	$(\{0,1\}^* \rightarrow \{0,1\}^s)_s$	Zero-extend (similar to the x86 instruction).
$\times$	$(\{0,1\}^s \times ... \times \{0,1\}^s \rightarrow \{0,1\}^s)_{s,\text{arity}>1}$	Modular multiplication (the modulus is equal to the size attribute).
`neg`	$(\{0,1\}^s \rightarrow \{0,1\}^s)_s$	Two's complement.
$\neg$	$(\{0,1\}^s \rightarrow \{0,1\}^s)_s$	Bitwise NOT.
$\vee$	$(\{0,1\}^s \times ... \times \{0,1\}^s \rightarrow \{0,1\}^s)_{s,\text{arity}>1}$	Bitwise OR.
`part18`	$\{0,1\}^* \rightarrow \{0,1\}^8$	Extracts bit segment [0:7] (0 refers to the least significant bit). The size attribute is equal to 8.
`part28`	$\{0,1\}^* \rightarrow \{0,1\}^8$	Extracts bit segment [8:15] (0 refers to the least significant bit). The size attribute is equal to 8.
`part116`	$\{0,1\}^* \rightarrow \{0,1\}^{16}$	Extracts bit segment [0:15] (0 refers to the least significant bit). The size attribute is equal to 16.
$\circlearrowleft$	$(\{0,1\}^s \times \{0,1\}^8 \rightarrow \{0,1\}^s)_s$	Rotate to the left the first operand by the number of bits specified by the second operand.
$\circlearrowright$	$(\{0,1\}^s \times \{0,1\}^8 \rightarrow \{0,1\}^s)_s$	Rotate to the right the first operand by the number of bits specified by the second operand.
`sbb`	$(\{0,1\}^s \times \{0,1\}^s \rightarrow \{0,1\}^s)_s$	Modular subtraction with borrow[2].
$\ll$	$(\{0,1\}^s \times \{0,1\}^8 \rightarrow \{0,1\}^s)_s$	Shift to the left the first operand by the numbers of bits specified by the second operand.
`shld`	$(\{0,1\}^s \times \{0,1\}^8 \times \{0,1\}^s \rightarrow \{0,1\}^s)_s$	Shift to the left and shift in from the right bits from the third operand.
$\gg$	$(\{0,1\}^s \times \{0,1\}^8 \rightarrow \{0,1\}^s)_s$	Shift to the right the first operand by the numbers of bits specified by the second operand.
`shrd`	$(\{0,1\}^s \times \{0,1\}^8 \times \{0,1\}^s \rightarrow \{0,1\}^s)_s$	Shift to the right and shift in from the left bits from the third operand.
$-$	$(\{0,1\}^s \times \{0,1\}^s \rightarrow \{0,1\}^s)_s$	Modular subtraction (the modulus is equal to the size attribute).
$\oplus$	$(\{0,1\}^s \times ... \times \{0,1\}^s \rightarrow \{0,1\}^s)_{s,\text{arity}>1}$	Bitwise XOR.
`load`	$(\{0,1\}^{32} \rightarrow \{0,1\}^s)_{s\in\{8,16,32,64\}}$	Memory read operation. We assume here that the size of an address is 32 bits.
`store`	$(\{0,1\}^{32} \times \{0,1\}^s \rightarrow \text{void})_{s\in\{8,16,32,64\}}$	Memory write operation. We assume here that the size of an address is 32 bits.

[2]Because the EFLAGS register is not represented at the DFG level, it is impossible to correctly specify these operations. They definitely have to be represented in DFGs since we cannot ignore the effect they have on their operand(s). But they are unaffected by the normalizations mechanisms. The EFLAGS register has been dropped for simplicity reason only. In fact, operations which rely on the EFLAGS register do not play an important role in the cryptographic algorithms that have been tested so far. Consequently, there was no need to normalize them and thus to accurately specify them.

given execution. This hypothesis, called the straight line code hypothesis, greatly simplifies what our DFG model would have been otherwise. In fact, according to the straight line code hypothesis we know exactly which instructions are executed and in which order.

2.2.2 Execution Trace

The sequence of dynamic instructions which are executed during a given execution, is called an execution trace. Depending on the architecture, the operating system, or if the code runs in kernel or user space different tools can be used to record execution traces. For the experimental results presented in this document, we used PIN [52], a DBI framework that is able to instrument x86 programs on both Windows and Linux. As a substitute for PIN, one could have used other DBI framework such as DynamoRio, Valgrind or Qemu to name but a few.

To reduce the size of execution traces, the sequence of dynamic instructions is divided into basic blocks. The basic blocks partitioning is obtained from the DBI framework. Because basic blocks are computed dynamically by the DBI framework, their definition may differ from what is traditionally considered in static analysis. In fact, as new branches in the CFG are encountered, already existing basic blocks may be subdivided. Therefore, a single instruction at a given address may belongs to several basic blocks of different sizes.

We adopt a simple trace format based on two structures: an array of basic blocks and a sequence of basic block indices. For multi-threaded programs there is one sequence of basic block indices per thread. This simple format could benefit from many improvements to reduce its storage consumption. It is mentioned here as an example in order to provide a complete description of the solution. The design of efficient execution trace formats is out of the scope of this work.

For reasons that will be detailed in Section 3.6.5, we also add the possibility to record in the execution trace the value of the address of every memory access.

2.2.3 Segment Selection

An important design principle is to limit the analysis to a trace segment. Previous works in the domain have already proposed numerous heuristics to locate with more or less precision possible locations of cryptographic algorithms. The idea is to use some of them as front end filters to select interesting trace segments. There is no hard constraint on the precision of the segment selection method. Obviously, at least one execution of the algorithm that we want to identify must be included in the segment. But using large segments should not affect too much the reliability of the primitive and mode of operation identification methods. Some of the heuristics which can be used to select suitable trace segments are discussed in Section 6.2.2.

Working only on small segments and not on the full execution trace has several advantages. The first and the most obvious one is performance. Some of the algorithms which are involved in the normalization phase and in the primitive identification phase have a high complexity. Some of them, such as the subgraph isomorphism algorithm, would not even be tractable if they were executed directly on a full execution trace. To give orders of magnitude, an execution trace for a realistic program contains billions of dynamic instructions whereas trace segments that are usually converted to DFGs only contain thousands of dynamic instructions. Simplicity is another advantage. Only x86 instructions which are frequent in cryptographic implementations need to be translated into DFGs. Normalization mechanisms only need to be effective for operations and terms which are common in cryptographic implementations. Finally, even though we claimed that our methods can process large trace segments, the smaller the trace segment is, the higher are the chances of success. This is especially true for primitive identification. Mode of operation identification should not be disturbed by large trace segments.

As explained at the end of Section 2.3.3 it is possible to increase the size of trace segment without needing to rerun the complete analysis. A common strategy is to first consider a small trace segment for the primitive identification (around one thousand dynamic instructions) and then to increase its size (up to a hundred thousand dynamic instructions) for mode of operation identification.

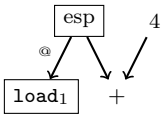

Figure 2.2: DFG associated with the instruction `pop eax`.

2.3 Construction

In this section we describe the method that is used to construct DFGs. This method takes as input an execution trace segment and returns a DFG that represents the different computations occurring in this segment. Iteratively, each instruction is mapped to a separate DFG. Then these small DFGs are concatenated to form a larger DFG that represents a sequence of instructions.

2.3.1 Mapping x86 Instructions to DFGs

Each x86 instruction is associated with a DFG that represents the different expressions that are evaluated by the instruction. These expressions are defined by the semantics of the assembly language. Immediate operands are represented by constants, register operands by variables symbols and memory accesses are explicitly transcribed using the `load` and `store` operation symbols. For instance, the DFG associated with the instruction `pop eax` is given in Figure 2.2. Operations affecting the EIP and the EFLAGS register are discarded.

Along with a DFG G_i, a mapping $\sigma_i \colon X_i \to V_{G_i}$ is also returned for each instruction i. $X_i \subset X$ is the set of registers that are modified by instruction i. This mapping is used to record the state of the registers after instruction i. If a register is modified by instruction i, σ_i will map it to its new content, that is to say, to the term that was written in it. For instance, the mapping associated with the instruction `pop eax` is:

$$\begin{cases} \sigma_{\texttt{pop eax}}(\texttt{esp}) = +(\texttt{esp}, 4) \\ \sigma_{\texttt{pop eax}}(\texttt{eax}) = \texttt{load}(\texttt{esp}) \end{cases}$$

These mappings are called substitutions and they will be used in Section 2.3.3 to compose DFGs.

2.3.2 SIMD Instructions

Single Instruction Multiple Data (SIMD) instructions, such as MMX or SSE instructions are relatively frequent in implementations of cryptographic primitives. Their use might be a deliberate choice made by software developers to achieve better performance. In that case, the code is likely to be written directly in assembly. But plain C/C++ code can also be compiled to SIMD instructions due to optimizations of the compiler. For instance, Gladman's implementation of AES [31] compiled with the newest versions of Clang (at least from version 3.5.0) with optimization level `-O2` includes numerous SSE instructions.

As motivated at the end of Section 2.1.3, it is better for SIMD instructions not to have their own equivalent in Σ. Therefore, we must try to decompose and translate SIMD instructions using only the operations given in Table 2.1. It involves two distinct mechanisms, namely instruction splitting and partial evaluation. They are detailed as follows.

Instruction Splitting

As suggested by their name SIMD instructions execute the same operation on multiple operands in parallel. Thus it is possible to transcribe SIMD instructions into smaller operations that process each, one part of the operands. The only difficulty is to determine in how many parts instructions they should be divided in. For some mnemonics such as `paddb` (modular addition on 8 bits) the answer is straightforward. There is only one partition of `paddb` that can be represented with the operations listed in Table 2.1: eight 8-bit words if it is applied to 64-bit operands or sixteen 8-bit words if it is applied to 128-bit operands. But, for other mnemonics such as `pxor` (bitwise XOR) several partitions are possible. In this latter case, we perform a backward search in the

sequence of instructions. If a partition has already been defined for one of the input register, we use it. Otherwise, we split the instruction according to the word size of the architecture. Because of specific mnemonics (which have a single valid partition), at some point in the sequence of instructions, we could determine with certainty the size of the individual variables that are stored in a SIMD register. The idea is to propagate this information to choose the right partition when translating others instructions that would access the same register. Obviously, a partition stands as long as the register is not overwritten. To be more complete, we could also have done a forward search to see if any following instruction imposes some constraints on the way registers must be partitioned. However, searching for a partition requires a good understanding of each instruction semantics as illustrated in the following code snippet.

```
pxor xmm2, [esp]          ; bitwise XOR
movdqa xmm1, xmm2         ; move aligned double quad word
psllw xmm1, xmm2          ; shift word left
```

If we perform a forward search, the partition of the `pxor xmm2, [esp]` instruction will depend on the following accesses to `xmm2`. `xmm2` is read by `movdqa`, but this mnemonic does not impose any particular partition. The partition of `movdqa` itself depends on the following accesses to `xmm1`. `xmm1` is next read by `psllw` and is interpreted as eight 16-bit words. Thus, `pxor` should be split into eight $\oplus$ operations. For practical reasons, in our implementation the execution trace is processed in a single pass over the sequence of instructions. Therefore, partition information is only available for the precedent instructions that have already been processed.

Notice that this scheme breaks the formalism of the DFG construction that has been introduced so far. Instructions cannot be processed separately. The context in which they are executed also needs to be taken into account to construct their DFG counterpart.

Partial Evaluation

We can take advantage of immediate operands to associate SIMD instructions to simpler semantically equivalent DFGs. Let us consider for instance the mnemonic `pinsrw`. It inserts in its first operand (which is either an MMX or an SSE register) the first least significant word of its second operand at the word position specified by its third operand. A first possibility is to associate instructions based on `pinsrw` to a series of `cmov` operations, one for each word of their first operand. Thereby, whichever position the word must be inserted at, the behaviour of the instructions will correctly be transcribed. A second possibility is to take the value of the third operand (which is always an immediate operand) into account, to determine the position of the insertion. Interpreted as such, instructions based on `pinsrw` are semantically equivalent to `mov` instructions. Thus, they can be associated with an empty DFG and a substitution that maps the right part of the destination register to the least significant word of the second operand. We call this mechanism partial evaluation. Part of an instruction is interpreted with respect to the value of an immediate operand to obtain a simpler expression. In our work, partial evaluation mostly concerns mnemonics that perform bits manipulation such as `palignr`, `pinsrw`, `pshufd` and `pslldq`.

2.3.3 DFG Composition

To obtain the DFG representing a sequence of instructions, we compose the DFGs associated with each of its instructions. A composition of two DFGs is defined with respect to a substitution.

Definition 3 (DFG Composition)**.** Let G and H be two DFGs and $\sigma\colon X' \to V_G$ be a substitution. The composition of H with G with respect to σ (noted $H \circ_\sigma G$) is equal to the union of G and H where every vertex $v \in V_H$ such that $labV_H(v) \in X'$ has been replaced by $\sigma(labV_H(v))$. More precisely:

- $V_{H\circ_\sigma G} = V_G \cup V_H \setminus \{v \text{ s.t. } labV_H(v) \in X'\}$;
- $E_{H\circ_\sigma G} = E_G \cup E_H$;

- $src_{H\circ_\sigma G}(e) = \begin{cases} src_G(e), & \text{if } e \in E_G \\ \sigma(labV_H(src_H(e))), & \text{if } e \in E_H \text{ and } labV_H(src_H(e)) \in X' \\ src_H(e), & \text{otherwise} \end{cases}$
- $dst_{H\circ_\sigma G}(e) = \begin{cases} dst_G(e), & \text{if } e \in E_G \\ dst_H(e), & \text{otherwise} \end{cases}$

The definitions of $labV_{H\circ_\sigma G}$ and $labE_{H\circ_\sigma G}$ are straightforward and are not detailed here.

Let us consider two instructions i_1 and i_2 mapped respectively to (G_{i_1}, σ_{i_1}) and (G_{i_2}, σ_{i_2}). The DFG representing the sequence $i_1 \parallel i_2$ is equal to $G_{i_2} \circ_{\sigma_{i_1}} G_{i_1}$. The substitution associated with $i_1 \parallel i_2$ is equal to:

$$\begin{array}{lcl} X_{i_1} \cup X_{i_2} & \rightarrow & V_{G_{i_2}\circ_{\sigma_{i_1}} G_{i_1}} \\ x & \mapsto & \begin{cases} \sigma_{i_2}(x), & \text{if } x \in X_{i_2} \\ \sigma_{i_1}(x), & \text{otherwise} \end{cases} \end{array}$$

By induction, this construction method can be extended to arbitrary long sequences of instructions.

From an implementation perspective, a substitution corresponds to an array that maps registers to vertices. But because of nested registers, its manipulation is not as straightforward as one may think. Let us consider the following assembly code snippet:

```
mov al, [eax]
add eax, 1
```

The first instruction modifies the least significant bits of `eax`. The second instruction reads the totality of `eax`. To compose the DFGs associated with these two instructions we need to replace the vertex labelled with `eax` in the second DFG by $\sigma_1(\texttt{eax})$. Unfortunately σ_1 is undefined for `eax` and we cannot consider that `eax` is an input variable, and leave it as it is, because part of it was assigned during the first instruction. The solution is to explicitly insert a term equal to the value of `eax` after the first instruction and to map it to `eax` in σ_1. This new term takes as input $\sigma_1(\texttt{al})$ and the most significant bits of `eax`. Then, we can perform the replacement as described in the composition definition. This example is illustrated in Figure 2.3.

Arrays implementing substitutions are manipulated using two functions:

- **set_exp**: This function takes two operands: a register r and a vertex v. It maps v to r. Every previous association made with a register that is included in r is deleted. This function is called every time an instruction writes a register.
- **get_exp**: This function takes as input a register r. It returns the vertex which is currently mapped to r or a vertex representing a new term. This latter case happens if either there is no vertex associated with r or if a register that is included in r has been modified more recently than r. This new term aggregates the vertices that were mapped to registers after the last modification of r and which are included in r. If there is none, the new term is a variable symbol. This function is called every time an instruction reads a register.

Inserting Annotated DFGs

Let G_S be a DFG representing a sequence of instructions S. With our construction method, it is possible to reuse G_S to construct the DFG representing any sequence of instructions that includes S. For instance to construct the DFG representing the sequence $S' \parallel S$, we construct $G_{S'}$ and then we compose G_S with $G_{S'}$. During the normalization phase, σ_S can be updated so that it points to terms which are semantically equivalent[3] to the original ones. Altogether

[3] A notion of equivalence between terms, called observable similarity, is defined more formally in Section 3.1.1. Since the normalization process preserves the observable similarity for the vertices of $\sigma_S(X_S)$, it is possible to update σ_S so that it keeps pointing to observably similar vertices.

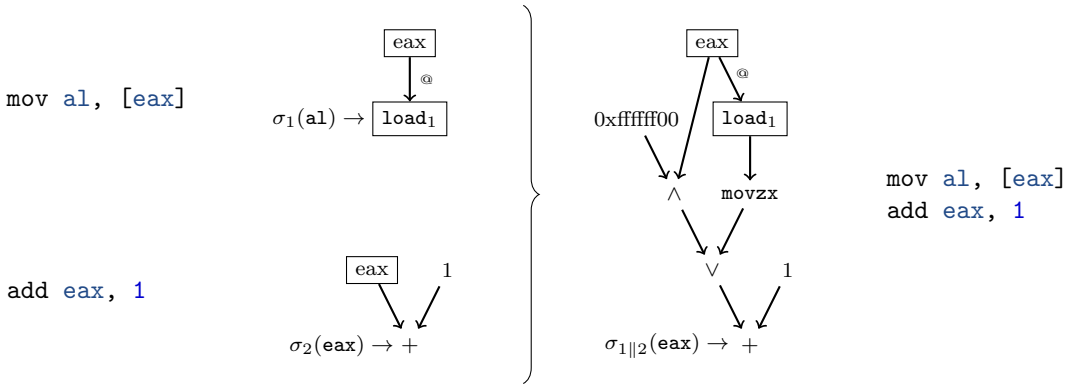

Figure 2.3: Combination of two DFGs with nested registers.

it is possible to reuse a DFG that has already been normalized and annotated (as a result of the primitive identification phase) to construct new DFGs. There are two advantages of doing so. First, parts of large DFGs intended for mode of operation identification can be processed for primitive identification separately at a lesser cost and with a higher chance of success. Despite a relative resistance to segment selection, the primitive identification method generally produces better results if it is executed on trace segments of limited size. Ideally, trace segments should contain a single execution of a primitive. However, the DFGs required for mode of operation identification are by definition much larger. The solution is to first extract small trace segments corresponding each to an execution of a primitive. Then, once primitives have been identified, these small DFGs are inserted along with their annotations to compute the larger DFGs which are required for mode of operation identification. The second advantage is to process only once the parts of large DFGs which are similar. If a cryptographic primitive is executed several times and if the sequence of instructions is the same for each of its executions (which is likely to be the case as explained in Section 2.2.1), only one of its executions needs to be analysed. To construct a DFG representing a segment including several executions, the same DFG can be inserted multiple time.

2.4 Built-in Mechanisms

In this section we present two mechanisms that facilitate the rewriting of DFGs. These mechanisms are transparently executed every time DFGs are modified. The first one is called dead code removal. It deletes every vertex that does not have at least one final vertex among its successor(s). The second one is called value range analysis. It over-approximates the set of values that a vertex could have under any assignment.

2.4.1 Dead Code Removal

Definition 4 (Final Vertices)**.** Let G_S be a DFG and $\sigma_S \colon X_S \to V_{G_S}$ be the substitution associated with G_S resulting from the construction method described in Section 2.3. A vertex $v \in V_{G_S}$ is said to be final if $v \in \sigma_S(X_S)$. Intuitively, a vertex is final if it represents a term that is either stored in a register or in a memory location at the end of the sequence of instructions S. A vertex labelled with `store` is always considered to be final. The set of all the final vertices of G_S is called $Root_{G_S}$. Final vertices are depicted with an ellipsis shape.

We call dead code any vertex that is not final and that does not have any final vertex among its successors. Dead code corresponds to terms which are computed but which are not stored in a register nor in a memory location at the end of the sequence of instructions. These terms are considered to be irrelevant and can be deleted to reduce the size of DFGs. Dead `store` elimination (if several `store` occur at the same address) is not handled by dead code elimination but by a normalization mechanism called memory access simplification (refer to Section 3.6).

A first pass to delete dead code is performed immediately after the DFG construction. Even if the initial code was correctly optimized, dead code can still be found. For instance, terms that

are only used to control the execution flow (such as a loop iteration counter) are dead code. If the directed acyclic graph is in inverse topological order[4] dead code removal can be implemented with a single pass over the vertices. If a vertex is not in $Root_G$ and has no outgoing edge, it is deleted. Later during the normalization phase, dead code removal is triggered every time an edge is deleted. Recursively, if its source endpoint is left with no outgoing edge and is not final, it is deleted.

2.4.2 Value Range Analysis

The goal of value range analysis is to compute the set of all the values that a register or a memory location can have at runtime. In other words, given a DFG G and a vertex $v \in V_G$ we want to determine the set of the evaluations of v under every possible assignment. This set is called the set of reachable values of v and it is noted $D_G(v)$. If $term_G(v)$ is a constant, $D_G(v)$ is a singleton. If $term_G(v)$ is a variable symbol, $D_G(V) = \{0,1\}^{\text{size}_G(v)}$. If $term_G(v)$ is a composite term of the form $f(t_1, ..., t_n)$, $D_G(v)$ can be computed based on the set of reachable values of each of its operands $D_G(v) = f(D_G(t_1), ..., D_G(t_n))$. Value range analysis is extensively used during the normalization phase, mostly by alias analysis but also by some other normalization mechanisms. Refer to Section 3.4.2, 3.5 and 3.6 for examples of normalization mechanisms that use value range analysis.

In practice, it is often impossible to efficiently compute and store exact sets of reachable values. Therefore, instead of working with the exact sets, we will consider over-approximations (supersets) that can be manipulated more easily. Working with over-approximations does not break the soundness of the normalization phase. In fact, value range analysis is used to ensure that modifications of the DFG are valid for any reachable value. Thus, any additional element to the set of reachable values, adds a new constraint on the modifications that can be applied to the DFG. But in any case, if a modification is valid for an over-approximated set of reachable values, it will still be valid for the real set of reachable values. In this work, sets of reachable values are over-approximated by RICs.

Definition 5 (RIC)**.** A Reduced Interval Congruence (RIC) is a finite set of integers of the form: $\{2^a \times x + c \pmod{2^d}$, with $x \leq b$ and $x \in \mathbb{N}\}$ where a, b, c and $d \in \mathbb{N}$. A RIC is represented by the tuple made of these four positive integers (a, b, c, d). The RIC domain is noted $\mathfrak{R}$.

To the best of our knowledge, RICs were first introduced in [8][5]. Their main benefit is their capacity to represent non-convex sets of integers. For instance, $\{1, 3, 5, 7\}$ is equal to the RIC $\{2x + 1 \pmod 8 \mid x \leq 3\}$. The following operations have been implemented for RICs.

Bitwise AND: $\mathfrak{R} \times \mathfrak{R} \to \mathfrak{R}$, given two RICs R_1 and R_2, it returns an over-approximation of the set $\{x_1 \wedge x_2 \mid x_1 \in R_1 \text{ and } x_2 \in R_2\}$. This set is not always a RIC: for instance $[0, 7]$ masked by 5 is equal to $\{0, 1, 4, 5\}$ which is not a RIC. Thus, in order to return a RIC, it has to be over-approximated.

Bitwise OR: $\mathfrak{R} \times \mathfrak{R} \to \mathfrak{R}$, given two RICs R_1 and R_2, it returns an over-approximation of the set $\{x_1 \vee x_2 \mid x_1 \in R_1 \text{ and } x_2 \in R_2\}$. This set is not always a RIC: for instance $[0, 7]$ ORed by 2 is equal to $\{2, 3, 6, 7\}$ which is not a RIC. Thus, in order to return a RIC, it has to be over-approximated.

Include: $\mathfrak{R} \times \mathfrak{R} \to \mathbb{B}$, given two RICs R_1 and R_2, it returns true if $R_1 \subset R_2$, false otherwise.

Intersect: $\mathfrak{R} \times \mathfrak{R} \to \mathbb{B}$, given two RICs R_1 and R_2, it returns true if $R_1 \cap R_2 \neq \emptyset$, false otherwise.

[4] A topological ordering of a directed graph is an ordering over its vertices such that the source vertex of an edge is placed before its destination vertex. Topological orderings exist if an only if the graph is acyclic.

[5] Compared to previous definitions, we add the congruence modulo 2^d to deal more naturally with modular operations and two's complement. In most of the cases d is equal to the size attribute. Another difference concerns the multiplication factor. In our definition it must be equal to a power of two. This choice was made for simplicity reasons. Multiplications by an integer which is not a power of two are not so common in cryptographic code. Therefore, the capacity to better approximate the result of such multiplications does not seem to be worth the extra complexity.

Modular Addition: $\mathfrak{R} \times \mathfrak{R} \to \mathfrak{R}$, given two RICs R_1 and R_2, it over-approximates the set $\{x_1 + x_2 \pmod{\text{size}} \mid x_1 \in R_1 \text{ and } x_2 \in R_2\}$. This set is not always a RIC: for instance $\{0, 1\}$ plus $\{1, 5\}$ is equal to $\{1, 2, 5, 6\}$ which is not a RIC. Thus, in order to return a RIC, it has to be over-approximated. Note that if one of the RICs is a singleton or if both the multiplication factor and the modulus are equal, the set of reachable values is a RIC and no over-approximation is required.

Modular Subtraction: $\mathfrak{R} \times \mathfrak{R} \to \mathfrak{R}$, given two RICs R_1 and R_2, it returns an over-approximation of the set $\{x_1 - x_2 \pmod{\text{size}} \mid x_1 \in R_1 \text{ and } x_2 \in R_2\}$. This operation is implemented using modular addition and two's complement on RICs.

Shift Right: $\mathfrak{R} \times \mathbb{N} \to \mathfrak{R}$, given a RIC $R = (a, b, c, d) \in \mathfrak{R}$ and an integer $n \in \mathbb{N}$ it returns $\{\lfloor x/2^n \rfloor \mid x \in R\}$. This set is a RIC and it is equal to:

$$\begin{array}{ll} \{0\} & \text{if } n \geq d \\ (a - n, b, \lfloor c/2^n \rfloor, d - n) & \text{if } n \leq a \\ (0, \lfloor (b + \lfloor c/2^a \rfloor)/2^{n-a} \rfloor - \lfloor c/2^n \rfloor, \lfloor c/2^n \rfloor, d - n) & \text{otherwise} \end{array}$$

Shift Left: $\mathfrak{R} \times \mathbb{N} \to \mathfrak{R}$, given a RIC $R = (a, b, c, d)$ and an integer $n \in \mathbb{N}$ it returns $\{2^n x \mid x \in R\}$. This set is a RIC and it is equal to $(a + n, b, 2^n c, d + n)$.

Two's Complement: $\mathfrak{R} \to \mathfrak{R}$, given a RIC $R = (a, b, c, d)$ it returns $\{-x \pmod{2^d} \mid x \in R\}$. This set is a RIC and it is equal to $(a, b, -c - 2^a b, d)$.

For operations that are not listed above, their result is simply over-approximated by $\{0, 1\}^s$, where s denotes the size attribute. We dot not try to compute any tight over-approximation for the result of `load` operations as it is the case for instance in Value Set Analysis (VSA) [8]. In VSA, the result of a `load` is over-approximated based on prior `store` operations, the address of which may alias the address of the `load`. In our work, the simplest scenarios (when it is possible to determine exactly which expression was previously stored at a given address) are handled by memory access simplification. Memory access simplification occurs during the normalization phase and replaces `load` by expressions. Value range analysis is regularly updated during the normalization phase. Thereby, better over-approximations might become available for the result of `load` operations as the DFG is normalized. However, in more complex scenarios (when there is at least one prior `store`, the address of which may not alias the address of the `load`) `load` operations will not be replaced by any expression and no tight over-approximation will ever become available. Refer to Section 3.6 for memory access simplification and alias analysis.

The complexity of value range analysis is kept to a minimum. In fact, we do not really need super accurate over-approximations for intricate terms. The solution we have described so far, with its limitations (over-approximated sets must fit the RIC format, multiplication factors must be equal to a power of two, only few operations are supported, no complex over-approximations for `load` operations), is sufficient in practice. Typical terms that we would like to over-approximate are: byte masking (in order to extract a byte from a 32-bit word for instance, which is a recurring feature in many cryptographic algorithms), scale multiplication (in x86 addressing mode) and addition by a constant. Our value range analysis is perfectly capable of computing a satisfactory result for these simple expressions.

Chapter 3

Normalization

The goal of the normalization phase is to reduce the differences that may exist between several implementations of the same algorithm. These differences are introduced by the software developer and by the compiler. Removing them is the key to producing generic signatures that cover a large scope of primitive implementations. Ideally, every implementation of a primitive should converge towards a single normal form. In this chapter, we will present the different transformations that are used to normalize DFGs. Normalization is essential for primitive identification. It is not strictly necessary for mode of operation identification except for memory access simplification. However a light normalization phase (made of a subset of the normalization mechanisms) can usually improve the quality of the results returned by mode of operation identification.

Contents

3.1 Formal Presentation . . . 46
3.1.1 Rewrite Rules . . . 47
3.1.2 Reduction System . . . 49
3.2 Practical Aspects . . . 51
3.3 Common Subexpression Elimination . . . 52
3.4 Constant Simplification . . . 54
3.4.1 Constant Folding . . . 54
3.4.2 Constant Distribution . . . 55
3.4.3 Constant Merging . . . 56
3.5 Constant Expression Detection . . . 57
3.6 Memory Access Simplification . . . 59
3.6.1 Naive Solution . . . 59
3.6.2 Aliased Pointers . . . 60
3.6.3 Correct Solution . . . 61
3.6.4 Static Pointer Comparison . . . 61
3.6.5 Dynamic Pointer Comparison . . . 65
3.6.6 Conclusion . . . 66
3.7 Memory Coalescing . . . 66
3.8 Commutative and Associative Operation Normalization . . . 70
3.9 Affine Expression Simplification . . . 72
3.10 Operation Size Expansion . . . 73
3.11 Miscellaneous Rewrite Rules . . . 75
3.12 Conclusion . . . 75

3.1 Formal Presentation

In this section we give the definition of a rewrite rule and of a rewrite step (application of a rewrite rule to a given DFG). Every normalization mechanism can be modelled by rewrite rules. We

prove that if a rewrite rule preserves a certain equivalence property called observable similarity, this property is also preserved during a rewrite step. The set of all rewrite rules, that is to say the set of all the normalization mechanisms, forms a reduction system. We finally introduce three properties of a DFG reduction system, namely termination, convergence and context-insensitivity. For each of them we explain why they are advantageous or mandatory and we discuss if they are satisfied or not in practice.

3.1.1 Rewrite Rules

Let us consider two DFGs G and H and a DFG morphism $f\colon G \to H$. f induces a substitution $\sigma\colon X \cap labV_G(V_G) \to V_H$ defined for all $x \in X \cap labV_G(V_G)$ by $\sigma(x) = f_V(v)$ where $labV_G(v) = x$. This definition is correct, because according to the definition of a DFG morphism $f_V(u) = f_V(v)$ for all $u, v \in V_G$ such that $labV_G(u) = labV_G(v)$.

Definition 6 (Rewrite Rule)**.** A rewrite rule is defined by a tuple (L, R, ψ), where:

- L is a DFG ;
- R is a DFG such that the variables symbols occurring in R occur also in L ;
- ψ is a surjective mapping from $Root_L$ to $Root_R$.

Definition 7 (Rewrite Step)**.** Let G be a DFG and $r = (L, R, \psi)$ a rewrite rule. Let f be an injective morphism from L to G. We call σ the substitution induced by f and l the morphism from R to $R \circ_\sigma G$ defined by:

$$l_V(v) = \begin{cases} \sigma(labV_R(v)), & \text{if } labV_R(v) \in X \text{ and } \sigma \text{ is defined for } labV_R(v) \\ v, & \text{otherwise} \end{cases}$$
$$l_E(e) = e$$

Then, there is a rewrite step from G to H (noted $G \to_{r,f} H$), where H is the DFG obtained from $R \circ_\sigma G$ by replacing the source vertex of every edge $e \in E_{R\circ_\sigma G}$ such that $src_{R\circ_\sigma G}(e) \in f_V(Root_L)$ by $l_V(\psi(w))$ where w is the preimage of $src_{R\circ_\sigma G}(e)$ under f_V. Note that $src_{R\circ_\sigma G}(e)$ has a unique preimage under f_V because $src_{R\circ_\sigma G}(e) \in f_V(Root_L)$ and f_V is injective. $Root_H$ is obtained from $Root_G$ by replacing every $v \in f_V(Root_L)$ by $l_V(\psi(w))$, where w is the preimage v under f_V. Dead code removal is implicitly triggered during the replacement operations.

The condition that imposes to f to be injective can be replaced by a weaker one: for every $u, v \in Root_L$, $\psi(u) \neq \psi(v)$ implies that $f_V(u) \neq f_V(v)$. Note that searching for an injective morphism from L to G is the same thing that searching for a subgraph of G that is isomorphic to L.

An example of rewrite rule and an example of a rewrite step associated with this rewrite rule are given in Figure 3.1.

Definition 8 (Observable Similarity)**.** Let G and H be two DFGs such that the variable symbols occurring in H occur also in G. H is observably similar to G, if there is a surjective mapping $\phi\colon Root_G \to Root_H$ such that $\theta \circ term_{G|Root_G} = \theta \circ term_H \circ \phi$ for every assignment θ.

Intuitively, H is observably similar to G if, given as input a subset of the input values of G, H computes the same set of output values as G. Different implementations of the same algorithm are not necessarily isomorphic since they may contain non-equal terms. But they are observably similar.

Theorem 3.1.1. *Given a rewrite rule $r = (L, R, \psi)$ if R is observably similar to L according to ψ, then for any rewrite step $G \to_r H$, H is observably similar to G.*

Proof. Let θ be an assignment. We will first prove that:

$$\forall v \in Root_L, \theta(term_G(f_V(v))) = \theta(term_H(l_V(\psi(v)))) \quad (1)$$

Let θ' be an assignment defined by:

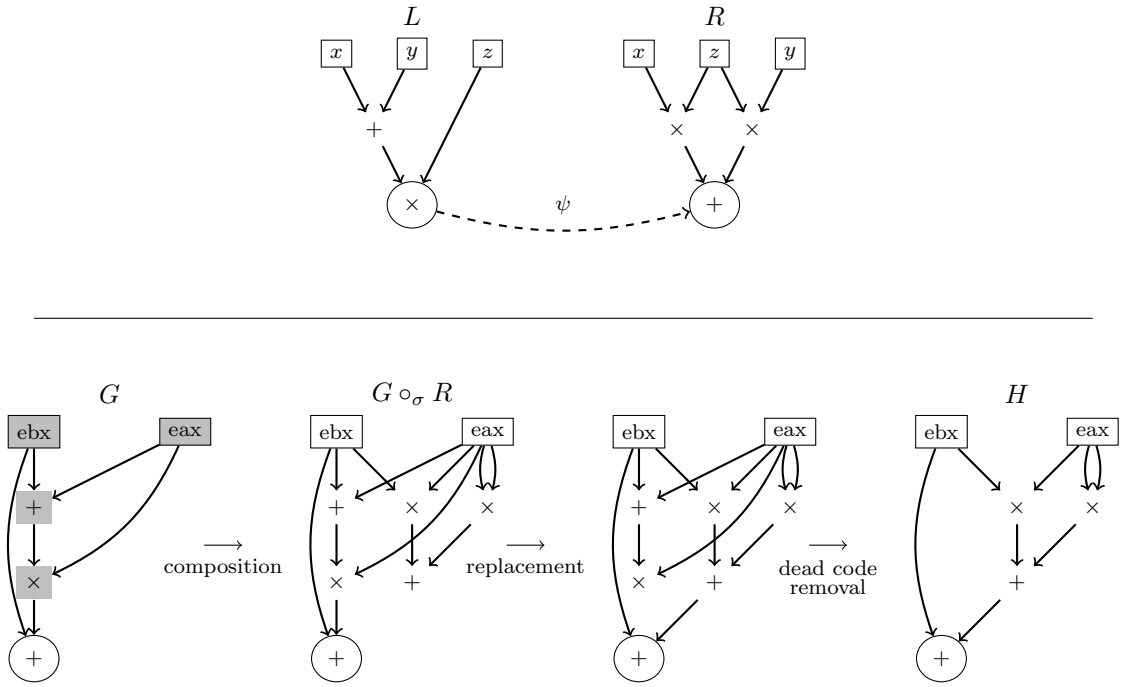

Figure 3.1: Example of a rewrite rule and of a rewrite step. The rewrite rule is given at the top. It illustrates the distributive property of the multiplication over the addition. The function ψ is represented by a dashed arrow and the final vertices are depicted with an ellipsis shape. The intermediate steps that are involved in the construction of H are given at the bottom from left to right. Vertices of G belonging to $f_V(L)$ are highlighted in grey. Note that in this example f is not injective but $f_{V|Root_L}$ is. As explained on the paragraph that follows definition of rewrite step, this is sufficient.

$$\theta'(x) = \begin{cases} \theta(term_G(\sigma(x))) \text{ if } \sigma \text{ is defined for } x \\ \text{any element of } D \text{ otherwise} \end{cases}$$

Let us consider the following statement: $\theta(term_G(f_V(v))) = \theta'(term_L(v))$ for $v \in V_L$. If v is labelled with a constant then $f_V(v)$ is labelled with the exact same constant. Every possible assignment maps a constant to the same element of D, hence the statement is true in that case. If v is labelled with a variable symbol then:

$$\begin{array}{rll} \theta'(term_L(v)) & = \theta(term_G(\sigma(term_L(v)))) & \text{according to the definition of } \theta' \\ & = \theta(term_G(f_V(v))) & \text{according to the definition of } \sigma \end{array}$$

Hence the statement is also true in that case. For the remaining case, where v is labelled with non-constant operation symbol, let us assume that the statement is true for all the direct predecessors $v_1, ..., v_n$ of v. Then, we have:

$$\begin{array}{rl} \theta'(term_L(v)) & = labV_L(v)(\theta'(term_L(v_1)), ..., \theta'(term_L(v_n))) \\ & = labV_G(f_V(v))(\theta(term_G(f_V(v_1))), ..., \theta(term_G(f_V(v_n)))) \\ & = \theta(term_G(f_V(v))) \end{array}$$

Thus, according to the induction principle mentioned in Section 2.1.1, the statement is true for all $v \in V_L$. We can show using the exact same proof that $\theta(term_{G\circ_\sigma R}(l_V(v))) = \theta'(term_R(v))$ for all $v \in V_R$. R is observably similar to L according to ψ, thus in particular $\theta'(term_L(v)) = \theta'(term_R(\psi(v)))$ for all $v \in Root_L$. If we combine this equality, with the two that we just proved, we obtain for all $v \in Root_L$:

$$\begin{aligned}\theta(term_G(f_V(v))) &= \theta(term_{G\circ_\sigma R}(l_V(\psi(v)))) \\ &= \theta(term_H(l_V(\psi(v)))) && \text{by construction of } H\end{aligned}$$

It concludes the first part of this proof. In the second part of this proof, we will show that:

$$\forall v \in V_G \cap V_H, \theta(term_G(v)) = \theta(term_H(v)) \tag{2}$$

We will proceed according to the induction principle. Given $v \in V_G \cap V_H$ then by construction of H, $labV_G(v) = labV_H(v)$. Hence, if v is labelled with a constant or with a variable, the equality holds. Now, let assume that the equality is true for all the direct predecessors of v that are in $V_G \cap V_H$. For simplicity, we assume that v has two direct predecessors in G: $v_1 \in V_G \cap V_H$ and $v_2 \in f_V(Root_L)$. We do not lose generality with this last assumption, since any other case can be proved in the same way. We have:

$$\begin{aligned}\theta(term_G(v)) &= labE_G(v)(\theta(term_G(v_1)), \theta(term_G(v_2))) \\ &= labE_H(v)(\theta(term_H(v_1)), \theta(term_G(v_2))) && \text{induction hypothesis} \\ &= labE_H(v)(\theta(term_H(v_1)), \theta(term_G(f_V(v_2')))) && v_2' \in V_L \text{ s.t. } f_V(v_2') = v_2 \\ &= labE_H(v)(\theta(term_H(v_1)), \theta(term_H(l_V(\psi(v_2'))))) && \text{according to 1} \\ &= \theta(term_H(v)) && \text{by construction of } H\end{aligned}$$

It concludes the proof of 2. Let ψ' be a mapping from $Root_G$ to $Root_H$ defined for all $v \in Root_G$ by:

$$\psi'(v) = \begin{cases} l_V(\psi(w)), & \text{if } v \in f_V(Root_L), \text{ where } w \text{ is the preimage of } v \text{ under } f_V \\ v, & \text{otherwise} \end{cases}$$

We have $\theta(term_G(v)) = \theta(term_H(\psi'(v)))$ for all $v \in Root_G$. The case where $v \in f_V(Root_L)$ is covered by 1 and the case where $v \notin f_V(Root_L)$ is covered by 2. By definition of $Root_H$, ψ' is surjective. That concludes the proof of the theorem. □

Observable similarity is a transitive relation. Therefore, for any sequence of rewrite steps involving observably similar rewrite rules, the final DFG will be observably similar to the first one. Every rewrite rule that takes part in the normalization phase has the observable similarity property. Hence, the whole normalization phase also has it. This is an essential observation. It guarantees that if for every rewrite steps there is no deviation from the original behaviour, then at the end there will be no deviation either. In fact, we cannot tolerate even a slight deviation from the original behaviour because once amplified by the whole set of rewrite rules, it could have dramatic effects on the primitive identification method (for both the false positive and the false negative ratio). The importance of the observable similarity property for primitive identification is discussed in Section 4.1.3.

3.1.2 Reduction System

We note $\mathcal{G}_{\Sigma,X}$ the set of all possible DFGs over Σ and X. Let $\mathcal{R}$ be a set of rewrite rules. We note $\rightarrow$ the binary relation over $\mathcal{G}_{\Sigma,X}$ defined by $\rightarrow = \bigcup_{r\in\mathcal{R}} \rightarrow_r$. The transitive-reflexive closure of $\rightarrow$ is noted $\rightarrow^*$. The tuple $(\mathcal{G}_{\Sigma,X}, \rightarrow)$ is a reduction system. An element $G \in \mathcal{G}_{\Sigma,X}$ is a normal form if there is no $H \in \mathcal{G}_{\Sigma,X}$ such that $G \rightarrow H$. We say that G is a normal form of H if $H \rightarrow^* G$ and G is a normal form. The relation $\rightarrow$ is terminating if there is no infinite sequence of the form: $G_0 \rightarrow G_1 \rightarrow G_2 \rightarrow ...$ If $\rightarrow$ is terminating then every element of $\mathcal{G}_{\Sigma,X}$ has a normal form. The relation $\rightarrow$ is convergent if every element of $\mathcal{G}_{\Sigma,X}$ has a unique normal form up to isomorphism. These two properties can be defined for any reduction system. The following property is specific to DFG reduction systems. A convergent relation over $\mathcal{G}_{\Sigma,X}$ is context-insensitive if for any $G, H \in \mathcal{G}_{\Sigma,X}$ such that there is a DFG morphism from G to H, there is also a DFG morphism from the normal form of G to the normal form of H.

In the remainder of this section, we explain why we would like these three properties to be satisfied by the reduction system used in the normalization phase. The objective is to point out constraints that need to be satisfied by the reduction system. While most of this chapter describes a reduction system which works relatively well in practice, it seems important to understand that some transformations cannot be performed during the normalization phase (or, at least, not without serious consequences on the primitive identification method). For each of the three properties we discuss whether or not they hold in practice. This informal discussion is by no mean a proof of any sort.

Termination. Termination is absolutely mandatory. Otherwise the normalization phase will last forever unless it is stopped at an arbitrary point. But in this case the reduction system is almost certainly not going to be convergent. If there is a rewrite rule in $\mathcal{R}$ which is not terminating then $\mathcal{R}$ is not terminating. But the fact that all rewrite rules of $\mathcal{R}$ are terminating does not imply that $\mathcal{R}$ is terminating. One of the main argument to prove that many rewrite rules are convergent is to compare the size of their input with the size of their output: if a transformation always decreases the number of vertices then it is convergent. The reduction system described in this chapter is not convergent unless rewrite rules are executed in a specific order. In fact it contains rewrites which counteract each other. But when rewrite rules are executed in the right order we did not face during our experiments a DFG with no fixed point. Therefore, we assume that termination is satisfied by our reduction system in practice.

Convergence. Convergence is highly recommended. Ideally every implementation of a primitive should converge towards the same normal form. In other words, there should be a single normal form per 'class' of observably similar DFGs. It clearly implies convergence. As for termination, a single non-convergent rewrite rule is sufficient to break convergence of $\mathcal{R}$; but proving that all rewrite rules in $\mathcal{R}$ are convergent is not sufficient to prove that $\mathcal{R}$ is convergent. An example of a non-convergent term rewrite rule is given below:

$$+(x, x, x) \rightarrow +(\times(x, 2), x)$$

Based on our experiments it is hard to determine whether or not our reduction system is convergent in practice. In our implementation, rewrite rules are always executed in the same order, thus even if we try to normalize several times the same DFG it will always produce the same sequence of rewrite steps. We have not tried to randomize the order in which rewrite rules are executed. In practice tough, the number of signatures per primitive is relatively small compared with the number of implementations that are successfully identified. Moreover, as detailed in Section 4.3.2, the differences between these signatures are generally not related with possible convergence issues. Thus, we assume that our system is convergent in practice.

Context-Insensitivity. Finally the primitive identification method should be independent of the segment selection. We should be able to identify a primitive in a trace segment, as long as it contains one execution of this primitive. It implies the context-insensitivity property. But context-insensitivity is not satisfied in practice. If the left hand side DFG of a rewrite rule contains two or more vertices that are connected and labelled with operation symbols, this rewrite rule does not satisfy the context-insensitivity property. Very few of the normalization mechanisms presented in this chapter satisfy the context-insensitivity property. It causes many problem in practice and reduces the chance of success for large trace segments. To mitigate this issue some rewrite rules (refer to section 3.8) implement mechanisms to limit the code distance between the different vertices involved in a single rewrite step. Thereby an instruction that is too far from a primitive implementation cannot directly influences the way this primitive implementation will be normalized. To conclude, formulated as such context-insensitivity is almost impossible to satisfy and in practice the success rate of the primitive identification method depends on the size of the segments.

3.2 Practical Aspects

Unique Set of Rewrite Rules. DFGs are always normalized with the same set of rewrite rules. The alternative approach is to use different sets of rewrite rules depending on the primitive that is to be identified. This latter approach has several advantages at first glance. Let us assume we want to insert a new rewrite rule into the reduction system to deal with a corner case that is specific to a given primitive. There is a risk that this new rule will break the termination or the convergence property of the system for primitives that were correctly detected so far. At least, it is highly probable that it will modify the normal form of other primitives breaking the compatibility with existing signatures. For these reasons, we may be tempted to use the new rewrite rule only to detect the specific primitive it was intended for. But, in our opinion this is not a good solution. First, if one wants to test several primitives, the same DFG will have to be normalized several times. Second, if a rewrite rule is useful for a primitive it may be useful for other primitives too. It seems a better practice to devise a rewrite rule once and for all than to let users craft their own rewrite rules every time they need to support new primitives. With our approach (a single set of rewrite rules for every signature), users do not need any particular knowledge of the normalization mechanisms to write new signatures (refer to Section 4.1.4 for more details on signature creation). Of course, it comes at a cost. Creating new rewrite rules requires extra caution to be compliant with the rest of the system and existing signatures will probably have to be updated.

Implementation. Despite the fact that every normalization mechanism can be formalized as a rewrite rule, we did not implement a unified algorithm to apply any rewrite rule. Instead each rewrite rules or each family of rewrite rules, is applied using its own specific algorithm. DFGs involved in rewrite rules are usually small. Hence a generic subgraph isomorphism algorithm such as the one presented in Section 4.2 is not mandatory. Some rewrite rules must only be applied if some additional conditions are met. Those conditions can be easily and efficiently be checked by a dedicated algorithm. This is the case for instance for memory access simplification. Many rewrite rules can be regrouped and applied efficiently in a single pass over the vertices whereas a generic subgraph isomorphism algorithm would probably have to be reset after each rewrite step. Moreover, a rewrite step can immediately create a new opportunity for its direct successors. If tested in the right order, some rewrite steps can be applied in a cascading fashion for an optimal efficiency. This is the case for instance for common subexpression elimination and constant folding. Finally, some families of rewrite rules contain an infinity of rewrite rules. Obviously we cannot search each of them independently. This the case for instance for common subexpression elimination and affine expression simplification.

Relation with Compiler Optimizations. Many of the normalization mechanisms presented in this chapter are in fact compiler optimization techniques. This is not a surprising observation.

Many terms represented in DFGs can still be optimized even though they were obtained from correctly optimized code. Because of machine code specificities (number and size of registers, instruction set, memory addressing mode and so on), the most efficient machine code does not always correspond to the smallest or most concise terms. This is especially true for SIMD instructions. For instance in the OpenSSL AES synthetic sample (refer to Section 4.3.1 for more details on synthetic samples), the AES encryption function which makes an intensive use of SIMD instructions, is first translated to a DFG of nearly 6000 vertices. Due to the popularity of the OpenSSL library, we have every reason to believe that its AES implementation was correctly optimized. However, after the normalization phase, the DFG contains less than 2500 vertices. Straight line code hypothesis is another source of possible optimizations. Unlike compilers, we normalize a single execution path. Local optimization techniques can be applied at a much larger scale resulting in many new optimization possibilities.

We will now explain, why terms represented in DFGs must be optimized as part of the normalization process. Here the word 'optimization' means reduction of the size of expressions. First, it goes without saying that from a performance perspective it is faster to work with small DFGs than with large ones. Second and most importantly, normalization mechanisms that reduce the size of DFGs are terminating and have a chance to be convergent. Their inverse transformations however, have very little chance to be terminating. Hence if a single normal form has to be reached from two

Table 3.1: Ordered list of normalization mechanisms. For each of them, we specify the complexity to the number of vertices and whether or not the vertices need to be in a topological ordering. We provide two estimations of the complexity: the one on the left corresponds to the worst case complexity where vertices have a number of direct predecessors and direct successors which is linear in the number of vertices and the one on the right corresponds to a more realistic situation where vertices only have a constant number of direct predecessors and direct successors. The cost of sorting the vertices in a topological ordering is not included in these complexity estimations.

Id	Name	Complexity		Topo. Order
1	Constant Folding	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	yes
2	Constant Expression Detection	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	yes
3	Miscellaneous Rewrite Rules	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	no
4	Common Subexpression Elimination	$\mathcal{O}(n^3.log(n))$	$\mathcal{O}(n^2)$	yes
5	Memory Access Simplification	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	no
6	Constant Distribution	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	no
7	Operation Size Expansion	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	no
8	Memory Coalescing	$\mathcal{O}(n.log(n))$	$\mathcal{O}(n.log(n))$	no
9	Affine Expression Simplification	$\mathcal{O}(n^2.log(n))$	$\mathcal{O}(n^2.log(n))$	no
10	Constant Merging	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	no
11	Comm. & Asso. Operation Normalization	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	no

observably similar DFGs, one well optimized and the other not, the most reasonable solution is to optimize the DFG that is not. Third, smaller expressions can be compared and over-approximated more precisely. This is especially important for address expressions. Simplification of address expressions leads to more precise address comparisons and as explained in Section 3.6 to more thorough memory access simplifications. Not every rewrite rule directly modifies expressions that will be involved in the primitive signature. Some of them, such as affine expression simplification, are just supposed to simplify the surrounding expressions, so that side conditions (aliasing in the case of memory access simplification) can be verified more precisely.

Overview. The list of rewrite rules presented in this chapter has been constructed progressively. New rewrite rules were inserted only when necessary. Thus, this list is by no means complete nor definitive. Important rewrite rules, to normalize primitives which are not covered by our experiments, may still be missing.

There are eleven families of rewrite rules also called normalization mechanisms. They are given in Table 3.1 in the same order as the order in which they are applied to DFGs. This order is not particularly important except for constant merging, constant folding and common subexpression elimination. Constant folding must be executed at least once between constant merging and common subexpression elimination otherwise the system is not terminating. This list is executed repeatedly until a fixed point is eventually reached.

In the remainder of this chapter, there is one section dedicated to each family of rewrite rules. The rewrite rules the left hand side graph of which has a single final vertex will be formulated as term rewrite rules. A term rewrite rule is a tuple of terms. There is no need to specify the mapping ψ because terms have a single root element.

3.3 Common Subexpression Elimination

Common subexpression elimination is a classical compiler optimization technique. If two vertices have the same label and share the same set of direct predecessors in the same order, they represent the same expression. Consequently they can be merged into a single vertex. This transformation correspond to a DFG morphism which maps the two equivalent vertices to a single one. It also corresponds to a rewrite step for the rewrite rule given at the top of Figure 3.2. This rewrite rule can be applied to any operation symbol except `load` and `store` which are handled by the memory access simplification mechanism. This rewrite rule is terminating, convergent [60] and context-insensitive. The normal form associated with this rewrite rule is called fully collapsed.

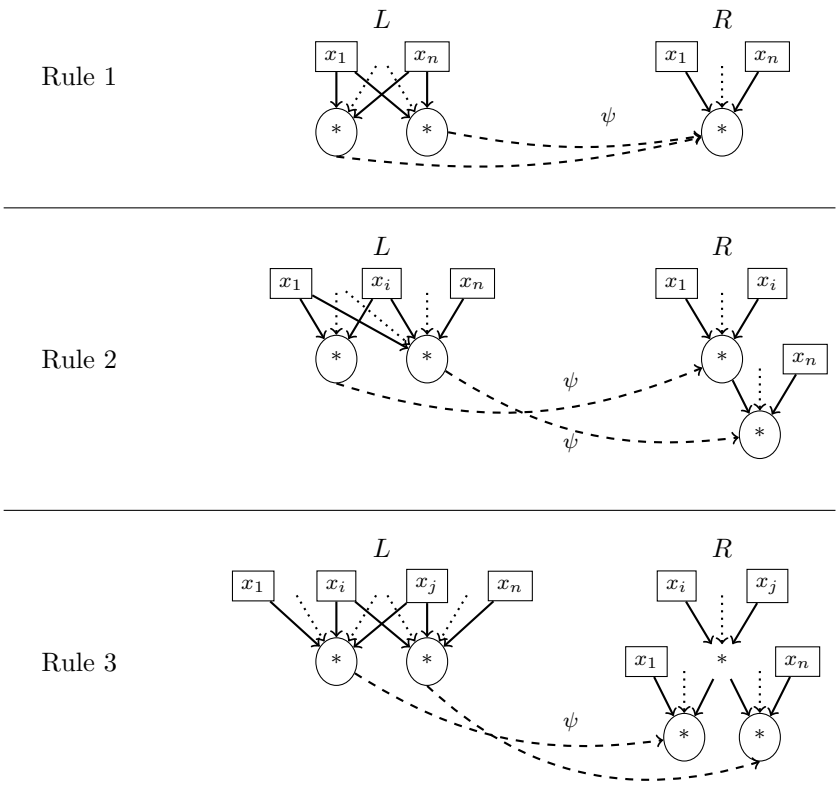

Figure 3.2: The three rewrite rules involved in common subexpression elimination. * is a placeholder for an operation symbol. For Rule 1, it can be substituted by any operation symbol except `load` and `store`. For Rule 2 and 3, it must be substituted by an commutative and associative operation symbol.

As motivated at the beginning of Chapter 2, fully collapsed representations are preferred to tree representations for performance reasons.

Two additional rewrite rules are involved in common subexpression elimination. These rewrite rules, noted Rule 2 and Rule 3 in Figure 3.2, are to be applied on commutative and associative operation symbols only. If two commutative and associative operations share at least two input operands, theses shared operands can be replaced by their partial result. These two rules are terminating. In fact, the number of edges is reduced by at least one. But they are not convergent. A counterexample is as follows. If a vertex shares input operands with two other vertices, then one rewrite step is possible for each of theses vertices. If the two sets of shared operands are neither disjoint nor equal, we will only be able to perform a single rewrite step at a time. Hence we can obtain two different normal forms, depending on which one is executed first. In practice though, we have never faced this issue.

Rule 1, 2 and 3 can be applied simultaneously using Algorithm 2. Each vertex is compared with all the other vertices to determine which input operands do they have in common. Since vertices are first sorted into topological order, we have the guarantee that when processing vertex v, every predecessor of v is fully collapsed. Hence at the end of the algorithm, when every final vertex has been processed, we have the guarantee that the DFG is fully collapsed. Assuming that the comparison of two sets of vertices takes time $\mathcal{O}(n.log(n))$, where n is the number of vertices in the DFG, then the complexity of Algorithm 2 is $\mathcal{O}(n^3.log(n))$. It corresponds to the worst case complexity, when every vertex can have $\mathcal{O}(n)$ direct predecessors.

Algorithm 2 Common Subexpression Elimination (Rule 1, 2 and 3)

```
sort V_G into topological order
for all v in V_G do
    for all u in V_G such that labV_G(u) = labV_G(v) do
        compare the direct predecessors of v with the direct predecessors u
        if the two sets are equal then
            apply Rule 1
        else if one set is included in the other then
            apply Rule 2
        else if if the intersection of the two sets contains more than one element then
            apply Rule 3
        end if
    end for
end for
```

3.4 Constant Simplification

In this section we present three families of rewrite rules, namely constant folding, constant distribution and constant merging. They have been regrouped because they are all related with constant terms and because they are highly dependent on each other. In fact, the only goal of constant distribution and constant merging is to leverage the effect of constant folding.

3.4.1 Constant Folding

We regroup by the name of constant folding, several rewrite rules that are triggered when an operation has one or several constant operands. Depending on the value of these constant operands and on their number, parts of or even the whole term can be replaced by a constant. These rewrite rules are given below for the $\times$ operation symbol:

$$\begin{aligned}
\times(0, x_2, ..., x_n) &\rightarrow 0 \\
\times(1, x_2, ..., x_n) &\rightarrow \times(x_2, ..., x_n) \\
\times(c_1, ..., c_n) &\rightarrow c_1 \times ... \times c_n \\
\times(c_1, ..., c_j, x_{j+1}, ..., x_n) &\rightarrow \times(c_1 \times ... \times c_j, x_{j+1}, ..., x_n)
\end{aligned}$$

Variables x_i refer to non-constant terms and variables c_i refer to constant terms. Similar rewrite rules exist for other operation symbols: $+$, $\wedge$, `imul`, $\vee$, $\ll$, $\gg$ and $\oplus$. For functions that do not have an absorbing element, the first rule is omitted. Constant folding can be applied efficiently in a single pass over the vertices if they are in a topological order.

Many normalization mechanisms, such as common subexpression elimination and memory access simplification, are unlocked by evaluating constant subterms. Constant folding rules play a key role in the normalization process even if the code has been correctly optimized. In fact, due to the straight line code hypothesis, many constant folding scenarios that were not possible at compile time, are now possible. For instance, memory offsets in loops that are computed based on the induction variable, can be partially evaluated once the loop is unrolled.

However, it is often necessary to rearrange groups of operations to efficiently implement constant folding. In fact, constants that are spread over several subterms need to be regrouped in order to apply constant folding rules. For this purpose, we devise two additional sets of rewrite rules: constant distribution and constant merging (detailed respectively in Section 3.4.2 and 3.4.3).

The following example illustrates the rewriting system composed of constant folding, constant distribution and constant merging. It is based on the following C code snippet (which computes the sum of an array):

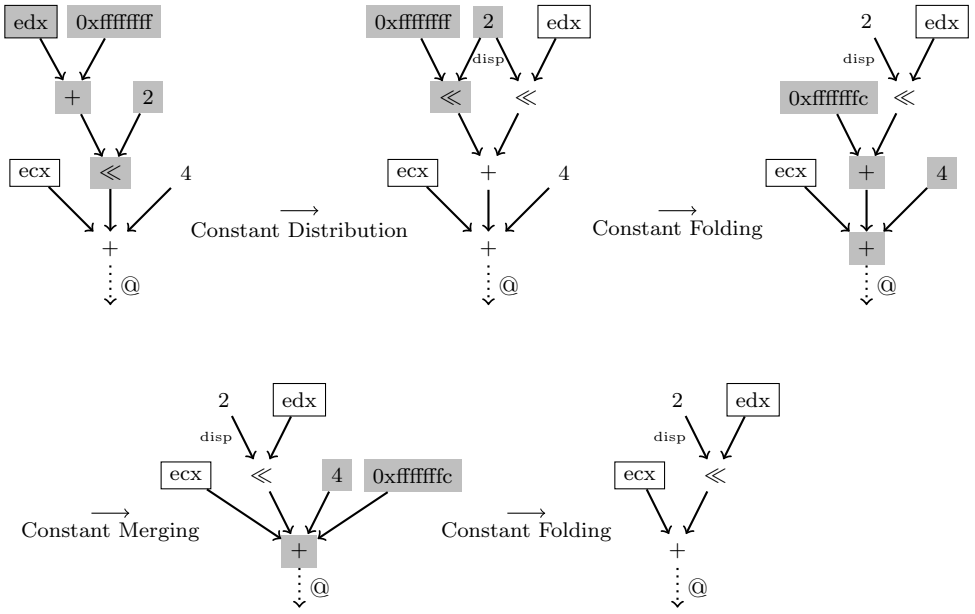

Figure 3.3: Illustration of the rewriting system composed of constant folding, constant distribution and constant merging. The rewriting starts at the top left and ends at the bottom right. The vertices which are directly involved in each rewrite step are highlighted in grey.

```
do {
    b += ptr[a --];
} while(a);
```

The x86 code returned by GCC 5.2.1 (with the `-O2` option) for this code snippet is as follows:

```
sub     edx,0x1
add     eax,[ecx+edx*4+0x4]
test    edx,edx
jne     0xf5
```

In this example we only consider one iteration of the loop. Figure 3.3 details step by step how the rewriting system transforms the DFG. The initial DFG is given at the top left (part of it has been omitted for simplicity). From left to right and top to bottom, the rules that are applied to the DFG are: constant distribution, constant folding, merging and constant folding. We first notice that without regrouping the constant terms on the same operation, no constant folding would have been possible. Second, this example underlines that even for optimized code, constant folding is essential to simplify expressions.

One important drawback of constant folding is that it causes the rewriting system to become more sensible to the context. For instance, if a variable is initialised in the trace segment, constant folding will completely remove that variable from the DFG. Thus, the normalized DFG will greatly differ from the one that would have been obtained if this variable was left uninitialised. During our experiments, we faced this issue for the MD5 synthetic samples.

3.4.2 Constant Distribution

We regroup in this section rewrite rules based on the distribution property. These rules have been introduced to create new constant folding possibilities. Given two operations that both have a constant operand, if we distribute one over the other (assuming that this is a valid transformation), at least one of the newly created terms could benefit from constant folding. We extend this criterion

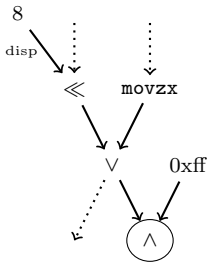

Figure 3.4: A typical DFG that could highly benefit from constant distribution rules. Such DFGs are common in cryptographic code. If an algorithm operates on variables that are smaller than the architecture word size and if several of these variables are stored in the same word, similar expressions will appear.

for certain pairs of distributive operations as follows. If in the distributed expression one of the newly created terms is constant according to value range analysis, then we perform the replacement. Rules based on this latter criterion are especially important to simplify expressions such as the one shown in Figure 3.4. The complete list of rewrite rules that compose the constant distribution subsystem is given below:

$$
\begin{aligned}
\times(+(c_1, x), c_2) &\rightarrow +(\times(c_1, c_2), \times(x, c_2)) \\
\ll(+(c_1, x), c_2) &\rightarrow +(\ll(c_1, c_2), \ll(x, c_2)) \\
\ll(\wedge(x_1, x_2), c_2) &\rightarrow \wedge(\ll(x_1, c_2), \ll(x_2, c_2)) \text{ if } D_G(x_1 \ll c_2) \text{ is a singleton} \\
\gg(\wedge(x_1, x_2), c_2) &\rightarrow \wedge(\gg(x_1, c_2), \gg(x_2, c_2)) \text{ if } D_G(x_1 \gg c_2) \text{ is a singleton} \\
\ll(\vee(x_1, x_2), c_2) &\rightarrow \vee(\ll(x_1, c_2), \ll(x_2, c_2)) \text{ if } D_G(x_1 \ll c_2) \text{ is a singleton} \\
\gg(\vee(x_1, x_2), c_2) &\rightarrow \vee(\gg(x_1, c_2), \gg(x_2, c_2)) \text{ if } D_G(x_1 \gg c_2) \text{ is a singleton} \\
\wedge(\vee(x_1, x_2), c_2) &\rightarrow \vee(\wedge(x_1, c_2), \wedge(x_2, c_2)) \text{ if } D_G(x_1 \wedge c_2) \text{ is a singleton}
\end{aligned}
$$

3.4.3 Constant Merging

Let u and v be two vertices such that there is an edge from u to v. If u and v have the same label and if this label is the symbol of a commutative and associative operation, we can collapse u and v into a single vertex w. The label of w will be equal to the label of u and v and w will have the operands of both u and v. As explained in Section 3.8, this transformation plays a key role in the normalization of commutative and associative operations. In that context, this transformation has many drawbacks and it must be limited to very specific scenarios. But here the situation is quite different. If both u and v have constant operands, merging them will create a new constant folding possibility since w will have at least two constant operands. For the same reason that motivated us to introduce constant distribution in the previous section, we will merge commutative and associative operations that have constant operands. This transformation is formalized by the following rewrite rule:

$$+(c_1, +(c_2, y_2, ..., y_m), x_3, ..., x_n) \rightarrow +(c_1, c_2, x_3, ..., x_n, y_2, ..., y_m)$$

Here + acts as a placeholder for any commutative and associative operation symbol. Currently in our solution, this rewrite rule has been implemented for + and $\wedge$. In Figure 3.5 there is an example of a normalization phase that involves a constant merging rewrite step. This example illustrates clearly that if constant folding is not executed between constant merging and common subexpression elimination, the reduction system is not terminating. This example also shows that,

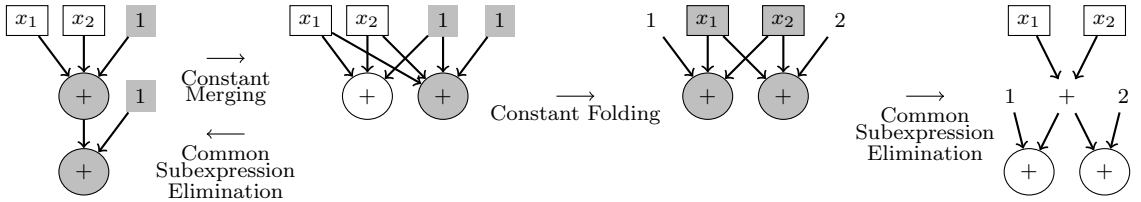

Figure 3.5: Example of a normalization phase that involves a constant merging rewrite step. Note that if common subexpression elimination is executed immediately after constant merging, there is a cycle.

once constants have been simplified, shared operands are regrouped due to common subexpression elimination (Rule 1 and 2 , refer to Section 3.3) and a hierarchical expression involving several operations is recreated. This is the main difference with commutative and associative expression normalization which is also based on the collapsing of commutative and associative operations.

3.5 Constant Expression Detection

The goal of constant expression detection is to find expressions that always evaluate to the same value for every assignment and to replace them by the corresponding constants. Stated as such constant expression normalization seems to encompass constant folding. This is true only in certain scenarios. Constant expression detection is very different from constant folding in the way it is implemented. It only supports a subset of the operations supported by constant folding and it cannot deal with identity elements. Constant expression detection relies on value range analysis and on influence measurement.

Influence Mask. To measure at the bit granularity the influence other vertices have on a given vertex v, we use a bit mask $M_v^{\leftarrow}\colon V_G \to \{0,1\}^*$. If the i^{th} bit of $M_v^{\leftarrow}(u)$ is equal to 0, it means that v is not influenced by the i^{th} bit of u. That is to say, for every assignment θ, $\theta(v)$ is not modified if we flip the i^{th} bit of $\theta(u)$. Conversely, if the i^{th} bit of $M_v^{\leftarrow}(u)$ is equal to 1, it means that v may be influenced by the i^{th} bit of u. This latter relation is treated in a conservative way. In particular it does not imply that there is an assignment θ such that flipping the i^{th} bit of $\theta(u)$ modifies $\theta(v)$. This is the default value if we cannot prove that a bit has no influence on v.

To compute $M_v^{\leftarrow}$, we start with $M_v^{\leftarrow}(v) = 11...1$ and we propagate backward the mask along the edges. For every edge e it traverses, the mask is updated. This update process is formalized by a function $I_{-e}\colon \{0,1\}^{\text{size}(dst(e))} \to \{0,1\}^{\text{size}(src(e))}$ which takes as input an influence mask for $dst(e)$ and returns an influence mask for $src(e)$. Let m be a mask in $\{0,1\}^{\text{size}(dst(e))}$. The i^{th} bit of $I_{-e}(m)$ is equal to 0 if, assuming that e is the only outgoing edge of $src(e)$, there is no $j \in \{1,...,\text{size}(dst(e))\}$ such that the j^{th} bit of m is equal to 1 and the i^{th} bit of $src(e)$ influences the j^{th} bit of $dst(e)$. Otherwise the i^{th} bit of $I_{-e}(m)$ is equal to 1. The update function I_{-e} takes every available information into account: the label of e, the label of $dst(e)$ and over-approximations of the other operands of $dst(e)$. $M_v^{\leftarrow}(u)$ is equal to the bitwise OR of the masks obtained from all the outgoing edges of u, in other words:

$$M_v^{\leftarrow}(u) = \bigvee_{e \in E_G, src_G(e)=u} I_{-e}(M_v^{\leftarrow}(dst_G(e)))$$

As an example let us consider the DFG on the left hand side of Figure 3.6. We assume that the size attribute of every vertex is equal to 32 bits. In this example our objective is to compute $M_{v_5}^{\leftarrow}$. We start with $M_{v_5}^{\leftarrow}(v_5) = \texttt{0xffffffff}$. From v_5 to v_4 the mask is updated as follows. v_5 is labelled with $\wedge$ which is one of the few operations with $\ll$ and $\gg$ to have a specific update mechanism. In the case of $\wedge$, we first over-approximate the result of the bitwise AND over its other operands. Luckily in this example there is only one other operand and it is a constant. To update the influence mask, we simply compute the bitwise AND of $M_{v_5}^{\leftarrow}(v_5)$ and of this constant.

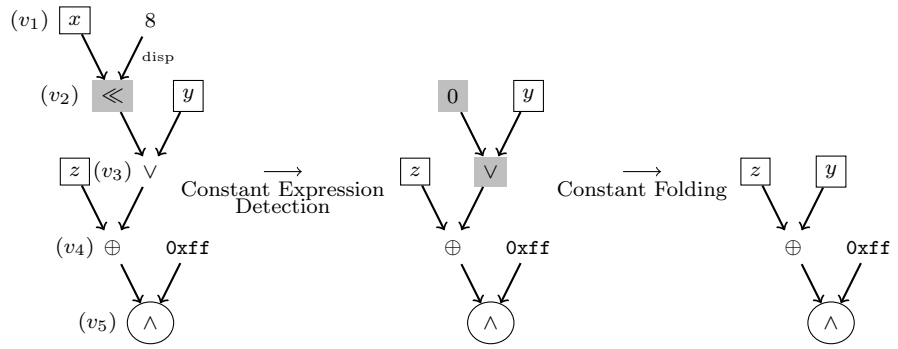

Figure 3.6: Example of a normalization phase that involves a constant expression detection rewrite step. $\{x \wedge M_{v_5}(v_2)$, where $x \in D(v_2)\} = \{0\}$, thus v_2 can be replaced by a 0 constant according to the constant expression detection mechanism. Finally, a constant folding rewrite step simplifies the $\vee$ operation.

v_4 has a single outgoing edge thus $M^{\leftarrow}_{v_5}(v_4) = \texttt{0x000000ff}$. Vertices v_3 and v_4 are treated in similar ways. They are both labelled with bitwise operation symbols that do not have a specific update mechanism. The mask is kept unchanged and $M^{\leftarrow}_{v_5}(v_3) = M^{\leftarrow}_{v_5}(v_2) = \texttt{0x000000ff}$. Finally from v_2 to v_1 the mask is updated as follows. v_2 is labelled with $\ll$ and v_1 is its first operand. In that case we right shift the influence mask by the value obtained from the over-approximation of its second operand: $M^{\leftarrow}_{v_5}(v_1) = \texttt{0x000000ff} \gg 8 = \texttt{0x00000000}$. As a conclusion v_1 has no influence on v_5.

Influence masks are used for constant expression detection and also in Chapter 5 to filter paths that do not reflect actual influence relations.

A rewrite step, which modifies parts of terms that do not influence the set of final vertices, preserves the observable similarity. More formally, given a rewrite rule $r = (L, R, \psi)$, a rewrite step $G \rightarrow_{r,f} H$ preserves the observable similarity if for every assignment θ and every vertex $v \in Root_L$:

$$\theta(term_L(v)) \wedge M^{\leftarrow}_{Root_G}(f_V(v)) = \theta(term_R(\psi(v))) \wedge M^{\leftarrow}_{Root_G}(f_V(v))$$

The influence mask $M^{\leftarrow}_{Root_G}$ is equal to the bitwise OR of the influence masks on each of the vertices of $Root_G$. Applied to constant expression detection, this property states that if there is a vertex v such that $D_G(v) \wedge M^{\leftarrow}_{Root_G}(v)$ contains a single element, we can replace v by a vertex labelled with the corresponding constant without breaking the observable similarity.

This normalization mechanism is illustrated in Figure 3.6. According to value range analysis, $D(v_2) = \{2^8 \times x$, with $x \in [0, 2^{24}]\}$ and according to influence measurement $M^{\leftarrow}_{v_5}(v_2) = \texttt{0x000000ff}$ (refer to the example in the previous paragraph). Thus $D(v_2) \wedge M^{\leftarrow}_{v_5}(v_2) = \{0\}$. We can replace the $\ll$ operation by a 0 constant. Although they share several characteristics, the examples of Figure 3.4 and 3.6 are not equivalent. In fact, in Figure 3.4, the $\vee$ operation can have several outgoing edges. For this reason, its influence mask over the final vertices is not necessarily equal to $\texttt{0x000000ff}$. Hence constant expression detection cannot be applied in that case. Conversely in Figure 3.6 the $\wedge$ operation must be distributed over the $\oplus$ and the $\vee$ operation to obtain a constant term. Such a sequence of distributions is out of the scope of constant distribution.

One may argue that the example of Figure 3.6 is unrealistic since it has very little chance to be encountered in correctly optimized code. Nevertheless, this exact example was found in the OpenSSL AES synthetic sample and motivated by itself the introduction of constant expression detection. Although no software developer would write such an expression in plain C code, it may still appear, as it is the case in OpenSSL, due to side effects of SIMD register manipulation.

If the vertices are sorted in inverse topological order, $M^{\leftarrow}_{Root}$ can be computed for all the vertices in a single pass. Thus, the worst case complexity of this normalization mechanism is $\mathcal{O}(n^2)$ and the average case complexity is $\mathcal{O}(n)$.

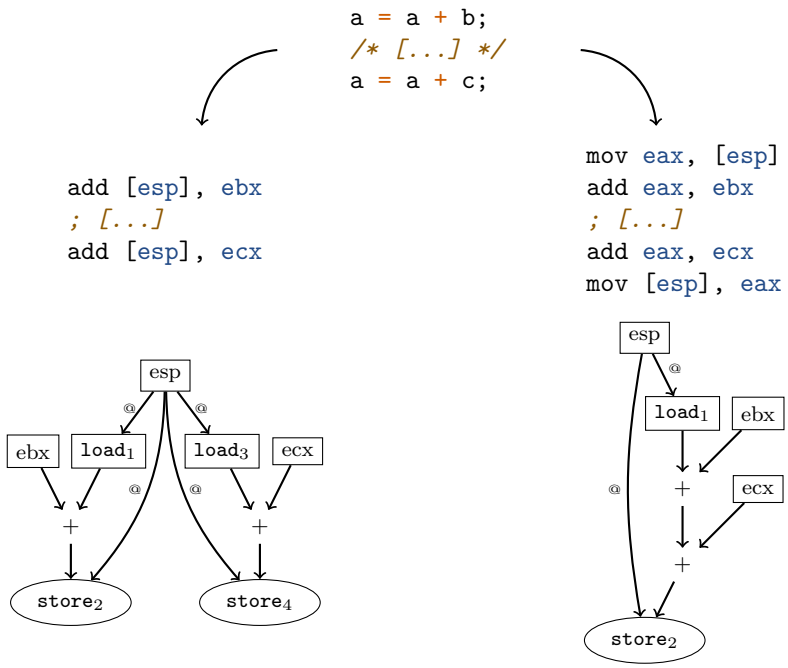

Figure 3.7: C code snippet compiled with two different register allocation strategies, resulting into different DFGs.

3.6 Memory Access Simplification

During construction of DFGs, memory accesses are replaced by `load` and `store` operations. Many of them are caused by register filling and spilling. Register allocation is highly dependent on the compiler and on its optimization level. Thus, these accesses reflect more implementation specificities than a generic characteristic of the algorithm. In the example of Figure 3.7, the same C code snippet is compiled using two different register allocation strategies. On the right the intermediate result is stored in a register whereas on the left it is stored in the stack. The DFGs that would have been obtained for these two strategies (given at the bottom of the figure) are rather different. Normalization must abstract DFGs from the way local variables are stored (either in registers or in memory). Moreover, for mode of operation identification, we will be interested in tracking values as they are written and read from memory (refer to Chapter 5). To fulfil these objectives, we rely on a normalization mechanism called memory access simplification. It tries to remove unnecessary memory operations. That is to say `load` operations, the address of which has already been accessed in the DFG and `store` operations, the address of which will be overwritten without any prior `load`. Ideally, normalized DFGs should be free of any memory operation except those corresponding to input or output variables.

3.6.1 Naive Solution

First we compute sequences of memory operations that have the same vertex for their address operand. Then, we traverse theses sequences and perform simplifications based on the following rules:

Rule 1	$\texttt{store}_1, \texttt{store}_2$	$\rightarrow$	$\texttt{store}_2$
Rule 2	$\texttt{store}_1, \texttt{load}_2$	$\rightarrow$	$\texttt{store}_1$
Rule 3	$\texttt{load}_1, \texttt{load}_2$	$\rightarrow$	$\texttt{load}_1$

In Rule 1, $\texttt{store}_1$ is simply deleted. This rewriting is more related with dead code removal than with a rewrite step. Since the value written by $\texttt{store}_1$ is overwritten by $\texttt{store}_2$, $\texttt{store}_1$ is not a final vertex. It does not have any successor. Thus it can be deleted according to the dead

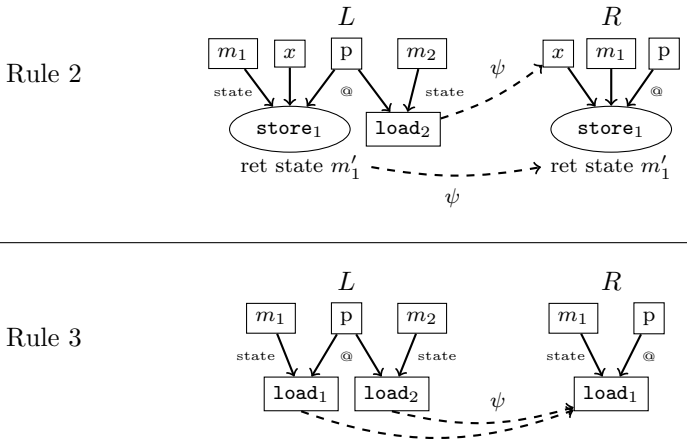

Figure 3.8: DFG rewrite rules resulting from Rule 2 and 3. The memory state, which is the first argument of both `load` and `store` operation, is explicitly depicted.

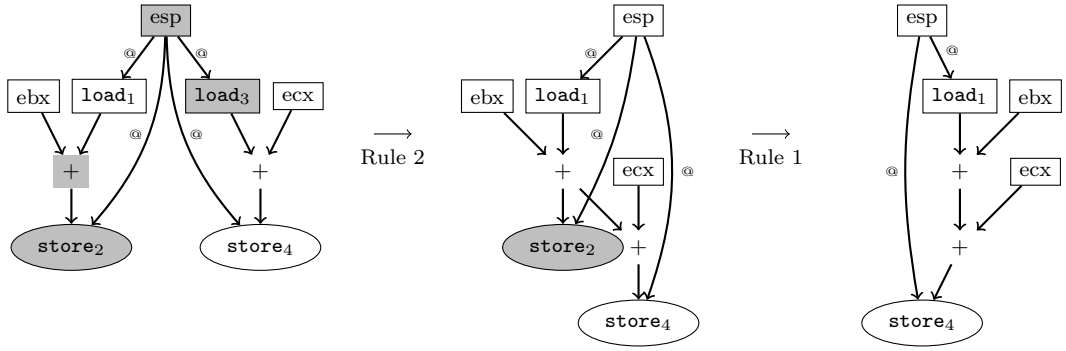

Figure 3.9: Memory simplification applied to the DFG on the left column of Figure 3.7. The simplification process goes from the left to the right.

code removal principle given in Section 2.4.1. In Rule 2, $\texttt{load}_2$ is replaced by the data operand of $\texttt{store}_1$. In Rule 3, $\texttt{load}_2$ is replaced by $\texttt{load}_1$. The DFG rewrite rules that result from Rule 2 and 3 are given in Figure 3.8. Any sequence of memory operations converges either to a single operation or to the sequence: $(\texttt{load}, \texttt{store})$. Defined as such, memory access simplification fulfils our objective: there is at most one `load` per input variable and one `store` per output variable. As illustrated in Figure 3.9, the DFG on the left hand side of Figure 3.7 converges toward the DFG on the right hand side of Figure 3.7. Even with two different register allocation strategies, once normalized the two DFGs are isomorphic.

3.6.2 Aliased Pointers

However, stated as such, Rule 1, 2 and 3 do not always result in observably similar rewrite steps (or in proper dead code removal). A vertex that is an operand of type address for at least one memory operation is called a pointer. Two pointers p_1 and p_2 are aliased if there is an assignment θ such that $\theta(p_1) = \theta(p_2)$. Given the naive solution that has been presented so far, one sequence of memory operations is obtained per pointer. These sequences are simplified independently. It leads to observably non-similar rewrite steps (or non-proper dead code removal) in the following scenarios.

- Rule 1: if an aliased `load` happens between $\texttt{store}_1$ and $\texttt{store}_2$. The value written by $\texttt{store}_1$ is accessed by the aliased `load`. Hence $\texttt{store}_1$ must not be deleted according to the dead code removal principle.
- Rule 2: if an aliased `store` happens between $\texttt{store}_1$ and $\texttt{load}_2$. The value read by $\texttt{load}_2$

might differ from the value that was written by $\texttt{store}_1$. We have no guarantee that the observable similarity still holds.
- Rule 3: if an aliased `store` happens between $\texttt{load}_1$ and $\texttt{load}_2$. The value read by $\texttt{load}_2$ might differ from the value that was first read by $\texttt{load}_1$. We have no guarantee that the observable similarity still holds.

Therefore, we must ensure that there is no problematic aliased memory access before performing memory simplifications. Aliasing is a well known problem in static analysis and in compiler theory. Aliased pointers are common in DFGs representing x86 code, especially at the early stages of the normalization process. For instance, before common subexpression elimination, different vertices may represent the exact same expression. If these vertices are pointers then they will obviously be aliased. To detect aliasing we must be able to compare any two pointers. The result of the comparison of two pointers p_1 and p_2 is either:

- must alias if $\theta(term_G(p_1)) = \theta(term_G(p_2))$ for all assignment θ ;
- cannot alias if $\theta(term_G(p_1)) \neq \theta(term_G(p_2))$ for all assignment θ ;
- may alias if none of the previous properties is true or if we are unable to prove that any of them is true.

For this work two methods have been proposed to compare pointers: a static one based on formal expression simplification and on value range analysis and a dynamic one that relies on concrete address values. They are respectively presented and discussed in Sections 3.6.4 and 3.6.5.

3.6.3 Correct Solution

Let $(Op_i)_{1\leq i\leq l}$ be the sequence of all the memory operations. Iteratively, each element Op_i is processed as follows. The address operand of Op_i is compared with the address operand of Op_{i-1}. If they must alias and if a rule applies to $(labV_G(Op_{i-1}), labV_G(Op_i))$, we perform the corresponding transformation. If they may alias and if Op_{i-1} is problematic with respect to any possible rule that could be applied later on Op_i, we stop. Otherwise, we move to the previous element and we compare the address operand of Op_i to the address operand of Op_{i-2}. We repeat this process until the first element of $(Op_i)_{1\leq i\leq l}$ is reached or until we stop. A pseudo code of this algorithm is given in Algorithm 3. It has been greatly simplified for the sake of clarity. For instance, it does not take the size of memory accesses into account.

The pointer comparison method is used to determine both, when to apply a memory access simplification rule and when to stop the search because a problematic aliased operation has been found. Let Op_i and Op_j be two memory operations such that $j < i$ and the addresses of Op_i and Op_j may alias. If $labV_G(Op_i) = \texttt{store}$ and $labV_G(Op_j) = \texttt{load}$ then Op_j is problematic with respect to Op_i. This case corresponds to the alias scenario associated with Rule 1. If $labV_G(Op_i) = \texttt{load}$ and $labV_G(Op_j) = \texttt{store}$ then Op_j is problematic with respect to Op_i. This case corresponds to the alias scenarios associated with Rule 2 and 3. Once a problematic aliased operation is found, no further simplification are possible for the current memory access. It is important for the comparison method to be accurate and not to conclude too quickly that pointers may alias. Otherwise we will miss many memory access simplifications.

This algorithm requires $\mathcal{O}(l^2)$ pointer comparisons. A single execution is sufficient to perform every possible memory access simplifications. However, the precision of the static pointer comparison method increases during the normalization phase. Therefore, when this comparison method is used, this algorithm will have to be executed regularly during the normalization phase.

3.6.4 Static Pointer Comparison

Given two pointers p_1 and p_2, our goal is to determine whether they can be equal or not at runtime. For this purpose, we over-approximate their difference. That is to say, we search a set Δ_{p_1,p_2}, such that: $\theta(term_G(p_1)) - \theta(term_G(p_2)) \in \Delta_{p_1,p_2}$ for all assignment θ. If $\Delta_{p_1,p_2} = \{0\}$ then (p_1, p_2) must alias. If $0 \notin \Delta_{p_1,p_2}$ then (p_1, p_2) cannot alias. Otherwise (p_1, p_2) may alias. It is important for the precision of the comparison method to find the smallest over-approximated sets possible.

Algorithm 3 Memory Access Simplification

```
for i = 1 to length(Op) do
    for j = i − 1 to 0 do
        compare addr(Op[i]) with addr(Op[j])
        if addr(Op[i]) must alias addr(Op[j]) then
            if labV_G(Op[i]) = store and labV_G(Op[j]) = store then
                apply Rule 1
            else if labV_G(Op[i]) = load and labV_G(Op[j]) = store then
                apply Rule 2 and break
            else if labV_G(Op[i]) = load and labV_G(Op[j]) = load then
                apply Rule 3 and break
            end if
        else if addr(Op[j]) may alias addr(Op[j]) and labV_G(Op[i]) ≠ labV_G(Op[j]) then
            break
        end if
    end for
end for
```

We consider differences between pointers because it gives us the opportunity to perform formal simplifications before computing over-approximations. Another possibility would be to compute directly and independently over-approximations for each pointer and to reach a conclusion based on the intersections of their over-approximated sets of reachable values. However, this latter solution would be much less effective. Let us consider for instance the two pointers defined as follows: $p_1 = +(\texttt{eax}, 10)$ and $p_2 = \texttt{eax}$. Without any additional information on `eax`, the best over-approximations we can compute leads to: $D(p_1) \cap D(p_2) = \{0,1\}^{32}$. Consequently, we can only conclude that (p_1, p_2) may alias which is correct but unsatisfactory. If we consider instead the difference between the two pointers, say the subtraction of p_2 from p_1, we obtain the following expression: $-(+(\texttt{eax}, 10), \texttt{eax})$. This expression can be simplified using the following term rewrite rule: $-(+(x_1, x_2), x_1) \to x_2$. Thereby we can establish that $-(p_2, p_1) = 10$ and we finally obtain: $\Delta_{p_1,p_2} = \{10\}$. With this comparison method we are able to conclude that (p_1, p_2) cannot alias.

The initial idea to simplify expressions resulting from pointers subtraction, was to reuse some of the rewrite rules that have been implemented for the normalization phase. Affine expression simplification (refer to Section 3.9) seemed particularly appropriate for this purpose. Due to performance reasons we limit the size of pointer expressions that are compared. Subtraction and simplification of large pointer expressions may not be worth the computing cost. It can be explained as follows. First, the complexity of pointer expressions may decrease during the normalization phase. It seems more efficient to wait for pointer expressions to be normalized before trying to simplify their difference. Otherwise, one will have to perform some internal rewritings that would have been taken care of by the normalization process anyway, every time one wants to compare two pointers. Second, it is not because one considers larger pointer expressions that one would be able to perform more simplifications while subtracting them. Therefore, we limit the depth and also the operation symbols that can be involved in expressions that represent pointers during alias analysis. Unfortunately the simplification of these partial expressions cannot be achieved through affine expression simplification and requires a specific algorithm.

Definition 9 (Partial Additive Tree)**.** Let v be a vertex in a DFG G and Y a sufficiently large set of variable symbols: $|Y| \geq |V_G|$. A partial additive tree rooted by v is a DFG T over Y and $\{+\}$ such that:

- T is a tree (it has a single root element and $\text{outdegree}_T(u) \leq 1$ for all $u \in V_T$) ;
- there is a morphism f from T to G such that the root of T is mapped to v.

A pointer p is represented by the largest partial additive tree rooted by p, the maximum depth of which is capped at a fix amount. This latter limit is mostly important during the early stage of the normalization process, when very high additive expressions may be encountered (induction variables for instance). The largest partial additive tree is extremely simple to compute. We start

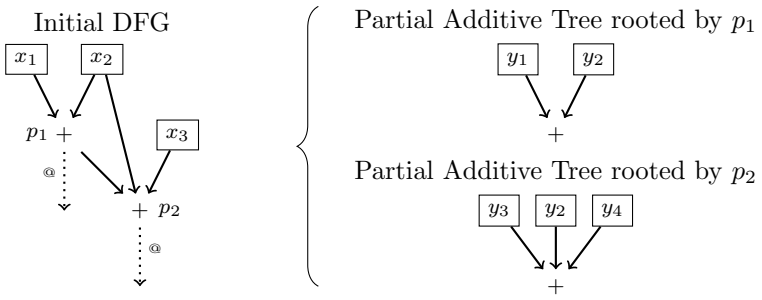

Figure 3.10: Example of two partial additive trees. The maximum depth is 1.

from the root element. If the current vertex is labelled with a + and if the maximum depth has not been reached, we insert it in T and recursively process each of its direct predecessors. Otherwise, insert in T a new vertex labelled with a variable symbol that must be unique for each vertex of G. Examples of partial additive trees can be found in Figure 3.10.

Partial additive tree comparison is more subtle than classical expression simplification. Due to the depth limit, sums represented by partial additive trees may correspond to only one part and not the totality of the original sums represented in DFGs. For this reason, leaves of a partial additive tree can correspond to any type of vertices in another partial additive tree. This scenario is depicted in Figure 3.10. The image of p_1 in the partial additive tree at the top right is the root element whereas its image in the partial additive tree at the bottom right is a leaf (either y_3 or y_4). As a consequence, the difference between two partial additive trees is not equal to the difference of their leaves.

Let p_1 and p_2 be two pointers, T_1 and T_2 be the largest partial additive trees rooted by p_1 and p_2 respectively and let f_1 and f_2 be two morphisms from T_1 (respectively T_2) to G which satisfy the condition given in the definition of a partial additive tree. We devised a specific algorithm to compare T_1 and T_2. It returns Δ_{p_1,p_2}. Its pseudo code is given in Algorithm 4.

Algorithm 4 Partial Additive Tree Comparison

tag every vertex in T_1 and T_2 with no predecessor
for all tagged vertices v_1 in T_1 **do**
  if there is a vertex v_2 in T_2 such that $f_{1,V}(v_1) = f_{2,V}(v_2)$ **then**
    remove v_1 from T_1
    remove the subtree rooted by v_2 from T_2
  end if
end for
for all tagged vertices v_2 in T_2 **do**
  if there is a vertex v_1 in T_1 such that $f_{2,V}(v_2) = f_{1,V}(v_1)$ **then**
    remove v_2 from T_2
    remove the subtree rooted by v_1 from T_1
  end if
end for
initialise Δ as the zero singleton
for all tagged vertices v_1 left in T_1 **do**
  $\Delta \leftarrow \Delta + D_G(f_{1,V}(v_1))$ ▷ the addition is an operation on RICs
end for
for all tagged vertices v_2 left in T_2 **do**
  $\Delta \leftarrow \Delta - D_G(f_{2,V}(v_2))$ ▷ the subtraction is an operation on RICs
end for
return Δ

Limitations. Of course, the precision of this static pointer comparison method, can benefit from many refinements. For instance, partial additive trees are not yet capable of simplifying pointer

expressions which include multiplication or left shift operations. These operations are extremely frequent in pointer expressions. For instance, to access an element in a buffer its index has to be multiplied by the size of an element. However, we did not address this particular issue. The reason is that the main limitation of static pointer comparison has nothing to do any more with the precision level of the comparison method. As motivated in Section 2.2.3 our identification methods (for both primitives and modes of operation) are only executed on small trace segments. In the following paragraph we explain why lack of context information greatly reduces the efficiency of static pointer comparison regardless of its own precision level.

If the initialisation of a pointer (or even the initialisation of a subterm of a pointer expression) happens before the beginning of the trace segment, it will be represented by a variable symbol. Unfortunately, no tight over-approximation can be made for a variable symbol and it cannot be simplified with any other expressions that do not dependent on it. Therefore, this pointer will automatically become a possible alias for any other pointers that are not connected with the same variable symbol. For primitive identification, we are able to cope with this issue thanks to a highly arguable heuristic presented in the next paragraph. But for mode of operation identification it is not possible. Modes of operation manipulate many data buffers (at least one for the plaintext, one for the ciphertext, one for the key and one the nonce) the addresses of which are usually defined outside of the analysis window. These buffers are accessed with mixed `load` and `store` operations. In this situation, very few memory access simplifications remain possible. Unfortunately mode of operation identification is extremely dependent on memory access simplification (refer to Section 5.3.1). Therefore, for mode of operation identification, we use dynamic pointer comparison instead of static pointer comparison.

ESP Heuristic. To introduce the `esp` heuristic we rely on a very simple primitive model. According to this model a primitive interacts with three groups of pointers. The first group contains pointers to input data. Input data is accessed at the beginning of the primitive by `load` operations. The second group contains pointers to output data. Output data is accessed at the end of the primitive by `store` operations. The third group contains stack pointers. Stack data is accessed throughout the primitive by both `load` and `store` operations. There is no problematic alias access inside each of these groups: the first one contains only `load` operations, the second one contains only `store` and since every pointer in the third group derive from `esp` we suppose that the comparison method will eventually return either must or cannot alias. There is no problematic aliased access between the first and the second group, since one happens before the other. However, there are plenty of problematic aliased accesses between the first and the third group and between the second and the third group.

The `esp` heuristic relies on the following intuition. The current stack frame is not supposed to overlap buffers that are allocated in other stack frames, in the heap or in any other memory region. We also assume that pointers that are used to access the current stack frame, depend on `esp` and that pointers that are used to access other memory regions, do not depend on `esp`. Consequently, a pointer that is reachable from a vertex labelled with `esp` cannot alias a pointer that is not reachable by any vertex labelled with `esp`.

Of course this heuristic is highly arguable. It may be ineffective: it may conclude that two pointers may alias even though in reality they cannot. And more preoccupying, it may be unsafe: it may conclude that two pointers cannot alias even though in reality they can. This is the case for instance if `ebp` is used to the access the current stack frame and if the link between `ebp` and `esp` does not appear in the trace segment. A solution to mitigate this particular issue is to always start trace segments at the beginning of a function.

However, the `esp` heuristic is a necessary evil. It solves completely the aliasing issues we had in our simple primitive model. In fact, according to the `esp` heuristic, the third group of pointers cannot alias the two other groups. Furthermore for reasons that are detailed in the following section, static pointer comparison is the only possible pointer comparison method for primitive identification.

3.6.5 Dynamic Pointer Comparison

For a given execution, we record in the execution trace the value of the address of every memory access. Those address values are then attached to memory operations during the DFG construction. To compare the addresses of two memory operations, we just compare their runtime values.

Obviously, the results returned by this comparison method are only valid for a single execution, that is to say, for a single assignment. If this comparison method is used for memory access simplification, we will probably lose the observable similarity property. The normalized DFG will be observably similar to its original form only with respect to the set of assignments that do not modify the relative ordering of memory addresses. This approximation is acceptable on condition that this set of assignments is somehow representative of every possible execution. In other words, it is acceptable if collisions between memory addresses remain more or less the same from one execution to another.

We assume that this hypothesis is valid for modes of operation. Two types of input data may be subject to variations: the address of memory buffers and the content of memory buffers. Here, by memory buffers, we mean cryptographic parameters such as the plaintext buffer or the key buffer for instance. The addresses of memory buffers vary if they are, for instance, allocated dynamically on the heap. We suppose that if they do not overlap for one execution they will never overlap and vice versa. We also suppose that the content of memory buffers is not used to compute new memory addresses. In fact any complex transformation performed on the content of memory buffers can be seen as a distinct primitive and be dealt with separately.

We cannot make the same assumption for primitives. In fact, in primitives the content of memory buffers can be used to compute new memory addresses. Substitution boxes are a perfect example. If for a given set of cryptographic parameters, the same location in a substitution box is read twice, Rule 3 will apply. As a consequence any expression involved in the computation of the second address will be deleted according to dead code removal principle and an edge will be inserted between the first and the direct successors of the second `load`. These are important modifications and they will prevent the creation of a generic signature that would be able to match every possible execution. For this reason, we cannot use dynamic pointer comparison for primitive identification.

Another disadvantage of dynamic pointer comparison is that the program really needs to be executed in a monitored environment. We can imagine several techniques to obtain straight line code without having to execute the program. Some primitives are implemented as a single basic block and hence, they already satisfy the straight line code hypothesis. In C code, function inlining is not always a difficult problem. We can enumerate every possible execution path and use external methods to filter execution paths that are clearly impossible. We do not claim that it is always possible to obtain, at a small cost, straight line code that is representative of a real program execution. But still, we have good hopes that in certain scenarios, where monitoring a program execution is difficult (privileged code for instance), the straight line code hypothesis could be satisfied without having to really execute the program. This is no longer the case with dynamic pointer comparison. Moreover, in practice, recording address values increases the size of execution traces and the time required to collect them, by at least one order of magnitude.

Limited Use of Address Values. In a classical analysis scenario we collect a single execution trace for both primitive and mode of operation identification. As a consequence, address values are available during primitive identification even though we cannot use them directly for pointer comparison. In this paragraph we describe two strategies, that are compatible with primitive identification, to exploit these runtime values.

- A first strategy is to use address values to determine if two pointers may alias. As explained in Section 3.6.4, the `esp` heuristic might erroneously conclude, in specific situations, that two pointers cannot alias whereas in reality they can. To reduce this risk, we can systematically check address values when the `esp` heuristic concludes that two pointers cannot alias. If the address values are equal, we change the result to may alias. Thereby, we correct the `esp` heuristic using runtime values. This strategy only deals with a corner case and it has a limited effectiveness.

- A second strategy is to use address values to determine if two pointers cannot alias. If their runtime values are not equal, then we automatically conclude that they cannot alias. Notice that to conclude that two pointers must alias, we still have to prove it using partial additive trees. This strategy is much more aggressive than the previous one and it clearly breaks the observable similarity. That being said, it is also very effective and it solves every aliasing conflict (that is to say, simplifications that are being blocked because we are unsure whether a possible alias exists or not). Let us apply this strategy to the substitution box example which caused the dynamic address comparison method to be rejected in a first place. To conclude that two accesses are made at the same location in a substitution box, we must be able to prove that their addresses are equal under any assignment. If this is untrue, we will not succeed and no memory access simplifications will ever be performed inside the substitution box. This strategy does not have the same problem as dynamic pointer comparison. At the same time, accesses to the substitution box do not interfere with other memory accesses because we can determine that they cannot alias based on their runtime values. This strategy is both conservative when it has to conclude that two pointers are equal and speculative when it has to reject possible alias.

We have implemented and tested these two strategies. The experimental results presented in Chapter 4 are only related with primitive identification. For these experiments we relied only on what was strictly necessary (that is to say, additive tree comparison and the `esp` heuristic) and we did not use any runtime address value. The experimental results presented in Chapter 5 cover both primitive and mode of operation identification. For these experiments we use simultaneously the two strategies. These two sets of experiments can easily be compared since they contain more or less the same primitive implementations. We did not notice much change though. Results were already good for static pointers comparison so there was not much room for improvement. The good news is that the second strategy, which is unsafe in theory since it breaks the observable similarity, does not lower the detection rate.

3.6.6 Conclusion

To conclude this section on memory access simplification, we summarize the main reasons that led us to choose different pointer comparison methods for primitive and mode of operation identification. We cannot use dynamic pointer comparison for primitive identification because of substitution boxes. The alternative is static pointer comparison. To improve the precision and reduce the complexity of static pointer comparison we use partial additive trees. But the most serious limitation of static pointer comparison comes from lack of context information. The `esp` heuristic is a possible workaround. It has multiple flaws and it only works in a simple primitive model. Another workaround is to make a limited use of runtime address values (if they are available) to decide, for instance, if two pointers cannot alias. For reasons that have not been revealed yet, mode of operation identification is extremely dependent on memory access simplification. At the same time, from a memory access perspective, modes of operation are much more complex than primitives. As a result, aliasing conflicts are extremely frequent and the `esp` heuristic is ineffective. Therefore, we have to use dynamic pointer comparison for mode of operation identification. It has an important impact on the usability of the method, since the program to analyse definitely has to be executed in monitored environment.

3.7 Memory Coalescing

To introduce memory coalescing, let us consider the following example. There are several possibilities to access 32 bits of data in memory. One can use, for instance, a single 32-bit memory operation or four 8-bit memory operations. This choice is usually made by the software developer even though some compiler optimizations are still able to modify it afterwards. There is no real argument from a performance perspective to prefer four 8-bit operations over a single 32-bit operation. But in practice, both versions can be encountered in different implementations of the same

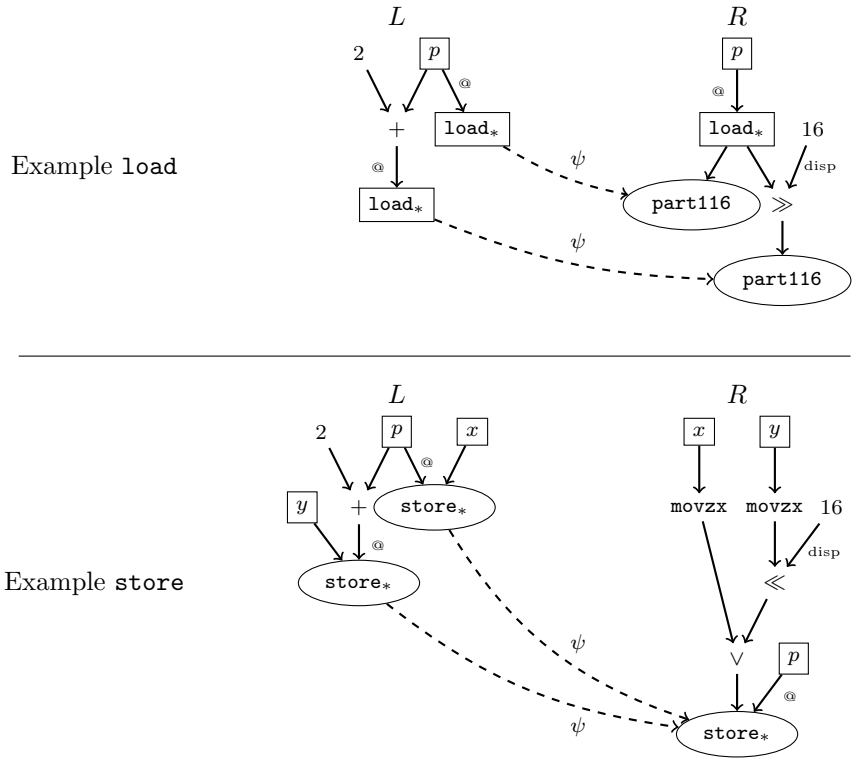

Figure 3.11: Examples of memory coalescing rewrite rules. The first rule at the top combines two 16-bit `load` into a single 32-bit `load`. The second rule at the bottom combines two 16-bit `store` into a single 32-bit `store`.

algorithm. For instance, to modify the endianness of a memory buffer[1], depending on the implementations, memory is accessed either by 8-bit reads or by 32-bit reads. From a normalization point of view, it is important to remove this difference.

We assume that the normalized size for memory operations is equal to the architecture word size. Larger memory operations should be fragmented and smaller memory operations should be combined. Fragmentation happens exclusively during the DFG construction when SIMD instructions are split. Combination is done through a set of rewrite rules called memory coalescing.

A memory coalescing rewrite rule replaces a set of memory operations, which access adjacent memory locations, by a single memory operation the size of which is equal to the sum of the sizes of the memory operations of the set. Examples of memory coalescing rewrite rules are given in Figure 3.11. Two rewrite rules are depicted, one for `load` operations and one for `store` operations. They both combine two 16-bit operations into a single 32-bit operation. Similar rewrite rules exist to combine every set of memory operations such that the sum of their sizes is equal to 32 bits.

A possibility to find sets of memory operations that access adjacent memory locations, is to reuse the pointer comparison methods that were presented in Section 3.6. But we did not do it, mostly for historical reasons. At the time when memory coalescing was implemented, the static pointer comparison method was much less efficient and generic than it currently is. In particular, it was not able to determine if there was a fixed offset between two memory addresses. We devised a specific method to determine sets of adjacent memory operations, the pseudo code of which is given in Algorithm 5. This method is rather basic. It only considers memory operations the address of which is the sum of two terms: one which is constant, called the *offset*, and one which is

[1]Endianness conversion is a common operation in cryptographic implementations, at least on little endian architecture. Some primitives have to be implemented with a particular endianness. For instance, SHA1 has to be implemented in big endian. But surprisingly, some primitives that do not have any endianness constraint, are still implemented in the inverse endianness. This is the case for some AES table implementations [49] that are in big endian even though the architecture is little endian. This mistake is so common that FindCrypt2 [34] only includes the big endian tables of AES.

not, called the *base*. If two memory operations have the same *base* then it is easy to see that their addresses differ by a fixed amount which is equal to the difference of their *offsets*. In Algorithm 5 two important points are mentioned: scheduling and alignment. They will be detailed in the followings paragraphs.

Algorithm 5 Memory Coalescing

initialise S as an empty set
for all v in V_G such that $labV_G(v)$ is either `load` or `store` **do**
  let addr(v) be the address operand of v
  if $labV_G$(addr(v)) $= +$ **and** addr(v) has two operands one of which is a constant **then**
    insert v in S
    let off(v) be the value of the constant operand of addr(v)
    let base(v) refer to the non-constant operand of addr(v)
    if base(v) is an address operand for a vertex u **and** $u \notin S$ **then**
      insert u in S and define off(u) $= 0$ and base(u) $= v$
    end if
  end if
end for
while there is an unprocessed vertex v in S **do**
  $S_v \leftarrow \{u \in S$ such that $labV_G(u) = labV_G(v)$ and base(u) $=$ base(v)$\}$
  for all sequences $(u_1, ..., u_n)$ of vertices of S_v such that:
    - $\sum_{1 \leq i \leq n} \text{size}_G(u_i) = 32$
    - $\forall i < n$, $\text{size}_G(u_i) = \text{off}(u_{i+1}) - \text{off}(u_i)$

do
    if $(u_1, ..., u_n)$ satisfies the scheduling and the alignment condition **then**
      coalesce $(u_1, ..., u_n)$ and **break**
    end if
  end for
  tag every element of S_v as processed
end while

Memory coalescing rewrite rules are the only rewrite rules which try to vectorize similar operations. It would make sense to create rewrite rules to vectorize other types of operations, such as bitwise operations for instance. But since we have not encountered during our experiments any scenario that requires such rules, we did not implement them.

Scheduling. Let us consider a memory coalescing rewrite step which replaces the memory operations op_1 and op_2 by $op_{1\|2}$. Let i_1 and i_2 be the position of op_1 and op_2 in the sequence of memory operations. Let j be the position where $op_{1\|2}$ is scheduled. We assume without losing generality that $i_1 < i_2$. Let us first consider the simple case where $i_2 = i_1 + 1$ and $j = i_1$. In that case the rewrite step preserves the observable similarity. In fact, the sequence of memory operations from 0 to j is unchanged, hence the memory state at position j is the same before and after the rewrite step. Consequently, if op_1 and op_2 are `load` operations, the result returned by $op_{1\|2}$ will be equal to the concatenation of the results returned by op_1 and op_2. And if op_1 and op_2 are `store` operations, the memory state returned by $op_{1\|2}$ will be equal to the memory state returned by op_2. Thus, the following memory operations will not be affected. Now, let us consider the general case. In the general case, the rewrite step does not preserve the observable similarity. For instance in the following assembly code snippet, if we apply the rewrite rule given at the top of Figure 3.11 to the first and the last `load` operation and if we schedule the result at the beginning of the sequence of memory operations, we will break the observable similarity.

```
mov ax, [esp]          ; op1, i1 = 0
mov [esp + 0x2], bx
mov cx, [esp + 0x2]    ; op2, i2 = 2
```

To detect whether or not a rewrite step breaks the observable similarity we proceed as follows. If we can move, in the sequence of memory operations, op_1 from i_1 to j and op_2 from i_2 to $j+1$, then we obtain the simple case that was first detailed. We showed, in that case, that the observable similarity is preserved. Moving a `load` operation from position i to j preserves the observable similarity if there is no `store` operation which accesses the same memory location between i and j. Otherwise the `load` operation may return different results if placed at position i and j since the memory state is different at those two positions. Moving a `store` operation from position i to j preserves the observable similarity if there is no `load` or `store` operation which accesses the same memory location between i and j. In fact, moving a `store` operation across a `load` operation may modify its memory state argument and, consequently, the `load` operation may return a different result. And moving a `store` operation across another `store` operation, may modify the memory state at the end of the sequence of memory operations. Thus, it clearly breaks the observable similarity. To test these conditions we reuse the static pointer comparison method that is presented in Section 3.6. To determine where we should place $op_{1\|2}$, we must find a position j such that moving op_1 and op_2 from their original position to j, (respectively $j+1$) preserves the observable similarity. For instance, in the previous assembly code snippet, we can swap the op_1 with the next memory operation (they do not access the same memory location). Thus, $op_{1\|2}$ must be placed at the end of the sequence of memory operations.

In practice, to limit the complexity of finding the right position to schedule $op_{1\|2}$, we only test a small number of positions which are equal to the positions of the memory operations that are to be replaced. That is to say, we test $j = i_1$ and $j = i_2$. Even though none of them preserves the observable similarity, we stop the analysis afterwards[2].

Alignment. Memory coalescing rewrite rules are terminating (every rewrite step reduces the number of memory operations by at least one) but they are not convergent. Let us consider the following assembly code snippet:

```
mov ax, [esp]
mov bx, [esp + 0x2]
mov cx, [esp + 0x4]
```

Two rewrite steps are possible for the rewrite rule given at the top of Figure 3.11. Either we combine the first two `load` operations or the last two `load` operations. Different normal forms are obtained for each of them. To solve this convergence issue we choose to only rewrite sets of memory operations the smallest address of which is aligned. A memory address p is said to be aligned if for every assignment θ, $\theta(p)$ is a multiple of the architecture word size. We choose to align the smallest address because it will become the address of the new memory operation. Assuming that there is an implementation in which memory operations already have the size of an architecture word, their addresses are most likely to be aligned. Therefore, if we have to limit memory coalescing rewrite steps, it seems a good idea to only keep those that will produce aligned memory operations. In the code snippet above, the smallest address of the two possible sets of memory operations is respectively `esp` and `esp` $+ 2$. If we assume that `esp` is always aligned on 32 bits then only the first address is aligned on 32 bits. Hence, according to the alignment condition, only the first two `load` operations can be rewritten. If we suppose that there is at most one memory operation of each type per memory location (which is the ideal result of memory access simplification), this alignment constraint solves the convergence issue. A memory operation can only be combined with the memory operations that access the same aligned word.

The difficult question is how to determine whether or not an address is aligned. A first method is to use runtime address values when they are available. Notice that here, using runtime address values does not break the observable similarity. In fact, rewrite steps preserve the observable similarity regardless of the alignment condition. But, different executions of the same code may lead to different normal forms. We devised a second method that does not depend on runtime address values. This method is not strictly accurate and reliable. But since there is no risk to

[2]Note that there is no need to test positions which are before the first operation or after the last operation that has to be replaced. In fact, if it is not possible to move the other memory operations up to those positions, it will neither be possible to move them beyond.

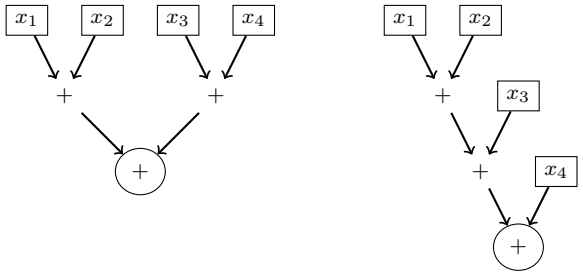

Figure 3.12: Example of two DFGs over $\{+\}$ and $\{x_1, x_2, x_3, x_4\}$ which are observably similar but do not converge to the same normal form.

break the observable similarity it is better than nothing. As previously mentioned we only consider addresses which are a sum of two terms: one called *offset* which is constant and one called *base* which is shared by several other addresses. We assume that *base* terms are always aligned. Thus, an address is aligned if and only if, its *offset* is a multiple of the architecture word size. This condition can be checked very easily. The arguable part concerns the alignment assumption on the *base* term. Because of other rewrite rules (mostly constant folding and constant merging), the *base* term is not, at least at the end of the normalization phase, a sum that involves a constant term. For this reason, it differs from the other, possibly misaligned, memory addresses. In a simple memory access model, where every address is equal to a buffer address plus a fixed displacement (this is the exact memory model that is being targeted by our implementation of memory coalescing), we have every reason to believe that the *base* term corresponds to a buffer address. And it makes sense to assume that memory buffers are aligned.

3.8 Commutative and Associative Operation Normalization

Let *op* be the symbol of a commutative and associative operation. Let G be a DFG of $\mathcal{G}_{\{op\},X}$ such that G has a single root vertex. Because of the commutativity and associativity properties of *op*, the vertices of G labelled with variable symbols can be reorganized in many configuration of subterms without breaking the observable similarity[3]. These reorganizations may strongly affect the structure of DFGs. In particular, it is possible to find DFGs which are reorganizations of one another and thus observably similar, but which do not converge towards the same normal form, with the set of rewrite rules that we have described so far. This situation is illustrated by an example in Figure 3.12. We used $\mathcal{G}_{\{op\},X}$ to introduce this normalization issue, but it also clearly affects $\mathcal{G}_{\Sigma,X}$. Given a DFG G in $\mathcal{G}_{\Sigma,X}$, there is a normalization issue for every part of G which is isomorphic to an element of $\mathcal{G}_{\{op\},X}$ and which is sufficiently large to allow multiple reorganizations. In practice we faced this issue for $+$, $\vee$ and $\oplus$.

The solution we come up with to normalize elements of $\mathcal{G}_{\{op\},X}$, is to merge their internal vertices (an internal vertex is a vertex which is neither final nor labelled with a variable symbol). Instead of having a cascading set of *op* vertices, each of them having exactly two input operands, we have a single vertex with multiple input operands. The structure of the internal vertices is totally hidden inside this new operation. This transformation can be implemented using the following rewrite rule:

$$op(op(x_1, ..., x_n), y_1, ..., y_m) \rightarrow op(x_1, ..., x_n, y_1, ..., y_m)$$

This rewrite rule is a generalization of the rewrite rule called constant merging which was presented in Section 3.4.3. This rewrite rule is terminating and convergent but not context-insensitive. As explained in the first section of this chapter, a rewrite rule which is not context-insensitive can prevent rightful signature detections. This is the case if parts of an expression which was supposed to match a signature, are combined with surrounding expressions resulting in new expressions which could not have been anticipated at the time the signature was created. As we said previously, the

[3]The observable similarity is preserved as long as, for each variable symbol x in G, the number of paths between vertices labelled with x and the root vertex stays unchanged.

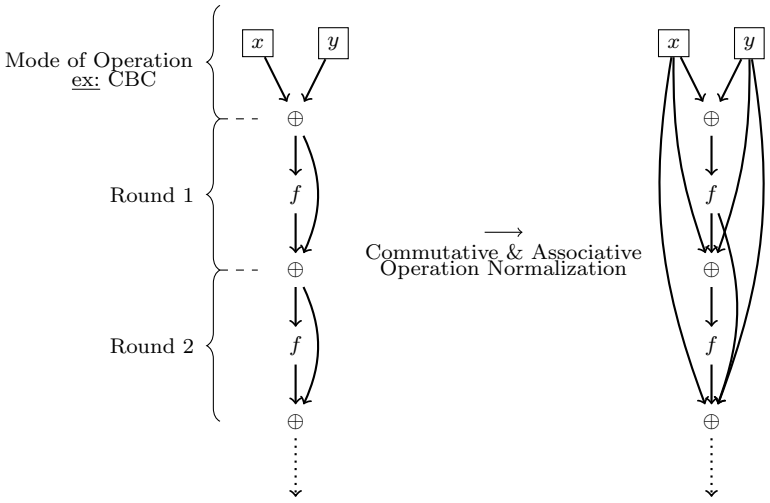

Figure 3.13: A DFG and its normal form. The distinctive structure of the round function which is clearly visible in the initial DFG on the left, was destroyed by commutative and associative operation merging during the normalization phase.

large majority of the rewrite rules presented in this chapter does not meet the strict context-insensitivity definition given in Section 3.1.2. In practice though, lack of context-insensitivity does not affect too much the primitive identification method. Unfortunately this last remark does not apply to the rewrite rule given above. An example is given in Figure 3.13. The DFG on the left represents two executions of a toy round function defined by $x \mapsto x \oplus f(x)$, where f denotes an unspecified function. It also contains at the top expressions corresponding to a piece of code that was executed before the round function. This piece of code might belong to the implementation of a mode of operation such as Cipher Block Chaining (CBC) for instance. To detect the round function we create a signature representing the expression: $\oplus(x, f(x))$. This is a sensible choice since this pattern appears twice (once per round) in the DFG on the left. But its normal form on the right, does not contain any subgraph which matches this pattern. The first round was mixed with expressions of the mode and the second round was mixed with expressions of the first round.

Lack of context-insensitivity is particularly harmful for commutative and associative operation merging, because it affects operations ($+$, $\vee$ and $\oplus$) which are frequent in cryptographic code and especially on signature boundaries (first and last iteration of an add round key for instance). To overcome this issue, we limit the situations in which we perform commutative and associative operation merging. Let u and v be two vertices of a DFG G such that there is an edge from u to v and both u and v are labelled with the same symbol of a commutative and associative operation. We merge u and v only if the following conditions are satisfied.

u must have no other direct successor than v. To show the interest of this condition, let us assume that u has another direct successor, called w. Let S be a signature and f be a matching of this signature in G, that is to say, an injective morphism from S to G. Let us assume that $u, v, w \in f_V(V_S)$ and that $labV_S(f_V^{-1}(u)) \in X$. This last assumption means that u belongs to the image of the boundary of S. The label of u is not specified by S, but our detection method must work whatever the label of u is. Let us first assume $labV_G(u) \neq labV_G(w)$. According to the commutative and associative operation rewrite rule, we merge u and v but not u and w. The newly created vertex and w do not have any common direct predecessor. Thus, there is no injective morphism from S to the normal form of G. This is the exact situation that is being illustrated in Figure 3.13. The case where $labV_G(u) = labV_G(w)$ is also problematic. We call $v \circ u$ (respectively $w \circ u$) the vertex resulting from the merging of u and v (respectively u and w). $v \circ u$ and $w \circ u$ share more direct predecessors than v and w did. There are more possibilities to map the vertex of S that was at first mapped to u. It induces parasitic signature detections which are problematic.

But in practice this condition is a little bit too strict. There are situations where we still want to merge commutative and associative operations even if some of their intermediate results

are reused by other expressions. For instance let us consider a DFG that computes the exclusive or of four variables. If, because of register allocation, an intermediate result is stored on the stack, then the vertex corresponding to that intermediate result will have two direct successors: the next $\oplus$ operation and a `store`. Consequently, it will not be possible to merge the concerned vertices. To allow more flexibility, we introduced several exceptions to the condition presented in this paragraph. These exceptions concern the $+$ and $\oplus$ operations. Typically, they authorize merging if the other direct successor(s) satisfy additional criteria. Merging vertices which have several direct successors, is in conflict with Rule 2 and 3 of common subexpression elimination presented in Section 3.3. To avoid creating a non-terminating reduction system, we exceptionally perform those rewrite steps at the end of the normalization phase.

u and v must originate from the same function execution. We introduced this condition in a first place to cope with AES CBC. The simplest way to analyse mode of operation is to consider trace segments which contain several executions of the primitive. In the case of AES CBC these trace segments contains the XOR operation of CBC followed by the XOR operation of the first add round key of AES. If these two XOR operations are merged, the identification of AES will be imprecise (we will not be able to pinpoint its input parameters). To separate expressions of the primitive from expressions of the mode of operation, we introduce the notion of function. In fact, primitives are usually implemented in a single function which is different from the function which implements the mode of operation[4]. To determine whether or not u and v originate from the same function execution, we proceed in two steps. First, we map u and v back to an index in the sequence of instructions. This conversion from vertices to dynamic instructions also plays a key role to exploit the result returned by the primitive identification method. It will be discussed in more details in Section 6.1.4. Second, we count the number of `call` and `ret` instructions between u and v. A counter is incremented for every `call` instruction and decremented for every `ret` instruction. If the counter falls below zero or if its final value is different than zero, u and v do not occur in the same function execution.

3.9 Affine Expression Simplification

As suggested by its name, this normalization mechanism tries to simplify affine expressions. Here, we only consider expressions which are affine with respect to the addition and the multiplication over $\mathbb{Z}$. An affine expression of the form $+(c_0, \times(c_1, x_1), ..., \times(c_n, x_n))$ can be simplified if there are i and j such that $x_i = x_j$. If that is the case, affine expression simplification will replace this expression by $+(c_0, \times(c_1, x_1), ..., \times(c_i + c_j, x_i), ..., \times(c_n, x_n))$. However, affine expressions in DFGs are usually not formatted as neatly as the expression above. They are made of an arbitrary number of nested subterms involving operation symbols taken from $\{+, \times, \texttt{imul}, \texttt{neg}, \ll, -\}$. For this reason it is hard to devise a set of rewrite rules that will be able to simplify any affine expression. Instead, we have implemented affine expression simplification through a custom algorithm. This algorithm is made of three steps which are detailed below.

1. Given as input a vertex u, this step returns a set of tuples (c_i, u_i) where c_i is a constant and u_i is a vertex. This set is computed as follows. We recursively traverse the DFG, starting from u, using a backward Depth First Search (DFS) algorithm. As we traverse the graph, we use a variable noted c to keep track of the constant which multiples the subterm represented by the current vertex. If the current vertex corresponds to an addition, a subtraction or a multiplication by a constant, we update the variable c and recursively process its direct predecessors (those which are not labelled with a constant). Otherwise we add to the set the tuple made of the current value of c and the current vertex.
2. Every pair of tuples which contains the same vertex, is replaced by a new tuple the first element of which is equal to the sum of the first elements of the pair.
3. If at least one simplification was performed during the previous step, we replace u in G with a DFG representation of the set of remaining tuples.

[4]Actually we found some exceptions to this observation during our experiments. They are discussed in Section 4.3.3.

This algorithm should not be mistaken for the partial additive tree comparison algorithm presented in Section 3.6.4. Here the maximum depth of the graph traversal is unbounded. Thus it makes sense to only compare the topmost vertices since they will eventually be reached regardless of the size of the different paths that lead to them. Complexity is not too much of an issue since, unlike static pointer comparison, this algorithm is only executed a number of times which is linear in the number of vertices. Unsurprisingly, one of the main motivation of affine expression simplification is to simplify pointer expressions so that more precise static pointer comparisons can be performed. As far as we are aware, there is no termination, convergence or practical context-insensitivity issue with this normalization mechanism.

3.10 Operation Size Expansion

Operation size expansion is a normalization mechanism which modifies the size attribute of vertices. The following example serves as an introduction and illustrates the interest of this normalization mechanism. Let us consider the two assembly code snippets given below:

```
; Code Snippet 1
add al, bl
movzx eax, al
```

```
; Code Snippet 2
add eax, ebx
and eax, 0xff
```

In Code Snippet 1, there is a modular addition on 8 bits the result of which is padded with zeros. In Code Snippet 2, there is a modular addition on 32 bits the result of which is masked to keep only the least significant byte. These two code snippets compute the exact same value, but if we construct their corresponding DFGs, it is clear that they will be different. We see with this example, that observably similar DFGs[5] can differ only because of the size of some of their vertices. To make such DFGs converge to the same normal form, we propose to equalize the size of their vertices. We define a reference size which is equal to the architecture word size. This choice is coherent with how SIMD instructions are split and with memory coalescing. Moreover, there are very few vertices which have a size that is larger than the architecture word size. Hence, we will only have to deal with size increase and not with size decrease. We call S_{exp} the set of operation symbols which are affected by operation size expansion.

$$S_{\text{exp}} = \{+, \wedge, \texttt{cmov}, \texttt{neg}, \neg, \vee, \ll, -, \oplus\}$$

The principle of operation size expansion is to replace every operation which is smaller than the reference size, by an observably equivalent expression which contains the same operation but the size of which is equal to the reference size. Operation size expansion can be modelled by a set of rewrite rules. For instance, the pair of DFGs corresponding to Code Snippet 1 and 2 forms a rewrite rule which can be one of those. For simplicity and efficiency reasons, operation size expansion is applied using a specific method. This method has the particularity to perform every possible size modification at once and to minimize the number of size modifier operations[6] that are inserted. The remainder of this section will be focused on this method rather than on the rewrite rule aspect. It is made of three steps, namely size increase, obsolete size modifier deletion and new size modifier insertion. A pseudo code of this method is given in Algorithm 6.
This method is illustrated in Figure 3.14. This figure shows that if we normalize the DFG corresponding to Code Snippet 1 with the operation size expansion method given above, we obtain a normal form which contains a subgraph which is isomorphic to the DFG corresponding to Code

[5] To be perfectly accurate, in this example the two DFGs are not observably similar because they dot not have the same set of variable symbols. It is a minor deviation from the definition and it does not affect the general idea.

[6] Size modifier operation symbols are: `movzx`, `part18`, `part28`, `part116`. The only purpose of these operations is to modify the size of a variable, either by padding it with zeroes or by extracting a subset of its bits. They receive a specific treatment during operation size expansion.

[7] Size modifier operations have a single operand.

[8] If $labV_G(v) = \texttt{movzx}$, a masking operation needs to be inserted between u and $dst_G(e)$. The mask is equal to $2^i - 1$ where i denotes the initial size (before operation size expansion) of u. If $labV_G(v) = \texttt{part28}$, a right shift operation needs to be insert between u and $dst_G(e)$.

[9] If $\text{size}_G(src_G(e)) < \text{size}_G(dst_G(e))$, this size modifier operation will be `movzx`, otherwise it will be `part18` or `part116`.

Algorithm 6 Operation Size Expansion (assuming a 32-bit architecture)

for all vertices v in G **do** ▷ Step 1
  if $labV_G(v) \in S_{\text{exp}}$ **and** $\text{size}_G(v) < 32$ **then**
    $\text{size}_G(v) \leftarrow 32$
  end if
end for
for all vertices v in G such that $labV_G(v)$ is a size modifier operation symbol **do** ▷ Step 2
  let u be the only direct predecessor of v [7]
  for all edges e in G such that $src_G(e) = v$ **do**
    if $\text{size}_G(dst_G(e)) = \text{size}_G(u)$ **then**
      add a new edge from u to $dst_G(e)$ and remove e [8]
    end if
  end for
end for
for all edge e in G such that $\text{size}_G(src_G(e)) \neq \text{size}_G(dst_G(e))$ **do** ▷ Step 3
  if the size of either $src_G(e)$ or $dst_G(e)$ was increased during step 1 **then**
    add a new vertex, noted v. v is labelled with a size modifier operation symbol [9]
    add two new edges, one from $src_G(e)$ to v and one from v to $dst_G(e)$, and remove e
  end if
end for

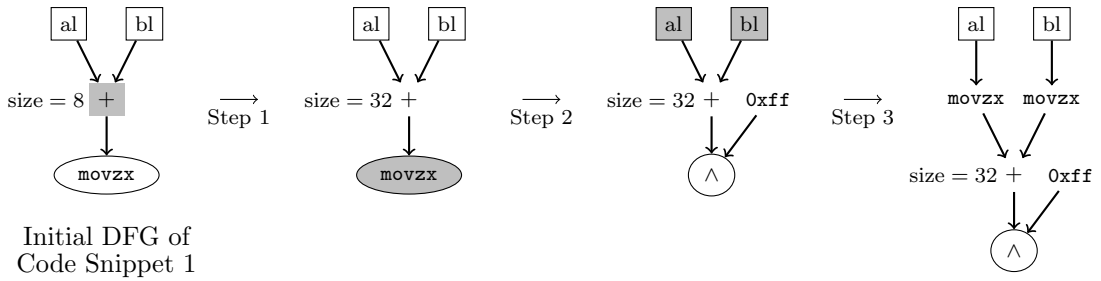

Figure 3.14: Example of an operation size expansion. The initial DFG on the left corresponds to Code Snippet 1 in Section 3.10. The three steps of the operation size expansion method are detailed: first size increase, then obsolete size modifier deletion and finally new size modifier insertion. Note that the final DFG contains a subgraph that is isomorphic to the DFG which corresponds to Code Snippet 2 in Section 3.10.

Snippet 2. Hence Algorithm 6 successfully solves the issue that was initially raised by Code Snippet 1 and 2.

Observable Similarity. In this paragraph we justify why the operation size expansion method given in Algorithm 6 preserves the observable similarity. The problem is the following. Let v be a vertex the size of which was increased during operation size expansion. v now takes wider input operand(s) and returns a wider result. Regarding the input operands, there are two possibilities: either they were replaced with `movzx` operations to adapt their size in Step 3 or they were subject to operation size expansion. In that latter case, the most significant bits which were appended to them, are not necessarily equal to zero. That being said, what guarantee do we have that the least significant bits return by v, those that were already returned by v before operation size expansion, will keep the same value for all the assignments?

A first observation is that the modulo operation distributes over every operation of S_{exp}. A second observation is that, given any operation in S_{exp}, the size of its operand(s) is equal to the size of its result. These two observations play a key role in the following proof. Let G and H be two DFGs such that H is the result of operation size expansion on G. Given a vertex $v \in V_H$, according to Algorithm 6, either $v \in V_G$ or v is labelled with a size modifier operation symbol and was inserted during Step 3. Let us assume $v \in V_G$ and $\text{arity}(labV_G(v)) > 0$. The operands of v in

G are noted $v_1, ..., v_n$ and $w_1, ..., w_n$ in H (v have the same number of operand in G and H but they might be different as explained previously).

Proposition. *Given an assignment θ, if $\theta(term_G(v_i)) = \theta(term_H(w_i))$ mod $size_G(v_i)$ for all $i \in \{1, ..., n\}$ then we have $\theta(term_G(v)) = \theta(term_H(v))$ mod $size_G(v)$.*

Proof. Let θ be an assignment which satisfies the initial hypothesis, then:

$$\begin{aligned}\theta(term_G(v)) &= labV_G(v)(\theta(term_G(v_1)), ..., \theta(term_G(v_n))) \\ &= labV_G(v)((\theta(term_H(w_1)) \bmod \text{size}_G(v_i)), ..., (\theta(term_H(w_n)) \bmod \text{size}_G(v_i)))\end{aligned}$$

According to the second observation:

$$\theta(term_G(v)) = labV_G(v)((\theta(term_H(w_1)) \bmod \text{size}_G(v)), ..., (\theta(term_H(w_n)) \bmod \text{size}_G(v)))$$

And according to the first observation:

$$\begin{aligned}\theta(term_G(v)) &= labV_H(v)(\theta(term_H(w_1)), ..., \theta(term_H(w_n))) \bmod \text{size}_G(v) \\ &= \theta(term_H(v)) \bmod \text{size}_G(v)\end{aligned}$$

□

Using this property, we can prove by induction that $\theta(term_G(v)) = \theta(term_H(v)) \bmod \text{size}_G(v)$ for all vertices $v \in V_H \cap V_G$ and all assignments θ. And finally, we conclude that operation size expansion, implemented as described in Algorithm 6, preserves the observable similarity.

3.11 Miscellaneous Rewrite Rules

Last but not least, we list in this section miscellaneous rewrite rules that do not belong to any particular category or family of rewrite rules. These rules pursue different objectives. Some of them were introduced to simplify expressions (the third rewrite rule of Table 3.2 for instance). Some others arbitrarily replace an expression by an equivalent one and they would probably have worked just the same if they had performed the inverse transformation (the fourth rewrite rule of Table 3.2 for instance). There is no particular convergence or termination issue to be reported for these rules. These rules are not implemented through a unified algorithm but instead they each rely on a specific routine to find injective morphisms, check side conditions and perform substitutions. The full list is given in Table 3.2.

3.12 Conclusion

The set of normalization mechanisms that has been presented in this chapter is by no means definitive. One may freely add new rewrite rules or even new normalization mechanisms if it helps to reduce the number of signatures that he needs in order to detect new implementations or new primitives. Rewrite rules have to preserve the observable similarity, otherwise normal forms may greatly differ from their original DFGs in terms of behaviour. One should also pay attention that the reduction system remains terminating, convergent and does not raise too many context-sensitivity issues in practice. To fix a non-terminating system one may impose a strict scheduling of the normalization mechanisms to avoid that those which are in conflict are executed immediately one after the other. This is the solution we adopted for common subexpression elimination and commutative and associative operation merging. We have no general advice or trick of any sort to address possible convergence issues. A possibility though, would be to delay the resolution of the problem. Instead of creating a write rule that would be applied during the normalization phase, one can create compound signatures (refer to Chapter 4). Compound signatures are very similar to rewrite rules but they are processed in a way that is unaffected by convergence problems. To

Table 3.2: Full list of miscellaneous rewrite rules.

$$
\begin{aligned}
\wedge(x, c) &\to x \text{ if } D_G(\wedge(x, \neg c)) = \{0\} \\
\times(x, 2^c) &\to \ll(x, c) \\
\vee(..., x, x, ...) &\to \vee(..., x, ...) \\
\circlearrowleft(x, c) &\to \circlearrowright(x, \text{size} - c) \\
\ll(\ll(x, c_1), c_2) &\to \ll(x, c_1 + c_2) \\
\ll(\gg(x, c_1), c_2) &\to \begin{cases} \wedge(\gg(x, c_1 - c_2), (2^{\text{size}-c_2} - 1) \ll c_2), & \text{if } c_1 \geq c_2 \\ \wedge(\ll(x, c_2 - c_1), (2^{\text{size}-c_2} - 1) \ll c_2), & \text{otherwise} \end{cases} \\
\texttt{shld}(x_1, x_2, x_1) &\to \circlearrowleft(x_1, x_2) \\
\gg(\gg(x, c_1), c_2) &\to \gg(x, c_1 + c_2) \\
\gg(\ll(x, c_1), c_2) &\to \begin{cases} \wedge(\ll(x, c_1 - c_2), 2^{\text{size}-c_2} - 1), & \text{if } c_1 \geq c_2 \\ \wedge(\gg(x, c_2 - c_1), 2^{\text{size}-c_2} - 1), & \text{otherwise} \end{cases} \\
\texttt{shrd}(x_1, x_2, x_1) &\to \circlearrowright(x_1, x_2) \\
-(x, c) &\to +(x, -c) \\
\oplus(..., x, x, ...) &\to \oplus(..., 0, ...) \\
\oplus(x, c) &\to \neg(x) \text{ if } c = 2^{\text{size}} - 1
\end{aligned}
$$

mitigate context-insensitivity, one can try to avoid rewrite steps which mix instructions of different functions. This is one of the solutions we adopted for commutative and associative operations normalization. It relies on the hypothesis that the primitive is implemented in a single function and since normalization essentially concerns primitive identification there is no need to normalize expressions that belongs to different functions (except maybe to simplify expressions to produce better over-approximations or to improve alias analysis).

That being said, we still believe that the mechanisms described in this chapter provide a solid basis for further developments.

Chapter 4

Subgraph Isomorphism for Primitive Identification

Signatures are distinctive DFG subgraphs which are used to identify terms which are specific to a given primitive. In this chapter, we first introduce the concept of signature. Then we explain how signatures are detected in normalized DFGs using a subgraph isomorphism algorithm. Finally, we present and discuss experimental results on primitive identification.

Contents

4.1 Signature 77
4.1.1 Introduction 77
4.1.2 Definition 78
4.1.3 Observable Similarity 79
4.1.4 Signature Creation 80
4.2 Signature Detection 81
4.2.1 Simple Signature Detection 81
4.2.2 Composite Signature Detection 83
4.2.3 Difference between Signatures and Rewrite Rules 88
4.3 Experimental Evaluation 88
4.3.1 Description of the Set of Synthetic Samples 90
4.3.2 Description of the Set of Signatures 91
4.3.3 Results 97
4.3.4 Performance 101
4.3.5 Conclusion 102

4.1 Signature

4.1.1 Introduction

In its most basic form, a signature is a DFG subgraph which is specific to a given primitive. The content of a signature is a trade-off between flexibility and precision. On one hand, a signature should be able to match a wide range of implementations. In this respect, we may be tempted to create very small signatures with only a few distinctive operations that we are sure to find in every possible implementation. This solution was adopted in [58]. But on the other hand, a signature should be complete and precise to reduce the number of false positives and to reveal every feature of the primitive we might be interested in. In particular, as motivated in Chapter 5, we want to locate parameters of primitives. Consequently, signatures must contain vertices corresponding to those parameters and thus they usually cover the full length of primitives.

Ideally the normalization process should be able to transform any implementation of a given primitive into a unique normalized DFG. Unfortunately the level of analysis required is sometimes

far beyond the reach of our normalization mechanisms. When several non-isomorphic normal forms exist for the same primitive, our last resort is to increase the number of signatures. To mitigate this issue, we introduce the concept of composite signatures.

Let us consider a primitive divided into n disjoint parts, each of them with x_i different normal forms. To detect the entire primitive, a signature must match its n parts simultaneously. The number of signatures required to detect every possible implementation is $\prod_1^n x_i$. The idea behind composite signatures is to create small signatures, that cover each a limited portion of the primitive, and to compose them in order to detect the entire primitive. Signature composition is implemented through special vertices which are labelled with signature symbols. Every signature exports a symbol. Intuitively, the composition of a signature S_1 with a signature S_2 is obtained by replacing every vertex of S_1 labelled with the symbol exported by S_2, by the graph of S_2. A signature S which contains a vertex labelled with a signature symbol s is called a composite signature, and we say that S imports the symbol s. The key point is that several signatures can export the same symbol. Signature symbols form an abstraction layer between a composite signature and the exact definition of the signatures it imports. A composite signature delegates to a set of signatures, which all export the same symbol, the task of detecting one or several of its subterms. Back to our example, for every part of the primitive, we create x_i non-composite signatures which all export the same symbol s_i. With theses non-composite signatures, we can identify independently each part of the primitive. Then, we create a composite signature which assembles symbols $s_1, ...$ to s_n and which covers the entire primitive. To conclude, we can detect every possible implementation with only $\sum_1^n x_i$ non-composite signatures and one composite signature.

Composite signatures have several other advantages. They can help to write signatures in a compact way. For instance, let us assume that we want to write a signature for a primitive that executes several times the same round function. Instead of writing a single signature that duplicates the round function expression, we create a non-composite signature which detects only one execution of the round function, and we create a composite signature which contains as many references to the one round function signature as necessary. Breaking down large signatures into smaller ones may also improve performance. This strategy is illustrated at the end of Section 4.3.4. Finally, dividing primitives into multiple parts and detecting each of them separately is of high interest for debugging purposes. If, for some reason, the normal form of a part of a primitive is not as expected, the other parts will still be correctly detected providing partial identification information and pointing out the problematic part.

4.1.2 Definition

We had difficulties trying to formalize the notion of signatures in a definition that would be coherent with the DFG definition given in Section 2.1.1. These difficulties come from the fact that in DFGs, vertices labelled with an operation symbol represent both an operation and the result of that operation. This has not been problematic so far, because we have only been interested in simple operations with a single output value. Unfortunately, it prevents a straightforward implementation of composite signatures for two reasons.

- As mentioned earlier, composite signatures contain special vertices labelled with signature symbols. These vertices were supposed to play a role similar to the role played by the vertices labelled with operation symbols. That is to say, they were supposed to represent both the computations defined by their corresponding signatures and the result of these computations. But in the general case, signatures define several expressions and thus can have several output values. Because we cannot dissociate operations from their result(s), there is no coherent way to represent an operation with several output values.

- To detect composite signatures we superimpose on normalized DFGs vertices representing the signature matches that have been found so far. The resulting graph is called an augmented DFG. Both composite signature detection and augmented DFGs are presented more clearly in Section 4.2.2. In an augmented DFG, a single value can result from several equivalent expressions. Let us consider a DFG G, a signature S and a matching f from S to G (that is to say an injective morphism). If a vertex v is equal to the image under f_V of an output value of S, then once the new vertex representing f has been superimposed on G, the output

value of v has two definitions: v and the newly inserted vertex. Here again, since we cannot dissociate operations from their result, there is no coherent way to represent a value resulting from several operations.

In our opinion the best solution to this problem would be to use hypergraphs instead of graphs as underlying structures for DFGs. For instance in term graphs [60] and in jungles [39], an operation is represented by an hyperedge with one source endpoint per input parameter and one destination endpoint per output parameter and a value is represented by a vertex. In this model a vertex can have several ingoing edges, that is to say, a value can result from several operations. Unfortunately, when we became aware of this solution it was too late to implement it. Therefore, in the rest of this chapter, we only present the workaround that was originally devised to address this problem. Note that this is mostly a formulation problem and it does not impact the success rate and the performance of our solution.

Definition 10. A non-composite signature S is defined by a DFG $(V_S, E_S, src_S, dst_S, labV_S, labE_S)$, a symbol s and two sequences of vertices noted respectively I_S and O_S. s is the symbol exported by S, I_S represents the input parameter(s) of S and O_S the output parameter(s) of S. If several signatures export the same symbol, we assume that they all have the same number of input and output parameters.

A composite signature S' is defined in a similar way, except $(V_{S'}, E_{S'}, src_{S'}, dst_{S'}, labV_{S'}, labE_{S'})$ is not quite a DFG. As previously stated, it contains vertices labelled with signature symbols. Let v be such a vertex and let s be its label. v has one ingoing edge per input parameter of s and one outgoing edge per output parameter of s. Its ingoing edges, respectively outgoing edges, are labelled with distinct positive integer ranging from 0 to $\text{indegree}_{S'}(v) - 1$, respectively from 0 to $\text{outdegree}_{S'}(v) - 1$. For a reason that will be explained in Section 4.2.2, the direct successors of v are labelled with operation symbols. But they do not represent any operation. They only represent the different output values of v. This is why they have no other direct predecessor than v.

An example of a composite signature is given on the right of Figure 4.1. Vertices labelled with signature symbols are depicted with rounded rectangle shape, input parameters have a rectangle shape and output parameters have a circle shape. We do not explicitly give the function which maps vertices of composite signatures to terms, because it is relatively complex due to the violations of the original DFG syntax mentioned above. But it remains possible to associate terms to vertices: for instance, the term associated with the bottom-most vertex of the composite signature given in Figure 4.1 is $1R(y, 1R(x, y))$.

4.1.3 Observable Similarity

At the beginning of Chapter 3 we presented a certain notion of semantics called observable similarity. Two DFGs are observably similar if for any set of input values they return the same set of output values. In respect to this definition, it seems reasonable to assume that two implementations of the same primitive are observably similar. The main steps of our primitive identification method are summarized as follows. First, we create a DFG G which represents a sequence of dynamic instructions. Then, during the normalization phase, we modify G using transformations which preserve the observable similarity. We note $\bar{G}$ the normal form of G. We assume here that the normalization process is working in the best possible way. That is to say, any possible implementation of a primitive $\mathcal{P}$ converges to a single normal form noted S. Finally, we search for S in $\bar{G}$ or more formally, we search for an injective morphism from S to $\bar{G}$.

Our primitive identification method is correct if finding S in $\bar{G}$ implies that G contains an implementation of $\mathcal{P}$ and reciprocally. More formally, our primitive identification method is correct if the following proposition holds.

Proposition (Primitive Identification Correctness)**.** *There is an injective morphism from S to $\bar{G}$, if and only if there is a DFG H such that H is observably similar to S and there is an injective morphism from H to G.*

If S is isomorphic to $\bar{G}$ then S is also observably similar to $\bar{G}$. Since G and $\bar{G}$ are observably similar, S is observably similar to G. Conversely, if S is observably similar to G then according

to our hypothesis regarding the normalization phase, $\bar{G}$ is isomorphic to S. Consequently the proposition above is valid in this corner case.

Unfortunately, in the general case where the morphism from S to $\bar{G}$ is not surjective and where S is not observably similar to G, this proposition is not true. First, the normalization phase is not context-insensitive: the fact that there is an injective morphism from H to G does not imply that there is an injective morphism from $\bar{H}$ to $\bar{G}$. Second, the normalization process does not preserve the observable similarity for any subgraph of G and any subgraph of $\bar{G}$ does not have an observably similar subgraph in G. More pragmatically the fact that this property does not hold has two consequences.

- First, if G contains an implementation of $\mathcal{P}$ then $\bar{G}$ does not necessarily contain an instance of S. It means that, regardless of the capabilities of the normalization mechanisms, there can be false negatives. We have presented this problem from a theoretical point of view but it has rather serious repercussions in practice. For instance let us consider a trace segment which contains one execution of a hash function. Let us assume that for some reason, this trace segment contains additional instructions after the execution of the hash function and that these instructions discard one part of the hash value. Observable similarity will only concern the remaining part of the hash value. Thus, according to the dead code removal mechanism, the portion of the DFG which produces the part of the hash value that has been discarded will be deleted. This problem does not exclusively affect the dead code normalization mechanism, but it affects any normalization mechanism which is not context-insensitive.

- Second, if $\bar{G}$ contains an instance of S then G does not necessarily contain an implementation of $\mathcal{P}$. It means that there can be false positives. This problem is a lot less concerning in practice than the first one though. From a theoretical point of view, a rewrite rule can insert a new expression in a DFG in such a way that this expression has no influence on the final vertices. But in practice this scenario is highly improbable: expressions covered by signatures are very large and most normalization mechanisms reduce the size of expressions.

4.1.4 Signature Creation

For the time being, signatures still have to be generated manually. We choose to put more effort into the normalization mechanisms to reduce the number of signatures rather than to try to automate their creation process. That being said, signature creation might not be particularly difficult to automate or, at least, to partially automate. For instance, given a primitive implementation in assembly language, one can annotate its parameters, construct its DFG representation while keeping track of the annotated parameters, normalize it, and finally delete every vertex that is not located on a path between an input and an output parameter.

Manual signature creation requires some knowledge of the assembly language and a good understanding of the different implementations of the primitive. If signatures had been syntactically correct DFGs, there would have been no prerequisite regarding the normalization mechanisms. In fact, it would have been possible to normalize signatures like any other DFG. But since it is not the case, one should have a clear vision of what the normal forms will look like in order to create signatures. And this implies to be well aware of the different normalization mechanisms. This is an important drawback of having syntactically incorrect signatures.

From an implementation perspective, signatures are specified using a textual graph description language of our own making, largely inspired by the DOT language [25]. A formal grammar specification of this language is given in Table 4.1. Basically, a signature is a list of edges, an edge is pair of vertices and a vertex is a number. We can optionally specify a symbol for signatures[1], edges and vertices. A parameter attribute can also be specified for vertices.

[1]Each signature also has a unique name. Signature names differ from signature symbols which may be shared by several signatures.

Table 4.1: Formal grammar specification of the signature language. Terminals are printed in bold font and non-terminals in italic. Literals are enclosed by quotes ‘and ’. Optional items are surrounded by square brackets [and]. A vertical bar | is used to separate different alternatives. *ID* is a numeral. *NAME* is a string of alphabetic characters, digits, underscores and spaces. *SYMBOL* is either a string of alphabetic characters, digits and underscores or an asterisk *.

sig_list:	*sig sig_list*
sig:	**‘”’** *NAME* **‘”’** [**‘(’** *SYMBOL* **‘)’**] *edge_list*
edge_list:	*edge edge_list*
edge:	*vertex* **‘->’** [**‘(’** *SYMBOL* **‘)’**] *vertex*
vertex:	*ID* [**‘(’** *SYMBOL* **‘)’**] [*parameter*]
parameter:	**‘[’‘I’|‘O’‘:’** *ID* **‘:’** *ID* **‘]’**

4.2 Signature Detection

In this section we present the algorithms which are used for signature detection. Given a signature S and a DFG G, we want to find all the subgraphs of G which are isomorphic to S. To put it differently, we want to find all the injective morphisms from S to G. This problem is referred to as the subgraph isomorphism problem. The two most popular algorithms to solve it are Ullmann's algorithm [72] and VF2 [20, 19]. In this work, we arbitrarily chose to use Ullmann's algorithm. As illustrated in Section 4.3.4, this algorithm has acceptable performance when applied in the context of signature detection. This section is structured as follows: first we present Ullmann's algorithm for subgraph isomorphism, then we introduce some additional mechanisms which are required to detect composite signatures.

4.2.1 Simple Signature Detection

As stated in Definition 2 (Section 2.1.1) a DFG morphism from G to H is defined by two mappings: one from V_G to V_H and one from E_G to E_H. To justify more clearly that the algorithms presented in this section return valid DFG morphisms, we consider a slightly different definition for DFG morphisms. According to this new definition and on condition that every vertex of G has at least one ingoing edge or one outgoing edge, a DFG morphism can be fully specified by a single mapping from E_G to E_H[2]. This definition is as follows.

Definition 11. Given two DFGs $G = (V_G, E_G, src_G, dst_G)$ and $H = (V_H, E_H, src_H, dst_H)$, a morphism from G to H is defined by a mapping $f_E \colon E_G \to E_H$ which satisfies the following properties[3]:

- $\begin{cases} src_G(e_1) = src_G(e_2) \Rightarrow src_H(f_E(e_1)) = src_H(f_E(e_2)) \\ dst_G(e_1) = src_G(e_2) \Rightarrow dst_H(f_E(e_1)) = src_H(f_E(e_2)) \\ dst_G(e_1) = dst_G(e_2) \Rightarrow dst_H(f_E(e_1)) = dst_H(f_E(e_2)) \end{cases}$ for all $e_1, e_2 \in E_G$;
- $labV_G(src_G(e)) = labV_H(src_H(f_E(e)))$ for all $e \in E_G$ such that $labV_G(src_G(e)) \notin X$, and $labV_G(dst_G(e)) = labV_H(dst_H(f_E(e)))$ for all $e \in E_G$;
- $labE_G(e) = labE_H(f_E(e))$ for all $e \in E_G$ such that $labV_G(dst_G(e))$ is non-commutative.

This definition is equivalent to Definition 2 if every vertex of G has at least one ingoing edge or one outgoing edge. In practice every signature satisfies this last requirement. In fact, there is no point in creating a signature with a unique vertex or in creating a disconnected[4] signature. These

[2]Because DFGs are build on top of multigraphs, the opposite is not true: a DFG morphism cannot be fully specified by a mapping from V_G to V_H.

[3]For simplicity this definition does not include the last requirement of the Definition 2, that is to say: $f_V(u) = f_V(v)$ for all $u, v \in V_G$ such that u and v are labelled with the same variable symbol. This requirement is not ensured by the algorithms presented in this section. Though, we can easily assume that in practice signatures have at most one vertex labelled with each variable symbol.

[4]A graph G is disconnected if there is pair of vertices $(v_1, v_2) \in V_G^2$ such that there is no undirected path from v_1 to v_2.

are the only two scenarios that would result in a vertex with no ingoing and outgoing edge. The following proof should be skipped on first reading.

Proof. Let (f_V, f_E) be a morphism from G to H according to Definition 2. Let e_1, e_2 be two vertices of G such that $src_G(e_1) = src_G(e_2)$, then:

$$\begin{aligned} src_H(f_E(e_1)) &= f_V(src_G(e_1)) \\ &= f_V(src_G(e_2)) \\ &= src_H(f_E(e_2)) \end{aligned}$$

It proves the first implication of Definition 11. The other two implications can be proved in a similar way. The second point of Definition 11 comes from the second point of Definition 2, that is to say: $labV_G(v) = labV_H(f_V(v))$ for all $v \in V_G$ such that $labV_G(v) \notin X$. And finally, the last point of Definition 11 is exactly the same that the third point of Definition 2. Conclusion: f_E also satisfies Definition 11.

Let f_E be a morphism from G to H according to Definition 11. Let f_V be a mapping from V_G to V_H defined as follows:

$$f_V(v) = \begin{cases} src_H(f_E(e)), & \text{if there is } e \in E_G \text{ such that } src_G(e) = v \\ dst_H(f_E(e), & \text{otherwise. In that case } e \in E_G \text{ and } dst_G(e) = v. \end{cases}$$

The definition of f_V is correct because if two edges of G have the same endpoint, their image under f_E will also have the same endpoint. According to this definition we have:

$$f_V \circ src_G = src_H \circ f_E$$

Let e_1 be an edge of G. If there is $e_2 \in E_G$ such that $dst_G(e_1) = src_G(e_2)$ then:

$$\begin{aligned} dst_H(f_E(e_1)) &= src_H(f_E(e_2)) \\ &= f_V(src_G(e_2)) \text{ by definition of } f_V \\ &= f_V(dst_G(e_1)) \end{aligned}$$

Otherwise, by definition of f_V: $f_V(dst_G(e_1)) = dst_H(f_E(e_1))$. It proves that:

$$f_V \circ dst_G = dst_H \circ f_E$$

Now, let v be a vertex of G which is not labelled with a variable symbol. If there $e \in E_G$ such that $src_G(e) = v$ then:

$$\begin{aligned} labV_G(v) &= labV_G(src_G(e)) \\ &= labV_H(src_H(f_E(e))) \text{ according to the second point of Definition 11} \\ &= labV_H(f_V(v)) \text{ by definition of } f_V \end{aligned}$$

Otherwise, there is $e \in E_G$ such that $dst_G(e) = v$ and we can prove similarly that $labV_G(v) = labV_H(f_V(v))$. Conclusion: (f_V, f_E) satisfies Definition 2. □

Naive Enumeration Algorithm for Subgraph Isomorphism. Let S be a signature and G be a DFG. Finding all the subgraph isomorphisms from S to G can be achieved using a rather simple DFS algorithm. For each edge e of S, we maintain a set of possible assignments called $A_E(e)$. $A_E(e)$ is initialised with the edges of G which have the same label than e and the endpoints of which have the same label than the endpoints of e. Thereby if we assign $f_E(e)$ to any element of $A_E(e)$, the last two points of Definition 11 will be satisfied. The algorithm expands a partial solution (initially empty) by recursively picking one element in each possible assignment set. Before picking e' in $A_E(e)$, we check that, if f_E has been defined for an edge which shares an endpoint with e, its image under f_E shares the same endpoint with e' (first point of Definition 11). After picking e' in $A_E(e)$, we delete all instances of e' in the possible assignment sets that have not yet been processed. This ensures that the solution is injective. Eventually, the algorithm either finds

a complete solution or reaches a point where the partial solution cannot be further expanded. In that latter case, we backtrack until we find a point where the partial solution can be expanded in a different way. A pseudo code of this algorithm is given in Algorithm 7. It is divided into three procedures. The first procedure initialises the sets of possible assignments. The second procedure performs the DFS. And the third procedure, called FILTER, checks that the first point of Definition 11 is satisfied by all selected edges. Unfortunately this algorithm is not tractable for the graphs which are used in the context of primitive identification.

Ullmann's Algorithm for Subgraph Isomorphism. Ullmann introduced a refinement procedure which preventively checks, for all possible assignments, whether or not they have a chance to satisfy the three implications given at the beginning of Definition 11. Its objective is to prune as early as possible unfruitful search paths. Let e_1 and e_2 be two edges of S with a shared endpoint. Let us assume, for instance, that $src_S(e_1) = src_S(e_2)$. Given $e'_1 \in A_E(e_1)$, if there is no $e'_2 \in A_E(e_2)$ such that $src_G(e'_1) = src_G(e'_2)$ then, according to the first implication of Definition 11, we cannot assign $f_E(e_1)$ to e'_1. Consequently, we can remove e'_1 from $A_E(e_1)$. A pseudo code of Ullmann's refinement procedure is given in Algorithm 8. It replaces the FILTER procedure of Algorithm 7 (the other two procedures remain unchanged). For a better efficiency, we adopt a slightly different formulation in Algorithm 8. Instead of filtering independently the set of possible assignments associated with each edge, we regroup the edges with a shared endpoint and we filter them simultaneously. Let us consider a group of edges with a shared endpoint v. Their possible assignments must all define the same set of vertices for the image of v. This set of vertices is noted $A_V(v)$ in Algorithm 8. If a possible assignment defines an image of v that is not also defined by at least one possible assignment for all the other edges, it can be deleted.

This filtering process is executed iteratively until a fixed point is eventually reached. The set $\mathcal{V}$ is used to keep track of the vertices that need to be (re-)filtered. Given an edge $e \in E_S$, if a possible assignment was removed from $A_E(e)$ after filtering one endpoint of e, further deletions may become possible for the edges attached to the other endpoint of e. Thus we add this other endpoint to $\mathcal{V}$.

Despite a high theoretical complexity (the subgraph isomorphism problem is NP complete), we were able to achieve acceptable performance using Ullmann's algorithm in our context (some typical computation time measurements are presented in Section 4.3.4). It can be explained by two factors. First, DFGs have specific characteristics and they usually differ from the worst case scenario. For instance, DFGs have a low density (probability for any two vertices to be adjacent) and their vertices have highly heterogeneous indegree and outdegree. Second, labels of vertices and edges dramatically reduce the search space.

4.2.2 Composite Signature Detection

For introductory purposes let us consider the example of Figure 4.1. In this example, our objective is to identify a toy block cipher. This toy block cipher is a two branch Feistel network. Its internal state consists of two registers (X_i, Y_i). At each round i, its internal state is updated as follows:

$$\begin{cases} X_{i+1} & \leftarrow Y_i \\ Y_{i+1} & \leftarrow +(\circlearrowleft (\oplus(X_i, Y_i), 1), X_i) \end{cases}$$

To identify this block cipher we create two signatures. The first one covers the execution of one round and exports the symbol '1R'. The second one covers the execution of two rounds and exports the symbol '2R'. This last signature is a composite signature: instead of including two times the round function expression, it contains two vertices labelled with the '1R' symbol. To detect these two signatures we proceed as follows. First, we search for the one-round signature using Ullmann's algorithm. For the DFG represented in Figure 4.1, two matches are found: one is highlighted on the top left and the other on the bottom left. Then, we add to the DFG one vertex labelled with the '1R' symbol per signature match. This transformation is called a *push* transformation. It is described more precisely in the next paragraph. We finally search for the two-round signature using a slightly modified version of Ullmann's algorithm to deal with the newly inserted vertices. One match is found; the block cipher has correctly been identified.

Algorithm 7 DFS Subgraph Isomorphism

function SUBGRAPH ISOMORPHISM(S, G)
  for all $e \in E_S$ **do**
    $A_E(e) \leftarrow \left\{ e' \in E_G \text{ such that } \begin{cases} labE_S(e) = labE_G(e') \\ labV_S(src_S(e)) = labV_G(src_G(e')) \\ labV_S(dst_S(e)) = labV_G(dst_G(e')) \end{cases} \right\}$
    assign $f_E(e)$ to undefined
  end for
  return RECURSIVE SEARCH(S, G, f_E, A_E)
end function

function RECURSIVE SEARCH(S, G, f_E, A_E)
  if $f_E(e)$ is defined for all $e \in E_S$ **then**
    return $\{f_E\}$
  end if
  Result $\leftarrow \emptyset$
  pick an edge $e_1 \in E_S$ such that $f_E(e_1)$ is undefined
  FILTER(S, G, f_E, A_E, e_1)
  for all $e'_1 \in A_E(e_1)$ **do**
    $f'_E \leftarrow$ copy(f_E) and assign $f'_E(e_1)$ to e'_1
    $A'_E \leftarrow$ copy(A_E)
    $A'_E(e_2) \leftarrow A'_E(e_2) \setminus \{e'_1\}$ for all $e_2 \in E_S$
    $A'_E(e_1) \leftarrow \{e'_1\}$ ▷ necessary for Algorithm 8
    Result $\leftarrow$ Result $\cup$ RECURSIVE SEARCH(S, G, f'_E, A'_E)
  end for
  return Result
end function

function FILTER(S, G, f_E, A_E, e_1)
  for all $e'_1 \in A_E(e_1)$ **do**
    if $\exists e_2 \in E_S$ s.t. $f_E(e_2)$ is defined and $\begin{cases} src_S(e_1) = src_S(e_2) \\ src_G(e'_1) \neq src_G(f_E(e_2)) \end{cases}$ **then**
      remove e'_1 from $A_E(e_1)$
      break
    end if
    if $\exists e_2 \in E_S$ s.t. $f_E(e_2)$ is defined and $\begin{cases} src_S(e_1) = dst_S(e_2) \\ src_G(e'_1) \neq dst_G(f_E(e_2)) \end{cases}$ **then**
      remove e'_1 from $A_E(e_1)$
      break
    end if
    if $\exists e_2 \in E_S$ s.t. $f_E(e_2)$ is defined and $\begin{cases} dst_S(e_1) = src_S(e_2) \\ dst_G(e'_1) \neq src_G(f_E(e_2)) \end{cases}$ **then**
      remove e'_1 from $A_E(e_1)$
      break
    end if
    if $\exists e_2 \in E_S$ s.t. $f_E(e_2)$ is defined and $\begin{cases} dst_S(e_1) = dst_S(e_2) \\ dst_G(e'_1) \neq dst_G(f_E(e_2)) \end{cases}$ **then**
      remove e'_1 from $A_E(e_1)$
      break
    end if
  end for
end function

Algorithm 8 Refinement Procedure Ullmann

function FILTER(S, G, f_E, A_E)
  $\mathcal{V} \leftarrow V_S$
  while $\mathcal{V} \neq \emptyset$ **do**
    pop any vertex v from $\mathcal{V}$
    $A_V(v) \leftarrow (\bigcap_{e \in E_S, src_S(e)=v} src_G(A_E(e))) \cap (\bigcap_{e \in E_S, dst_S(e)=v} dst_G(A_E(e)))$
    for all $e \in E_S$ s.t. $src_S(e) = v$ **do**
      $A'_E(e) \leftarrow \{e' \in A_E(e) \text{ s.t. } src_G(e') \in A_V(v)\}$
      if $A'_E \neq A_E$ **then**
        $\mathcal{V} \leftarrow \mathcal{V} \cup \{dst_S(e)\}$
      end if
      $A_E \leftarrow A'_E$
    end for
    for all $e \in E_S$ s.t. $dst_S(e) = v$ **do**
      $A'_E(e) \leftarrow \{e' \in A_E(e) \text{ s.t. } dst_G(e') \in A_V(v)\}$
      if $A'_E \neq A_E$ **then**
        $\mathcal{V} \leftarrow \mathcal{V} \cup \{src_S(e)\}$
      end if
      $A_E \leftarrow A'_E$
    end for
  end while
end function

Push and Pop Transformation. Let G be a DFG, S be a signature and f be an injective morphism from S to G. The vertices of I_S, respectively O_S, are noted $x_1, ..., x_n$, respectively $y_1, ..., y_m$. When we say that we *push* f on G, we refer to the following modifications of G:

1. we insert a new vertex v and we assign its label to the symbol exported by S ;
2. for every vertex $x_i \in I_S$, we insert an edge from $f_V(x_i)$ to v and we assign its label to i ;
3. for every vertex $y_i \in O_S$, we insert an edge from v to $f_V(y_i)$ and we assign its label to i.

In the example of Figure 4.1, when we *push* the one-round signature match highlighted on the top left, we obtain a vertex labelled with the '1R' symbol and connected to `eax` (image of the I1 parameter), `ebx` (image of the I2 parameter) and + (image of the O2 parameter). The opposite of a *push* transformation is a *pop* transformation. A *pop* transformation consists in removing all the vertices labelled with a given signature symbol. For the two reasons given in Section 4.1.2, vertices resulting from *push* transformations break the DFG definition. We will refer to the graph resulting from *push* transformation(s) as an augmented DFG.

Subgraph Isomorphism in Augmented DFGs. Intuitively a signature S defines an equivalence relation between two graphs: one is the DFG which is explicitly defined by S, that is to say $(V_S, E_S, src_S, dst_S, labV_S, labE_S)$, and the other corresponds to the small graph which is superimposed on a given DFG during a *push* transformation. As a reminder, this small graph is composed of a vertex v labelled with the symbol exported by S plus one vertex per input and output parameter of S, all directly connected to v. In this regard, signatures are very similar to rewrite rules. One may think of a *push* transformation as a rewrite step, but instead of substituting one graph by the other, we superimpose them. As a consequence, augmented DFGs contain multiple equivalent subgraphs. To detect a composite signature, we let the subgraph isomorphism algorithm pick among these equivalent subgraphs those that better match the signature. Intuitively, a match of composite signature in an augmented DFG G is valid, if it is possible to create an equivalent non-composite signature that matches either the same vertices of G or equivalent ones. A valid match is defined more formally as follows.

Definition 12 (Valid Match)**.** Let f_1 be an injective morphism from a signature S_1 to an augmented DFG G. f_1 is a valid match if:

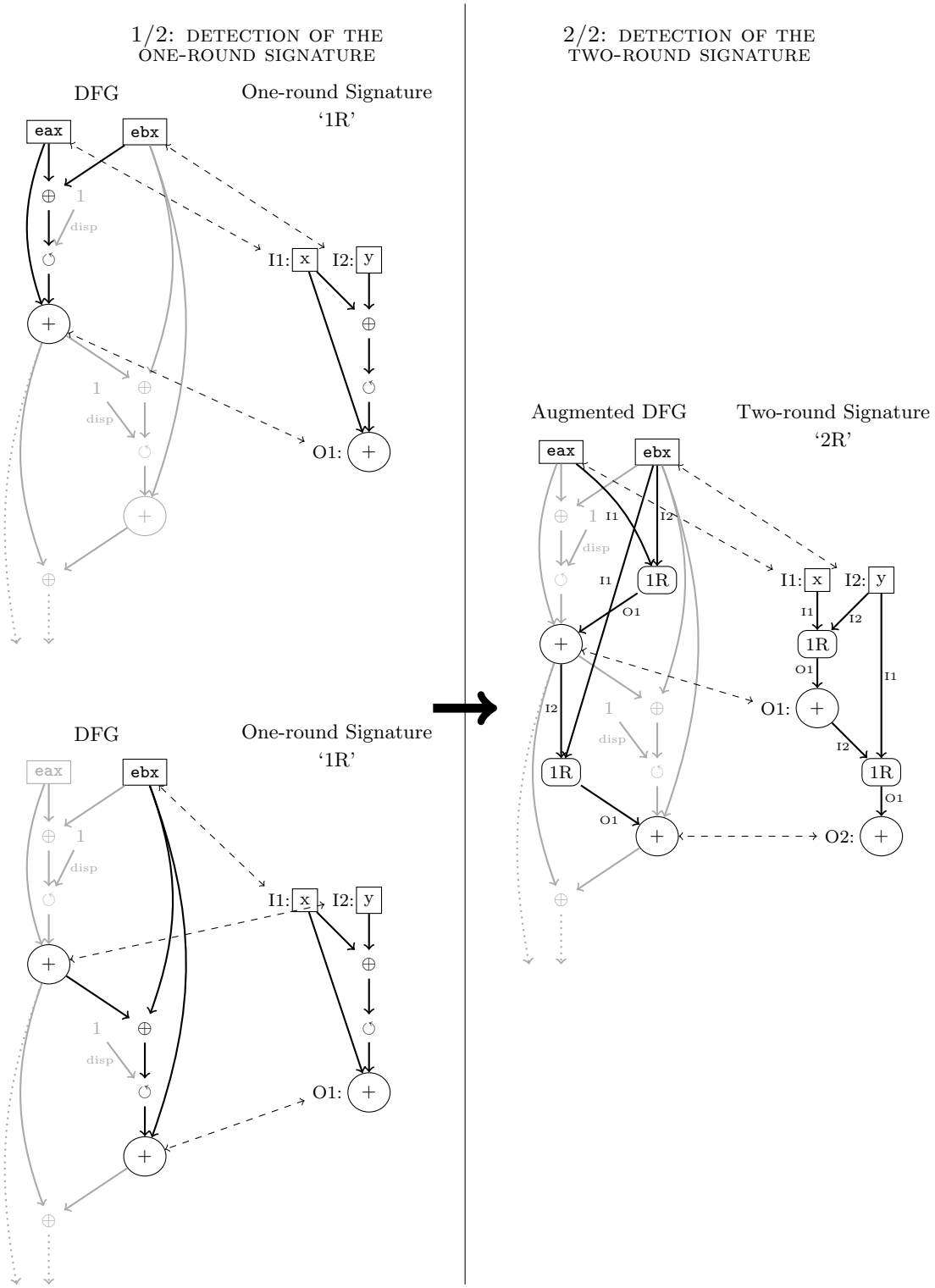

Figure 4.1: Example of a composite signature detection. This example involves two signatures: a one-round signature which exports the '1R' symbol and a two-round signature which contains two references to the '1R' symbol. First, on the left, we search for the one-round signature in the initial DFG. Two matches are found. Then, on the right we append these two matches to the initial DFG using *push* transformations and we search for the two-round signature. One match is found. Matches are depicted in black while the rest of the DFGs are depicted in light grey. Dashed lines are used to symbolize subgraph isomorphisms. Vertices labelled with a signature symbol have a rounded rectangle shape.

- S_1 does not contain any vertex labelled with a signature symbol ;
- given any vertex v in S_1 labelled with a signature symbol, let S_2 be the signature and f_2 be the injective morphism from S_2 to G such that $f_{V,1}(v)$ has resulted from the *push* transformation of f_2. We note S'_1 the signature which is obtained from S_1 by replacing v by S_2 [5]. Let $f'_{V,1}$ and $f'_{E,1}$ be two mappings defined as follows:

$$
\begin{array}{lccl}
f'_{V,1}: & V_{S'_1} & \mapsto & V_G \\
 & v & \rightarrow & \begin{cases} f_{V,2}(v) & \text{if } v \in V_{S_2} \\ f_{V,1}(v) & \text{otherwise} \end{cases} \\
f'_{E,1}: & E_{S'_1} & \mapsto & E_G \\
 & e & \rightarrow & \begin{cases} f_{E,2}(e) & \text{if } e \in E_{S_2} \\ f_{E,1}(e) & \text{otherwise} \end{cases}
\end{array}
$$

  f_1 is a valid match if $f'_1 = (f'_{1,V}, f'_{1,E})$ is an injective morphism from S'_1 to G and if f'_1 is a valid match.

In particular, a match f of a signature S is not valid if $f(S)$ contains two equivalent subgraphs. Equivalent subgraphs only exist simultaneously in augmented DFGs. They are reformulations of a single initial set of expressions. If $f(S)$ contains two equivalent subgraphs, sooner or later in the recursive definition above, we will replace one of this subgraphs by the other and the resulting morphism will be non-injective. To determine which elements of an augmented DFG can be simultaneously selected by a valid match, we introduce two relations over the edges of an augmented DFG.

- Direct collision relation: an edge e_1 is in direct collision with an edge e_2, noted $e_1 \lhd e_2$, if e_2 results from the *push* transformation of a match f of a signature S such that $e_1 \in f_E(E_S)$.
- Indirect collision relation: en edge e_1 is in indirect collision with an edge e_2, noted $e_1 \bowtie e_2$, if e_1 and e_2 do not result from the same *push* transformation and if there is an edge e_3 such that $e_3 \lhd^* e_1$ and $e_3 \lhd^* e_2$, where $\lhd^*$ denotes the reflexive-transitive closure of the direct collision relation $\lhd$.

Even though we are not going to prove it, we assume that the following claim is true.

Claim 4.2.1. *A match f of a signature S in an augmented DFG G is valid if only and if there are no two distinct edges e_1 and e_2 in $f_E(S_E)$ such that e_1 is in indirect collision with e_2.*

From an implementation perspective, the indirect collision relation is modelled by an undirected graph. This graph is updated every time a signature match is pushed or popped. During Ullmann's algorithm for subgraph isomorphism, after each edge selection, we filter the remaining sets of possible assignments to remove any edge that is in indirect collision with the newly selected edge. Thereby only valid matches are returned by the algorithm.

Processing a Set of Composite Signatures. To minimize the number of *push* and *pop* transformations and also to minimize the DFG size[6], we devised a simple strategy to efficiently search for all the elements of a set of composite signatures. Let *SIG* be a set of signatures such that all

[5] We referred to this transformation as a signature composition in Section 4.1.1. More precisely, S'_1 is equal to the union of S_1 and S_2, where:

- every input parameter x_i of S_2 has been replaced by $src_{S_1}(e)$ where e is the edge of S_1 such that $dst_{S_1}(e) = v$ and $labE_{S_1}(e) = i$;
- every output parameter y_i of S_2 has been replaced by $dst_{S_1}(e)$ where e is the edge of S_1 such that $src_{S_1}(e) = v$ and $labE_{S_1}(e) = i$;
- and v has been deleted.

[6] Pushing signature matches may significantly increase the size of a DFG. For instance, for certain signatures with a large number of automorphisms, such as the AT4 signature (refer to Section 4.3.2), it nearly doubles the size of the initial DFG.

the signature symbols imported by an element of *SIG* is also exported by at least one element of *SIG*. We define a directed graph structure over the elements of *SIG*. There is an edge from a signature S_1 to a signature S_2, if S_2 imports the symbol exported by S_1. Based on this graph structure, we perform a Breadth First Search (BFS) starting from the elements of *SIG* with no predecessor. A more detailed description of the algorithm is given below.

1. Pick any signature S_1 in *SIG* such that S_1 has not been processed and all the direct predecessors of S_1 have been processed.
2. Push all the matches of the direct predecessors of S_1 which have not been pushed yet.
3. Run the algorithm for subgraph isomorphism on S_1 and mark S_1 as processed.
4. For every signature S_2 such that all the direct successors of S_2 have been processed, pop all matches of S_2. If there is still a signature in *SIG* which has not been processed, go back to step 1.

Note that this algorithm does not work with recursive signatures, that is to say, signatures that import the symbol they export.

4.2.3 Difference between Signatures and Rewrite Rules

As pointed out in the previous section, composite signatures are very similar to rewrite rules. The only noticeable difference is that during a rewrite step we substitute a subgraph by an equivalent one, whereas during a push transformation we superimpose on a subgraph an equivalent one. Two questions are worth to be asked. Can we replace signatures by rewrite rules? And conversely, can we replace rewrite rules by signatures?

- Is it possible to detect composite signatures by performing a substitution instead of a superposition? The answer is yes in most of the cases but there is a risk to miss detections. Rewrite rules suffer from a problem that does not affect composite signatures, this problem is convergence. When several rewrite rules can be applied to the same vertices, depending on the rewrite rule that we decide to apply, we may obtain different results. With composite signatures, every match that is found is pushed on the augmented DFG, and we let the subgraph isomorphism algorithm select those which are relevant and discard those which are not. The benefit of composite signatures over rewrite rules is clearly visible in the example of Figure 4.2. The same DFG is processed using both, rewrite rules (on the left) and composite signatures (on the right). Unfortunately, on the left the algorithm stalls after the first rewrite step (rewrite rule R_1) and, in particular, it was not possible to apply the last rewrite rule R_3. On the right however, after pushing the two matches of the first signature and the one match of the second signature, the last signature is correctly detected. To conclude, our composite signature detection scheme offers more flexibility than classical rewrite rules do. One can write new signatures without bothering about possible convergence issues.

- Can we remove the normalization phase and transform every normalization mechanisms into a set of composite signatures? The answer is clearly no. First as explained in Chapter 3, not all the normalization mechanisms can be implemented using a finite set of rewrite rules. Second, composite signatures are significantly less efficient than rewrite rules. We can assume that the size of DFGs does not increase with the number of rewrite steps whereas it increases linearly in the number of *push* transformations. In practice, the number of rewrite steps to reach a normalized DFG is several orders of magnitude larger the number of *push* transformations required to identify a primitive. Thus, it will not be tractable to use composite signatures instead of rewrite rules.

4.3 Experimental Evaluation

The goal of this section is to demonstrate the validity of our solution. In particular the following two fundamental points need to be verified in practice. First, is the normalization process able to effectively remove implementation variations? That is to say, can we make all the possible

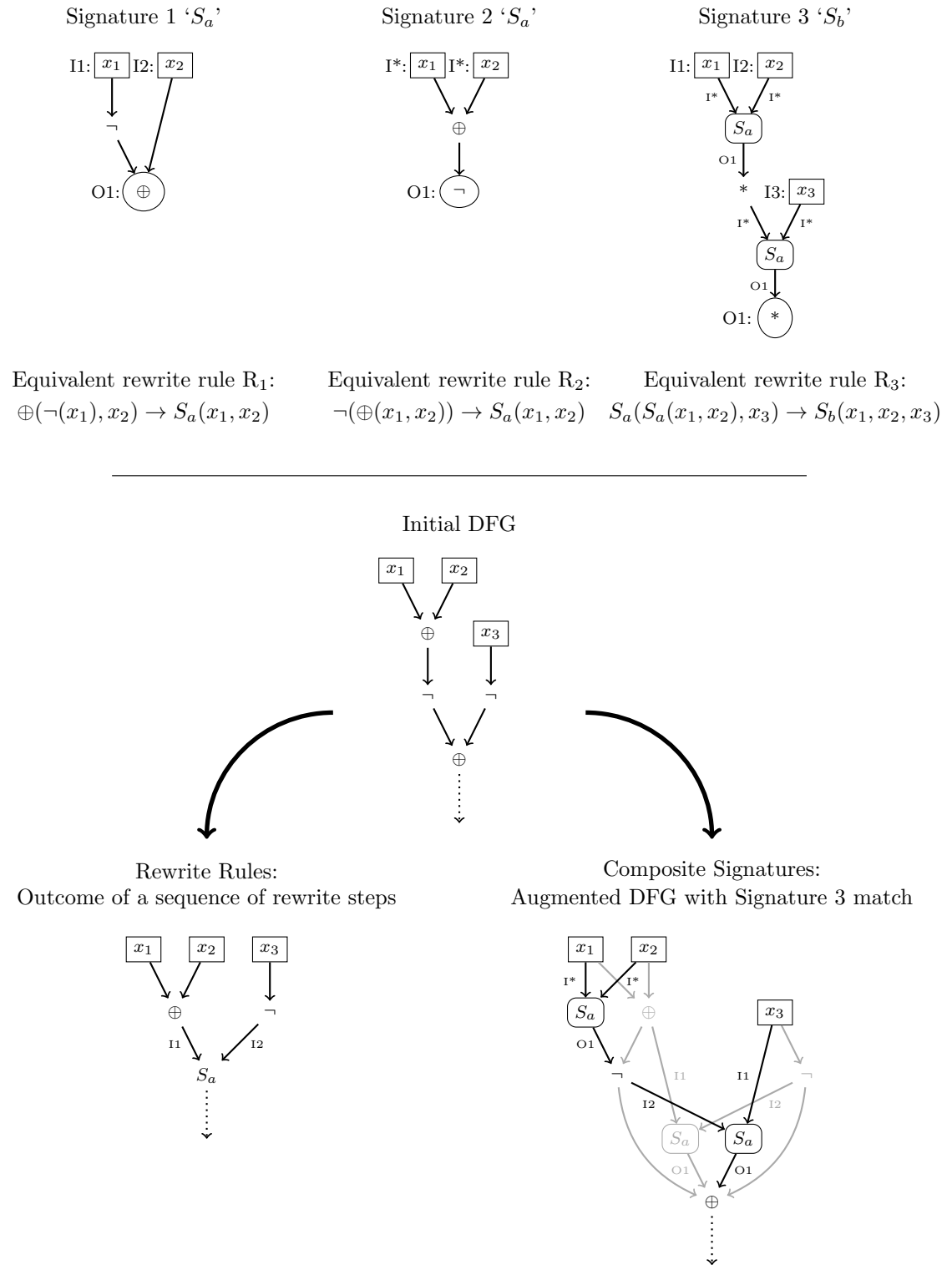

Figure 4.2: Comparison between rewrite rules and composite signatures. At the top we give three signatures and their equivalent rewrite rules. The first two are non-composite signatures and they both export the symbol 'S_a'. The third is a composite signature and it contains two vertices labelled with the 'S_a' symbol. The DFG depicted in the middle is processed in two different ways: on the left we use rewrite rules and on the right we use signatures. The graph on left is the outcome of a possible sequence of rewrite steps (R_1). The graph on right is the augmented DFG where the matches of the first two signatures have been pushed.

implementations of a given primitive converge to a small set of normal forms? Second, can we efficiently detect large expressions using subgraph isomorphisms?

We have implemented a prototype of our solution. This prototype is divided into two parts. The first part collects execution traces. It is based on the PIN DBI framework and works on both Linux and Windows. The second part analyses execution traces. It performs everything from DFG construction to subgraph isomorphism. It is written in C and relies on the XED library [40] to disassemble execution traces. For the time being it can only analyse Intel Architecture 32-bit (IA-32) execution traces.

This prototype was evaluated against a large set of test programs. These test programs, also called synthetic samples, cover five different cryptographic primitives, namely AES, MD5, RC4, SHA1 and XTEA and for each primitive several implementations and a broad range of compilation conditions. The results presented in this section are limited to synthetic samples. For results on real life programs refer to Chapter 6.

4.3.1 Description of the Set of Synthetic Samples

Synthetic samples are very simple programs. They focus on a single cryptographic primitive, that they execute a small number of times. Compared to more realistic programs, synthetic samples provide well-controlled environments to extensively test our solution. It is easy to create a large amount of synthetic samples to thoroughly investigate if a given parameter has an influence on the detection rate. Because our method takes only preselected trace segments as input, the detection rate is not supposed to change significantly moving from synthetic samples to real life programs[7]. For each primitive, we created two kinds of synthetic samples.

The first kind of synthetic samples results from a publicly available source code implementation that we compiled using different compilers and different optimization levels. We used three compilers, namely GCC, Clang and MSVC, and at least two optimization levels per compiler. The objective is to evaluate the influence of the compiler and of its optimization level on the detection rate. For AES we used Gladman's implementation [31]. Gladman's AES implementation is written in C and can be configured in many different ways. We selected four different configurations, noted respectively V_0, V_1, V_2 and V_3. Their characteristics are detailed in Table 4.2. For MD5 we used a C implementation of our own. It is almost the same than the one given in the appendix of the RFC [62], except that it uses the slightly more efficient version of the first and second round function. For RC4 we used a C implementation of our own. For SHA1 we used the C implementation given in the RFC [6]. And finally for XTEA we used the C source code given on its Wikipedia web page[8].

The second kind of synthetic samples are made of well known cryptographic libraries that were used as distributed in their respective Debian package (that is to say, we did not recompile them). The objective is to test our solution on real cryptographic binaries and to show its resilience to source code variation. The list of the cryptographic libraries that was used, as well as the specific functions that were analysed for each of them, is given in Table 4.3.

The following listing specifies the actual computation that is performed by each synthetic samples.

- The AES synthetic samples encrypt and decrypt one block of data for the three standardized key lengths: 128, 192 and 256 bits.
- The MD5 synthetic samples hash an 80-byte message. Due to the message size, the compression function is executed two times.
- The RC4 synthetic samples encrypt a 12-byte message with a 4-byte key. Note that the size of the key is not particularly important here, since our tests only target the Pseudo Random Generation Algorithm (PRGA) and not the key schedule algorithm.

[7] It is true that dealing with larger programs puts more pressure on the segment selection method. But assuming that the segment selection method is working correctly, we should obtain similar segments whenever a primitive implementation is executed from a synthetic sample or from a real piece of software. And if the primitive was correctly detected in the first situation, there is no reason why it should not be correctly detected in the second situation.

[8] `http://en.wikipedia.org/w/index.php?title=XTEA&oldid=618892433` This implementation is adapted from the source code that was initially published by Needham *et al.* [78].

Table 4.2: Description of the synthetic samples based on Gladman's AES implementation.

	Unrolling	Table
Gladman V_0	Full	4 Tables
Gladman V_1	Partial	4 Tables
Gladman V_2	None	4 Tables
Gladman V_3	Full	1 Table

- The SHA1 synthetic samples hash a 112-byte message. Due to the message size, the compression function is executed two times.
- The XTEA synthetic samples encrypt and decrypt one block of data.

4.3.2 Description of the Set of Signatures

In this section we describe the set of signatures that we used for our experiments. Signature names are usually structured in two parts separated by an underscore. The first part specifies the coverage of the signature and the second part its version. A signature is declined into different versions when the portion of the algorithm it covers has several normal forms. In this regard, describing the different versions of a signature helps to understand the weakness of the normalization process. Multiplying the number of signatures is a way to artificially boost the detection rate. But it will go against our objective which is to show that with only a limited number of signatures we can detect a wide range of primitive implementations. Conversely, relying on too few signatures artificially deteriorates the detection rate. It will be difficult to extrapolate the detection rate for more realistic sets of signatures. We hope that we have found, for each primitive, the right number of signatures to convincingly evaluate our method.

We use the term *layer* to describe the organization of composite signatures. Every signature is assigned to a layer. A signature imports symbols defined by the signatures of lower layers and exports its symbol to signatures of higher layers.

AES. We use a set of fourteen signatures to detect AES (encryption and decryption). These signatures are detailed in Table 4.4. They are organized in four layers. The first layer contains signatures that cover one round of AES. The higher layers contain composite signatures that cover 10, 12 and 14 rounds of AES respectively. Composite signatures play a key role here. In fact for reasons that are discussed below, an AES round has many normal forms, and different normal forms can occur within a single AES implementation. Testing extensively every possible combination would have required many more signatures. Note that all these signatures only target table implementations of AES.

The number of tables differs between AT4 and AT1_V* signatures. The most common implementations of AES use four 256×32 bits lookup tables. These four lookup tables are rotations of each other. To save memory and to mitigate timing attacks, some implementations only keep a single table and use rotation operations to compute the result of the other lookups. AT1_V1 and AT1_V2 were created to deal with those single table implementations. Depending on the table that is kept two different normal forms are obtained. To detect both we created two versions of this single table signature.

Signatures AL_V1 to AL_V8 specifically target the last round. Their specificities are presented in Table 4.5. There are two reasons why we need such a large amount of signatures to detect the last round of AES.

- There is no *MixColumns* in the last round. The original substitution box S seems perfectly sufficient to fulfil any reasonable performance requirements. There is no clear reason to use large lookup tables. Yet to save a few shift operations or to guarantee aligned memory accesses, many implementations have devised their own custom table format for the last round. Unfortunately, these different table formats often result in different normal forms. For instance, let us consider the following two code snippets.

Table 4.3: List of cryptographic libraries which are involved in the set of synthetic samples. For each of them, the exact functions that were analysed, as well as the programming language they are written in, are specified.

AES	
Botan 1.10.8	C++
	`AES_128::encrypt_n, AES_128::decrypt_n, AES_196::encrypt_n, AES_196::decrypt_n, AES_256::encrypt_n, AES_256::decrypt_n,`
Crypto++ 5.6.1	ASM, C++
	`Rijndael_Enc_AdvancedProcessBlock, Rijndael::Dec::ProcessAndXorBlock`
Nettle 2.7.1	C
	`nettle_aes_encrypt, nettle_aes_decrypt`
OpenSSL 1.0.1t	ASM
	`AES_encrypt`
TomCrypt 1.17	C
	`rijndael_ecb_encrypt, rijndael_ecb_decrypt`
MD5	
Botan 1.10.8	C++
	`MD5::compress_n`
Crypto++ 5.6.1	C++
	`IteratedHashBase<unsigned int, HashTransformation>::HashMultipleBlocks`
Nettle 2.7.1	C
	`_nettle_md5_compress`
OpenSSL 1.0.1t	ASM
	`MD5_Update, MD5_Final`
TomCrypt 1.17	C
	`md5_process, md5_done`
RC4	
Botan 1.10.8	C++
	`ARC4::generate`
Crypto++ 5.6.1	C++
	`ARC4_Base::ProcessData`
Nettle 2.7.1	C
	`nettle_arcfour_crypt`
OpenSSL 1.0.1t	ASM
	`RC4`
TomCrypt 1.17	C
	`rc4_read`
SHA1	
Botan 1.10.8	C++
	`SHA_160::compress_n`
Crypto++ 5.6.1	C++
	`SHA1::Transform`
Nettle 2.7.1	C
	`_nettle_sha1_compress`
OpenSSL 1.0.1t	ASM
	`SHA1_Update, SHA1_Final`
TomCrypt 1.17	C
	`sha1_process, sha1_done`
XTEA	
Botan 1.10.8	C++
	`XTEA::encrypt_n, XTEA::decrypt_n`
Crypto++ 5.6.1	C++
	`XTEA::Enc::ProcessAndXorBlock, XTEA::Dec::ProcessAndXorBlock`
TomCrypt 1.17	C
	`xtea_ecb_encrypt, xtea_ecb_decrypt`

Table 4.4: List of AES signatures.

Name	Coverage	Exported Symbol	Imported Symbol(s)
AT1_V1, AT1_V2	1 **A**ES round, **1 T**able	A1	-
AT4	1 **A**ES round, **4 T**ables	A1	-
AL_V1, ..., AL_V8	**L**ast **A**ES round	A1	-
A10	**10 A**ES rounds	A10	A1
A12	**12 A**ES rounds	A12	A1, A10
A14	**14 A**ES rounds	A14	A1, A12

```
/* Code Snippet 1 */
static const uint8_t S[256] = {
        0x63, 0x7c, ...
        ..., 0xbb, 0x16};

uint32_t x = (uint32_t)S[i];
```

```
/* Code Snippet 2 */
static const uint32_t table[256] = {
        0x00000063, 0x0000007c, ...
        ..., 0x000000bb, 0x00000016};

uint32_t x = table[i];
```

They both assign the same value to variable `x`. The second code snippet is quite representative of the so-called optimizations that are used in practice for the last round. However their normal forms differ. In the second code snippet there is a $\ll$ operation that does not exist in the first, and in the first code snippet there is a `movzx` that does not exist in the second. For the time being there is no normalization mechanism capable of eliminating this kind of differences.

- Another source of variations is the formula that is used to merge the results of four lookups into a 32-bit word. This formula depends on the table format, but also on the compiler and on its optimization level. In this respect, we have identified two types of expressions which are not normalized in a satisfactory manner even though they result from different compilations of the same source code.

  The first one is related to the distribution of left shifts over bitwise XORs. It is illustrated by signatures AL_V1 and AL_V3. If we were able to distribute left shifts over bitwise XORs, signature AL_V3 would converge to signature AL_V1. Conversely, if we were able to factorize some of the terms of AL_V1 by 2^8 it would converge to AL_V3. Unfortunately, inserting a generic rewrite rule to either distribute or factorize bit shifts over logical bitwise operations raises many concerns. Unlike the constant distribution normalization mechanism, such a rewrite rule would not directly result in expression simplification. On the contrary, it would possibly lead to a significant increase of DFG size. But even more concerning it would strongly affect the structure of many expressions and blatantly break the context-insensitivity property. If we factorize, bitwise XORs will be moved upward and if we distribute, bitwise XORs will be moved downward. In any case bitwise XORs that are located on a signature boundary (such as the first and last *addRoundKey* for AES) will be mixed with any surrounding bit shit operation (coming for instance from an non-optimized endianness shift). With those difficulties in mind, we chose not to implement a generic rewrite rule to distribution or factorize bit shifts over logical bitwise operations. Of course it remains possible to add an ad hoc rewrite rule that would directly replace the merge expression of AL_V1 by the merge expression of AL_V3 (or the other way around).

  The second type of expressions is related to the equivalence between bitwise OR and bitwise XOR when they are applied on variables with disjoint sets of active bits. We say that a bit is active if there is at least one assignment for which it is equal to 1. Conservatively, we consider that a bit is active if we cannot prove that it is equal to 0 for all the assignments. This problem is illustrated by signatures AL_V1 and AL_V2. To solve it we tried to insert the following rewrite rule.

$$\oplus(x_1, x_2) \rightarrow \vee(x_1, x_2) \text{ if } D(\wedge(x_1, x_2)) = \{0\}$$

Table 4.5: Specificities of each of the eight versions of the last AES round signature.

Name	Table Size	Access Size	Merge Expression
AL_V1	256×8	8 bits	$\oplus(x_1, \ll(x_2, 8), \ll(x_3, 16), \ll(x_4, 24))$
AL_V2	256×8	8 bits	$\vee(x_1, \ll(x_2, 8), \ll(x_3, 16), \ll(x_4, 24))$
AL_V3	256×8	8 bits	$\oplus(x_1, \ll(\oplus(x_2, \ll(\oplus(x_3, \ll(x_4, 8)), 8)), 8))$
AL_V4	256×32	16 and 32 bits	$\vee(\oplus(x_1, x_2), \ll(\oplus(x_3, x_4), 16))$
AL_V5	256×32	8 and 32 bits	$\oplus(x_1, x_2, x_3, x_4)$
AL_V6	256×32	8 bits	$\oplus(x_1, \ll(x_2, 8), \ll(x_3, 16), \ll(x_4, 24))$
AL_V7	256×32	32 bits	$\oplus(x_1, \ll(x_2, 8), \ll(x_3, 16), \ll(x_4, 24))$
AL_V8	256×32	32 bits	$\oplus(x_1, \ll(\oplus(x_2, \ll(\oplus(x_3, \ll(x_4, 8)), 8)), 8))$

However because we use RICs to represent sets of reachable values, this rewrite rule is not convergent. Let us consider the following expression as a counterexample: $\oplus(x_1, \ll(x_2, 8), \ll(x_3, 16))$ and let us assume that $D(x_1) = D(x_2) = D(x_3) = [\texttt{0x0}, \texttt{0xff}]$. A first possible sequence of rewrite steps is as follows.

$$\begin{aligned}
\oplus(x_1, \ll(x_2, 8), \ll(x_3, 16)) &\rightarrow \oplus(\vee(x_1, \ll(x_2, 8)), \ll(x_3, 16)) \\
&\quad (D(\ll(x_2, 8)) = \{2^8 x |\ x \in [\texttt{0x0}, \texttt{0xff}]\} \text{ and } D(\wedge(x_1, \ll(x_2, 8)), \ll(x_3, 16)) = \{0\}) \\
&\rightarrow \vee(\vee(x_1, \ll(x_2, 8)), \ll(x_3, 16)) \\
&\quad (D(\vee(x_1, \ll(x_2, 8))) = [\texttt{0x0}, \texttt{0xffff}] \text{ and } D(\wedge(\vee(x_1, \ll(x_2, 8)), \ll(x_3, 16))) = \{0\}) \\
&\rightarrow \vee(x_1, \ll(x_2, 8), \ll(x_3, 16))
\end{aligned}$$

In that case the reduction system behaves as expected. In particular, if we extend this rewrite sequence to the merge expression of AL_V1 it will converge to AL_V2. However, a second possible sequence of rewrite steps is as follows.

$$\begin{aligned}
\oplus(x_1, \ll(x_2, 8), \ll(x_3, 16)) &\rightarrow \oplus(\ll(x_2, 8), \vee(x_1, \ll(x_3, 16))) \\
&\quad (D(\ll(x_3, 16)) = \{2^{16} x |\ x \in [\texttt{0x0}, \texttt{0xff}]\} \text{ and } D(\wedge(x_1, \ll(x_3, 16))) = \{0\})
\end{aligned}$$

No more rewrite steps are possible since $D(\vee(x_1, \ll(x_3, 16))) = [\texttt{0x0}, \texttt{0xffffff}]$. RICs are not well suited to represent the set of reachable values of $\vee(x_1, \ll(x_3, 16))$. We have to perform a large over-approximation to represent this set by a RIC. Unfortunately, based on this over-approximation we cannot conclude that $\ll(x_2, 8)$ and $\vee(x_1, \ll(x_3, 16))$ have disjoint sets of active bits. RICs were introduced to accurately over-approximate additions and bit shifts (they are the most common operations in address expressions), but they perform very poorly with logical bitwise operations such as $\wedge$, $\neg$, $\vee$ and $\oplus$. In this example, a good way to over-approximate those operations would be to use an active bit mask. The idea is not to definitely replace RICs by active bit masks since they are much less accurate in many situations, but it is to use them simultaneously. Thereby depending on the type of operations, we could switch between the different representations to always rely on the most accurate one. None of this has been implemented though and the rewrite rule was not included in the normalization process since at the moment it raises as many issues as it actually solves.

As a partial conclusion, the two situations that we have described (distribution of bit shifts over logical bitwise operations and replacement of bitwise XORs by bitwise ORs) are due to weaknesses of the normalization process which are yet unaddressed. We had no other choice but to increase the number of signatures to correctly detect the last round of AES.

Note that none of the signatures A10, A12 and A14 covers the first *AddRoundKey*. In fact as explained in Section 5.1.3, lack of context at the beginning of the signature makes it very difficult to break the symmetries of *AddRoundKey*. As a consequence, inserting the first *AddRoundKey* in a signature will lead to many problematic false positive detections.

MD5. We use nine signatures to detect MD5. They are organized in two layers. The first layer contains the following signatures:

- M1_V1 and M1_V2 cover one execution of the step function used in the first round ;
- M2_V1 and M2_V2 cover one execution of the step function used in the second round ;
- M3_V1 and M3_V2 cover one execution of the step function used in the third round ;
- M4_V1 and M4_V2 cover one execution of the step function used in the fourth round.

Every step function includes a rotation. Because there is no rotation operator in the C language, the easiest way to implement a rotation is to use a right shift, a left shift and a bitwise OR. We will refer to this expression as the expanded form of the rotation. Some compilers, with the right optimization level, recognize the expanded form of the rotation and translate it into the rotation instruction of the x86 instruction set. But compilers do not always perform this translation and sometimes they keep the expanded form in the assembly code they produce. To fix this issue, our initial idea was to perform the missing transformation in the normalization phase. It can be achieved using the following rewrite rule.

$$\vee(\ll(x, c_1), \gg(x, c_2)) \rightarrow \circlearrowleft(x, c_1) \text{ if } c_1 + c_2 = size$$

Unfortunately, in the case of MD5, this rewrite rule happened to be ineffective. In fact, the x variable in the above formula corresponds, in the case of MD5, to a sum that involves a constant term (refer to Section A.1.2). According to the constant distribution normalization mechanism, the left shift contained within the expanded form of the rotation is distributed over the addition. The result cannot be identified by the above rewrite rule. A second possibility would be to perform the inverse transformation, that is to say, to replace the rotation operation by its expanded form. We have not investigated this second option though. Instead we simply created two signatures for each step function: M*_V1 to detect implementations that use the rotation instruction and M*_V2 to detect implementations that use its expanded form.

The second layer contains a single signature called MC. It covers the **M**D5 **C**ompression function. It was necessary to use a composite signature to detect MD5. Since each step has two different normal forms, the overall number of combinations for the 64 steps is 2^{64}. It is obviously impossible to test 2^{64} signatures. Thanks to composite signatures we only have to double the number of step signatures.

RC4. We use four signatures to detect RC4. They are called respectively RP_V1, RP_V2, RP_V3 and RP_V4. They all cover one iteration of the **R**C4 **P**seudo random generation algorithm. To explain why four signatures are necessary to detect a wide range of RC4 implementations, let us consider the following C code snippet.

```
j += S[++i];
tmp = S[i];
S[i] = S[j];
S[j] = tmp;
key_stream = S[(S[i] + S[j]) & 0xff];
/* [...] */
```

This code is a possible implementation of the loop body of the PRGA. This is the part of the algorithm that is covered by our four signatures. Variables `i` and `j` are two 8-bit internal counters and variable `S` is the permutation array. We have identified three causes of variation in the normalized DFG of the PRGA. They are detailed below.

- Internal counter initialisation. This variation is due to segment selection. Variables `i` and `j` are initialised to zero. If the trace segment contains this initialisation step, constant folding will simplify the terms $+(S, i_0)$ and $+(j_0, \texttt{load}(S))$ for the first iteration. Furthermore, since variable `i` is incremented by a fixed amount at each iteration, the term $+(S, i_n)$ will also be simplified for every iteration n.

- The size of the permutation representation. This variation is due to a difference at the source code level. Most implementations use an 8-bit array, but some others use a 32-bit array. As a consequence the address expressions that are used to access the permutation differ (an additional $\ll$ is necessary to address a 32-bit array). This issue is out of the scope of our normalization mechanisms.
- Redundant memory `load`. This variation is due to a difference at the source code level. The sequence of memory operations for the C code snippet given above is as follows[9]:

  $$(\texttt{load}_1(i), \texttt{load}_2(i), \texttt{load}_3(j), \texttt{store}_4(i), \texttt{store}_5(j), \texttt{load}_6(i), \texttt{load}_7(j))$$

  This sequence of memory operations can be simplified according to the memory access simplification mechanism, and we finally obtain:

  $$(\texttt{load}_1(i), \texttt{load}_3(j), \texttt{store}_4(i), \texttt{store}_5(j), \texttt{load}_6(i))$$

  $\texttt{load}_6$ cannot be simplified. In fact, `j` is a possible alias for `i` and thus it is perfectly correct not to simplify $\texttt{load}_6$. But if `i` = `j`, according to the rest of the algorithm the value written by $\texttt{store}_4$ will be equal to the value written by $\texttt{store}_5$. Consequently, performing the simplification will nevertheless preserve the observable similarity. This reasoning is far out of the reach of our normalization mechanisms though. Some RC4 implementations, to save one memory read, manually perform this simplification by replacing the last line of the C code snippet by: `key_stream = S[(tmp + S[i]) & 0xff];`

SHA1. The tests presented in this chapter are limited to the message schedule[10]. We use two signatures, namely SMS_V1 and SMS_V2, to detect the **S**HA1 **M**essage **S**chedule. The difference between these two signatures is due to a variation at the source code level. In [51] optimizations that make a better use of SIMD instructions are presented. Let W_i denote the i^{th} expanded message word. The original formula to compute W_i is as follows:

$$\circlearrowleft (\oplus(W_{i-3}, W_{i-8}, W_{i-14}, W_{i-16}), 1)$$

One of the sore point of this formula is the dependency between W_i and W_{i-3}. It prevents straightforward vectorization with four-element SIMD instructions. A possible optimization is to compute W_i using the following equivalent formula:

$$\circlearrowleft (\oplus(W_{i-6}, W_{i-16}, W_{i-28}, W_{i-32}), 2)$$

Unfortunately implementations resulting from this last formula are not detected by SMS_V1 which was designed to detect implementations based on the original formula. The root cause is that we do not distribute rotations over bitwise XORs during the normalization phase. This issue has already been discussed in the paragraph dedicated to AES signatures. In the case of SHA1, we simply created a new signature called SMS_V2 to address this issue.

XTEA. We use six signatures to detect XTEA. Even though it was not strictly necessary in this case[11], we created two layers of signatures. The first layer contains two signatures, one for encryption and one for decryption named XE1 and respectively XD1. They both cover one cycle. They have three input arguments and one output argument. The second and the third input arguments are round keys. As a reminder, the round keys of round $2i-1$ and $2i$ are equal to `s + key[(s >> 11) & 3]` and `s + key[s & 3]` respectively. The key schedule of XTEA is extremely simple. In practice it is implemented either directly in the encryption/decryption function, and as such it is executed every time a block is processed, or in a separate function that

[9] For simplicity we have omitted variable `S`. Every address is the sum of `S` plus either `i` or `j`, thus removing variable `S` does not affect the reasoning.

[10] Originally, the SHA1 message schedule was used to test the memory access simplification mechanism at the time it was developed. In fact, in most SHA1 implementations, expanded message words are stored in a large memory buffer. This memory buffer is extensively accessed through mixed `load` and `store`. We assume that the rest of the compression function is similar to the MD5 compression function. And therefore we could expect similar results.

[11] In practice a cycle of XTEA has a unique normal form. Thus searching for cycles independently of the rest of the algorithm does not help to reduce the number of signatures which need to be searched.

is executed only once for all blocks. In this latter scenario, there is no guarantee that the key schedule will be contained within the trace segment. Therefore XTEA signatures do not cover the key schedule. The second layer contains four signatures, two for **E**ncryption and two for **D**ecryption noted respectively XE32_V1, XE32_V2, XD32_V1 and XD32_V2. They all cover **32** cycles. The differences between the two versions of both the encryption and the decryption signature are due to segment selection and to the key schedule implementation. There are some values of the counter `s` such that `(s >> 11) & 3` is equal to `s & 3`. When it happens the $2i - 1^{\text{th}}$ round key is equal to the $2i^{\text{th}}$ round key. Assuming that the counter initialisation and the key schedule are part of the trace segment, the counter will be replaced by a constant for every round according to the constant folding mechanism, and the round keys which are equal will be pruned according to the common subexpression elimination mechanism. As a result in signature XE32_V2 and XD32_V2, some of the round keys are shared by several rounds and the number of fragments of the round key argument input is reduced. Conversely, none of this will happen if either the counter initialisation or the key schedule is not part of the trace segment. It corresponds to signatures XE32_V1 and XD32_V1.

4.3.3 Results

In this section we present experimental results for primitive identification. We follow the same methodology for all the synthetic samples. First, execution traces are divided in terms of function executions. A function execution is a trace segment which starts immediately after a `call` instruction and ends with a `ret` instruction. Two dynamic instructions are separated by fewer `ret` instructions than `call` instructions if they belong to the same function execution. Then, we manually select the smallest function execution which contains the primitive that we want to detect.

We use the same format to present all the results. For brevity, only signatures belonging to the highest layer are reported. Green indicates a correct detection, orange indicates a partial detection and red indicates a missed detection. For partial detections we also specify what was the expected number of signature matches. Execution time measurements were performed on a Pentium Dual-Core T4200 processor. They correspond to the execution time of the complete analysis process, including both the normalization and the subgraph isomorphism.

AES. Primitive identification results for AES synthetic samples are given in Table 4.6. For each key length we analysed both the encryption and the decryption primitive. Every signature of the first layer (namely AT4, AT1_V1, AT1_V2 AL_V1, ... and AL_V8) has either 4 or 24 automorphisms (automorphisms are discussed in Section 5.1.3). There is no way to avoid these automorphisms. We did not count them as false positives though, because they all return the same correct set of vertices for each parameter. Only the ordering of the parameter fragments varies. Automorphisms are propagated to the upper layers. Consequently, signatures A10, A12 and A14 also have either 4 or 24 automorphisms. If we sum signature matches that are found for the encryption and for the decryption, we finally obtained a total of either 8 or 48 signature matches. We detail below the synthetic samples that were not correctly identified.

- Gladman V_3 Clang `-O0`, `-O2`, `-O3` & MSVC `-O0`. No signature was found for Clang `-O0` and MSVC `-O0` and round signatures are missing for Clang `-O2` and `-O3`. As indicated in Table 4.2, Gladman V_3 synthetic samples use a single large lookup table. The missing tables are replaced by rotations. As explained in Section 4.3.2 in the paragraph dedicated to MD5 signatures, depending on the compiler and on its optimization level, rotations are implemented using either their expanded form or a dedicated x86 instruction. Unfortunately, our normalization mechanisms are not able to eliminate this variation. None of the two signatures, AT1_V1 and AT2_V2, that are supposed to detect the single table implementation of AES, covers the expanded form of the rotation. Thus, if an AES round uses the expanded form of the rotation it will not be detected by our solution.

- Botan. The last round was not detected. For the last *AddRoundKey*, data is manipulated at the byte granularity. This case is not covered by any version of the last round signature. To remove this specificity we need normalization mechanisms able to vectorize bitwise XORs.

The only normalization mechanism that can vectorize operations is memory coalescing but it only affects `load` and `store`. In this regard, memory coalescing was not even possible in this case. The round key buffer and the output buffer are both accessed at the byte granularity and can therefore benefit from memory coalescing. But possible aliasing conflicts (mixed read access to the substitution box, the round key buffer, and write access to the output buffer) prevent any application of memory coalescing.

- Crypto++. We found 64 signature matches for the encryption where four were expected, and four signature matches for the decryption which is the correct result. The functions which implement AES encryption in Crypto++ also XOR the ciphertext with a given memory buffer. The bitwise XORs corresponding to that extra computation are merged with the bitwise XORs of the last *AddRoundKey* according to the commutative and associative operation normalization mechanism. It results in bitwise XORs with three operands: one corresponding to a fragment of the internal state, one corresponding to a round key and one corresponding to a fragment of the additional memory buffer. The subgraph isomorphism algorithm tries to match these bitwise XORs with those of the signature which only have two operands. There is two possible matchings per fragment and a total of 16 possible matchings per initial signature match ($64 = 16 \times 4$). These are considered to be true false positives since they provide incorrect information regarding the location of the last round key. This issue would also have affected the detection of the first round key for the decryption if the first *AddRoundKey* was covered by the signatures. In fact, in a symmetric way, the ciphertext is XORed with a given memory buffer before decryption.

- OpenSSL. No signature was found. The AES implementation provided in OpenSSL does not use large lookup tables. Instead, it explicitly computes the *xtime* transformation using SIMD instructions.

MD5. Primitive identification results for MD5 synthetic samples are given in Table 4.7. The compression function is executed twice per synthetic sample. Consequently we expect to find two matches of the MC signature per synthetic sample. The detection was only partial for all the synthetic samples based on our own implementation of MD5. Only the second execution of the compression function was correctly detected. As a reminder, the chaining value for the first message block is initialised with a fixed constant. If this initialisation happens within the trace segment, which is the case for all the synthetic samples based on our own implementation of MD5[12], due to constant folding, important simplifications will occur within the first two steps of the Feistel network. Consequently, no signature (neither M1_V1 nor M1_V2) will be found for these two steps and that will prevent the detection of the MC signature.

The first layer signatures that were detected are also mentioned in Table 4.7. We notice that for MSVC `-O2` both versions of the third step function signature are found simultaneously. This means that the third step function is, for some steps, implemented using the expanded form of the rotation and, for some other steps, implemented using the rotation instruction.

RC4. Primitive identification results for RC4 synthetic samples are given in Table 4.8. These synthetic samples encrypt a 12-byte message. Thus, we expect to find a total of 12 signature matches per synthetic sample. Results were successful except for the Botan library. In that case we had to stop the computation after one hour without getting any result. The RC4 implementation provided in Botan uses a buffering mechanism to XOR the key stream with the message. First, the library generates a key stream of 256 bytes, then this key stream is XORed with the message. As a first consequence the trace segment is very large: around 88000 dynamic instructions. It explains

[12]Usually hash function APIs are structured in three functions. The first one, often called `hash_init`, initialises some internal data structure such as the chaining value and the length counter. The second one, often called `hash_update`, takes a message of any length, split it into blocks and calls the compression function as many times as necessary. The third one, often called `hash_final`, pads the remaining data, calls the compression function one last time and returns the hash value. If a MD5 implementation follows this API structure, there will be little chance for `hash_init` to be included in the trace segment. But in our MD5 implementation we condensed these three functions into a single one. As a consequence, the initialisation of the chaining value is always included in the trace segment.

Table 4.6: Primitive identification for AES synthetic samples.

Source	Compiler	Opt.	Result 128	Result 196	Result 256	Time
Gladman V_0	$GCC_{4.9.2}$	-O0	A10:48	A12:48	A14:48	2.78 s
		-O1	A10:48	A12:48	A14:48	2.58 s
		-O2	A10:48	A12:48	A14:48	2.58 s
		-O3	A10:48	A12:48	A14:48	2.57 s
	$Clang_{3.5.0}$	-O0	A10:48	A12:48	A14:48	2.9 s
		-O1	A10:48	A12:48	A14:48	2.7 s
		-O2	A10:48	A12:48	A14:48	2.68 s
		-O3	A10:48	A12:48	A14:48	2.68 s
	$MSVC_{18.0}$	-O0	A10:48	A12:48	A14:48	2.52 s
		-O2	A10:48	A12:48	A14:48	2.35 s
Gladman V_1	$GCC_{4.9.2}$	-O0	A10:48	A12:48	A14:48	2.77 s
		-O1	A10:48	A12:48	A14:48	2.56 s
		-O2	A10:48	A12:48	A14:48	2.58 s
		-O3	A10:48	A12:48	A14:48	2.58 s
	$Clang_{3.5.0}$	-O0	A10:48	A12:48	A14:48	2.87 s
		-O1	A10:48	A12:48	A14:48	2.65 s
		-O2	A10:48	A12:48	A14:48	2.56 s
		-O3	A10:48	A12:48	A14:48	2.57 s
	$MSVC_{18.0}$	-O0	A10:48	A12:48	A14:48	2.73 s
		-O2	A10:48	A12:48	A14:48	2.36 s
Gladman V_2	$GCC_{4.9.2}$	-O0	A10:48	A12:48	A14:48	2.77 s
		-O1	A10:48	A12:48	A14:48	2.6 s
		-O2	A10:48	A12:48	A14:48	2.59 s
		-O3	A10:48	A12:48	A14:48	2.59 s
	$Clang_{3.5.0}$	-O0	A10:48	A12:48	A14:48	2.88 s
		-O1	A10:48	A12:48	A14:48	2.69 s
		-O2	A10:48	A12:48	A14:48	2.55 s
		-O3	A10:48	A12:48	A14:48	2.54 s
	$MSVC_{18.0}$	-O0	A10:48	A12:48	A14:48	2.75 s
		-O2	A10:48	A12:48	A14:48	2.53 s
Gladman V_3	$GCC_{4.9.2}$	-O0	A10:48	A12:48	A14:48	2.85 s
		-O1	A10:48	A12:48	A14:48	2.72 s
		-O2	A10:48	A12:48	A14:48	2.71 s
		-O3	A10:48	A12:48	A14:48	2.71 s
	$Clang_{3.5.0}$	-O0	∅	∅	∅	1.38 s
		-O1	A10:48	A12:48	A14:48	2.85 s
		-O2	A10:48	A10:48/192	A10:48/432	1.98 s
		-O3	A10:48	A10:48/192	A10:48/432	1.99 s
	$MSVC_{18.0}$	-O0	∅	∅	∅	0.92 s
		-O2	A10:8	A12:8	A14:8	2.31 s
Botan	-	-	AT4:384, AT1_V2:8	A10:56/104	A12:56/104	2.01 s
Crypto++	-	-	A10:68/8	A12:68/8	A14:68/8	2.93 s
Nettle	-	-	A10:8	A12:8	A14:8	2.36 s
OpenSSL	-	-	∅	∅	∅	0.82 s
TomCrypt	-	-	A10:48	A12:48	A14:48	2.43 s

Table 4.7: Primitive identification for MD5 synthetic samples. Name of the signatures that were detected in the lower levels are given in brackets.

Source	Compiler	Opt.	Result	Time
Own Source	$GCC_{4.9.2}$	-O0	MC:1/2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.16 s
		-O1	MC:1/2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.13 s
		-O2	MC:1/2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.13 s
		-O3	MC:1/2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.13 s
	$Clang_{3.5.0}$	-O0	MC:1/2 (M1_V2, M2_V2, M3_V2, M4_V2)	0.2 s
		-O1	MC:1/2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.13 s
		-O2	MC:1/2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.13 s
		-O3	MC:1/2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.13 s
	$MSVC_{18.0}$	-O0	MC:1/2 (M1_V2, M2_V2, M3_V2, M4_V2)	0.29 s
		-O2	MC:1/2 (M1_V2, M2_V2, M3_V1, M3_V2, M4_V2)	0.16 s
Botan	-	-	MC:2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.3 s
Crypto++	-	-	MC:2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.48 s
Nettle	-	-	MC:2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.17 s
OpenSSL	-	-	MC:2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.18 s
TomCrypt	-	-	MC:2 (M1_V1, M2_V1, M3_V1, M4_V1)	0.14 s

Table 4.8: Primitive identification for RC4 synthetic samples.

Source	Compiler	Opt.	Result	Time
Own Source	$GCC_{4.9.2}$	-O0	RP_V1:1, RP_V2:11	11 s
		-O1	RP_V1:1, RP_V2:11	9.6 s
		-O2	RP_V1:1, RP_V2:11	9.6 s
		-O3	RP_V1:1, RP_V2:11	9.87 s
	$Clang_{3.5.0}$	-O0	RP_V1:1, RP_V2:11	12.5 s
		-O1	RP_V1:1, RP_V2:11	10 s
		-O2	RP_V1:1, RP_V2:11	1.12 s
		-O3	RP_V1:1, RP_V2:11	1.29 s
	$MSVC_{18.0}$	-O0	RP_V1:1, RP_V2:11	11.2 s
		-O2	RP_V1:1, RP_V2:11	10 s
Botan	-	-	$\emptyset$	$>$ 1 h
Crypto++	-	-	RP_V3:12	0.42 s
Nettle	-	-	RP_V3:12	0.03 s
OpenSSL	-	-	RP_V4:12	0.1 s
TomCrypt	-	-	RP_V2:12	0.02 s

why the analysis did not return in a reasonable time. As a second consequence, if we try to reduce the size of the trace segment to capture just a few iterations of the PRGA, we will miss the final XOR. This final XOR is contained in every RC4 signatures and thus, without it, no signature will be found.

SHA1. Primitive identification results for SHA1 synthetic samples are given in Table 4.9. The compression function is executed twice per synthetic sample and so is the message schedule. Consequently we expect to find two matches of the SMS_V* signatures per synthetic sample. Results were successful except for RCF 3174 Clang `-O0` & MSVC `-O0`. No signature were reported for these two synthetic samples. When every optimization is disabled, Clang and MSVC keep the expanded form of the rotation in the assembly code they produce. This raises a normalization issue that has been discussed in Section 4.3.2. Unfortunately none of the two signatures employed to detect SHA1 targets the expanded form of the rotation.

XTEA. Primitive identification results for XTEA synthetic samples are given in Table 4.10. These synthetic samples encrypt and decrypt one block of data. Thus, we expect to find for each

Table 4.9: Primitive identification for SHA1 synthetic samples.

Source	Compiler	Opt.	Result	Time
RFC 3174	$GCC_{4.9.2}$	-O0	SMS_V1:2	0.26 s
		-O1	SMS_V1:2	0.25 s
		-O2	SMS_V1:2	0.25 s
		-O3	SMS_V1:2	0.23 s
	$Clang_{3.5.0}$	-O0	∅	0.01 s
		-O1	SMS_V1:2	0.22 s
		-O2	SMS_V1:2	0.22 s
		-O3	SMS_V1:2	0.22 s
	$MSVC_{18.0}$	-O0	∅	1.25 s
		-O2	SMS_V1:2	0.3 s
Botan	-	-	SMS_V1:2	0.35 s
Crypto++	-	-	SMS_V1:2	0.5 s
Nettle	-	-	SMS_V1:2	0.18 s
OpenSSL	-	-	SMS_V2:2	0.3 s
TomCrypt	-	-	SMS_V1:2	0.23 s

Table 4.10: Primitive identification for XTEA synthetic samples.

Source	Compiler	Opt.	Result	Time
Wikipedia	$GCC_{4.9.2}$	-O0	XE32_V2:1, XD32_V2:1	0.09 s
		-O1	XE32_V2:1, XD32_V2:1	0.08 s
		-O2	XE32_V2:1, XD32_V2:1	0.08 s
		-O3	XE32_V2:1, XD32_V2:1	0.08 s
	$Clang_{3.5.0}$	-O0	XE32_V2:1, XD32_V2:1	0.09 s
		-O1	XE32_V2:1, XD32_V2:1	0.08 s
		-O2	XE32_V2:1, XD32_V2:1	0.08 s
		-O3	XE32_V2:1, XD32_V2:1	0.08 s
	$MSVC_{18.0}$	-O0	XE32_V2:1, XD32_V2:1	0.1 s
		-O2	XE32_V2:1, XD32_V2:1	0.09 s
Botan	-	-	XE32_V1:1, XD32_V1:1	0.23 s
Crypto++	-	-	XE32_V2:1, XD32_V1:1	0.28 s
TomCrypt	-	-	XE32_V1:1, XD32_V1:1	0.07 s

synthetic sample, one match of an encryption signature and one match of a decryption signature. The detection was successful for all the synthetic samples.

4.3.4 Performance

In this section we present some detailed execution time measurements. The objective is not to thoroughly investigate the influence of a given parameter on the execution time. In fact, predicting the execution time of both, the normalization phase and the signature detection phase, seems to be a rather difficult problem. It is out of the scope of this work. All the execution time measurements were obtained with a Pentium Dual-Core T4200 processor[13].

The first series of results, given in Table 4.11, concerns the normalization phase. Columns correspond to trace segments and rows to normalization mechanisms. Trace segments were taken from the LibTomCrypt synthetic samples. For each trace segment we specify the size of the initial DFG. The theoretical complexity of many normalization mechanisms (refer to Table 3.1) is either linear or quadratic in the number of vertices. In practice though, the relation between the number of vertices and the execution time is less clear. For instance, the SHA1 trace segment is slightly larger than the AES trace segment, but it took less than half the time of the AES trace segment

[13]This processor is relatively old. It was commercialized in 2009. We could expect a performance increase by a factor of at least two with a more recent CPU.

Table 4.11: Detailed execution times for the normalization phase.

	AES_{256} Enc *(3209 vertices, 3747 edges)*	MD5 *(1327 vertices, 1801 edges)*	RC4 *(663 vertices, 795 edges)*	SHA1 *(3333 vertices, 4267 edges)*	XTEA Enc *(771 vertices, 1029 edges)*
Const. Folding	2 ms	1 ms	< 1 ms	3 ms	< 1 ms
Const. Expr. Detection	3 ms	1 ms	< 1 ms	5 ms	1 ms
Misc. Rewrite Rules	3 ms	1 ms	< 1 ms	3 ms	< 1 ms
Comm. Subexpr. Eli.	3 ms	1 ms	< 1 ms	5 ms	1 ms
Mem. Access Simpl.	121 ms	11 ms	3 ms	41 ms	6 ms
Const. Distribution	1 ms	< 1 ms	< 1 ms	1 ms	< 1 ms
Oper. Size Expansion	1 ms	< 1 ms	< 1 ms	2 ms	< 1 ms
Mem. Coalescing	7 ms	1 ms	< 1 ms	2 ms	< 1 ms
Affine Expr. Simpl.	3 ms	2 ms	< 1 ms	4 ms	1 ms
Const. Merging	< 1 ms	< 1 ms	< 1 ms	1 ms	< 1 ms
TOTAL	**152 ms**	**19 ms**	**6 ms**	**66 ms**	**10 ms**

to be normalized. We should bear in mind that normalization mechanisms are iteratively applied until a fixed point is eventually reached. Thus the execution time depends not only on the size of the DFG but also on the distance between the initial DFG and its normal form. Memory access simplification is by far the most time consuming normalization mechanism.

The second series of results, given in Table 4.11, concerns the signature detection phase. Columns correspond to trace segments and rows to signatures. We used the same trace segments as in Table 4.12. The number of vertices and edges are different though, because they correspond to the size of the DFGs after the normalization phase. Subgraph isomorphism is a well known NP-complete problem, but, as illustrated in Table 4.12, it can be solved efficiently in the majority of the cases encountered in our context. A grey cell indicates that the corresponding signature was not found. We notice that our tool quickly returns (≤ 1 ms) when there was no signature to be found. This is a reassuring result since it will be the most common scenario while testing large databases of signatures with weak segment selection heuristics. The AES trace segment was much slower than the other trace segments. This is not a surprising result. First the DFG associated with the AES trace segment is the largest. Second, the set of signatures used to detect AES has a relatively complex structure (signatures are organized into four layers) and covers a large amount of code. By comparison, the SHA1 trace segment has a DFG of similar size, but the SMS signature only covers a small portion of it (corresponding to the message schedule). And last but not least, AT4 has 24 automorphisms and so do A10, A12 and A14. This is the largest quantity of automorphisms that any signature used in those experiments has. The complexity of the subgraph isomorphism algorithm is, in the worst case, linear in the number of automorphisms.

Dividing large signatures into smaller ones using composite signatures may have a positive impact on performance. For instance, in Table 4.12, if we had used a single large signature per primitive instead of using several composite signatures as we did for AES, MD5 and XTEA, we would have obtained the following execution time measurements:

- 1.072 s in total for AES instead of 495 ms ;
- 29 ms in total for MD5 instead of 38 ms ;
- 84 ms in total for MD5 instead of 41 ms.

For both AES and XTEA, execution times were divided by a factor two, thanks to composite signatures. However as illustrated by MD5, the performance increase is not guaranteed. Sometimes, smaller signatures have fewer structural constraints leading to a greater number of automorphisms. And sometimes, the benefit of smaller signatures does not compensate the overhead implied by the composite signature mechanism.

4.3.5 Conclusion

Problems that were raised during the experimental evaluation are summarized in Table 4.13. They concern either signature versions that could have been avoided, or synthetic samples that were

Table 4.12: Detailed execution times for the subgraph isomorphism.

	AES-256 Enc *(1693 vertices, 2154 edges)*	MD5 *(802 vertices, 1206 edges)*	RC4 *(425 vertices, 545 edges)*	SHA1 *(1354 vertices, 2139 edges)*	XTEA Enc *(741 vertices, 997 edges)*
AT4	109 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms
A10	187 ms	-	-	-	-
A12	44 ms	-	-	-	-
A14	36 ms	-	-	-	-
M1_V1	< 1 ms	3 ms	< 1 ms	< 1 ms	< 1 ms
M2_V1	< 1 ms	5 ms	< 1 ms	1 ms	< 1 ms
M3_V1	< 1 ms	10 ms	< 1 ms	5 ms	< 1 ms
M4_V1	< 1 ms	1 ms	< 1 ms	< 1 ms	< 1 ms
MC	-	11 ms	-	-	-
RP_V2	< 1 ms	< 1 ms	1 ms	< 1 ms	< 1 ms
SMS_V1	< 1 ms	< 1 ms	< 1 ms	49 ms	< 1 ms
XE1	< 1 ms	< 1 ms	< 1 ms	< 1 ms	6 ms
XE32_V1	-	-	-	-	33 ms
TOTAL	**495 ms**	**38 ms**	**1 ms**	**59 ms**	**41 ms**

not correctly identified. We organized them into two groups. The first group contains problems which, in our opinion, can be solved with additional normalization mechanism(s). The second group however, contains problems that seem more difficult to solve with our current solution. It includes problems which are related to the context-insensitivity property for instance, such as unintended constant folding and unintended associative and commutative operation merging. Most of the problems listed in Table 4.13 are caused by variations at the source code level and not by compilers. At the end of Table 4.13 we gathered implementations which are difficult to normalize and which have a low probability of occurrence. It is almost impossible to efficiently detect those implementations with our solution. Here by efficient, we mean, without creating a dedicated set of signatures. The efficiency of our primitive identification method depends on the stability of primitive implementations.

From a performance perspective our solution is perfectly relevant. In average a primitive is detected in less than one second. The only exceptions are the AES synthetic samples and some of the RC4 synthetic samples. The AES synthetic samples are identified in approximately 2.5 seconds. But we should bear in mind that it is not one, but six primitives which are identified: three key lengths for both encryption and decryption. Thus these results are in fact compliant with our conclusion. Some of the RC4 synthetic samples took around 10 seconds to be identified. For a reason that has not been investigated the normalization phase was extremely slow for these synthetic samples (the signature detection took the usual time, that is to say around 15 ms).

Table 4.13: List of problems that were raised during the experimental evaluation.

MAY BE SOLVED WITH ADDITIONAL NORMALIZATION MECHANISM(S)

	Affected Signature(s)	Affected Synthetic Sample(s)
Distribution/Factorization of bit shifts over logical bitwise operations	AL_V3, AL_V8, SMS_V2	-
Equivalence OR/XOR	AL_V2	-
Rotation	M1_V2, M2_V2, M3_V2, M4_V2	AES Gladman V_3 Clang `-O0`, AES Gladman V_3 Clang `-O2`, AES Gladman V_3 Clang `-O3`, AES Gladman V_3 MSVC `-O0`, SHA1 RFC 3174 Clang `-O0`, SHA1 RFC 3174 MSVC `-O0`
Operation vectorization	-	AES Botan

DIFFICULT TO SOLVE WITH THE CURRENT SOLUTION

	Affected Signature(s)	Affected Synthetic Sample(s)
Unintended constant folding	RP_V1	MD5 Own Source GCC *, MD5 Own Source Clang *, MD5 Own Source MSVC *
Unintended associative and comm. operation merging	-	AES Crypto++
Unprovable aliasing	RP_V1, RP_V2	AES Botan
Memory data format	AL_V4, AL_V5, AL_V6, AL_V7, RP_V4	-
Uncommon API	-	AES Crypto++, MD5 Own Source GCC *, MD5 Own Source Clang *, MD5 Own Source MSVC *, RC4 Botan
Uncommon primitive implementation	SMS_V2	AES OpenSSL

Chapter 5

Dynamic Slicing for Mode of Operation Identification

Relying on primitive identification methods, in particular on DFG isomorphism, we devise a semi-automated solution to identify modes of operation. This solution returns a concise representation, called a slice, which summarizes the main data transfers occurring within an implementation of a mode of operation. The task of interpreting slices to identify modes of operation is the responsibility of human analysts.

In a first section, we present the requirements in terms of primitive identification. Then, we formally define a slice using syntactic concepts and discuss the advantages and disadvantages of this definition with respect to the notion of completeness and readability. In the following sections, we present adjustments based on semantics and we describe an approximation algorithm to compute slices. Finally, the last section of this chapter is dedicated to experimental results obtained for several modes of operation including CBC, HMAC and OCB.

Contents

5.1 Primitive Identification 105
5.1.1 Requirements 106
5.1.2 Subgraph Isomorphism 106
5.1.3 Detection Issues - Automorphism 107
5.2 Slicing Definition 108
5.2.1 Formal Definition 108
5.2.2 Completeness-Readability Trade-off 109
5.3 Adjustments Based on Semantics 112
5.3.1 Memory Operations 113
5.3.2 Fine-Grained Influence Tracking 114
5.4 Slice Construction 115
5.4.1 Light Normalization 115
5.4.2 Naive Algorithm 115
5.4.3 Greedy Algorithm 116
5.5 Experimental Evaluation 117
5.5.1 Extensive Evaluation on Basic Modes of Operation 117
5.5.2 Examples on More Complex Modes of Operation 119
5.5.3 Conclusion 122

5.1 Primitive Identification

At the beginning of this section, we detail the requirements for the primitive identification method. Then we justify, why the DFG isomorphism method described in the previous chapter meets

Table 5.1: Comparison of three main primitive identification methods based on the requirements of the mode of operation identification method.

	IOR	Avalanche Effect	DFG Isomorphism
Primitive Identity	Yes	No	Yes
Parameter Location	Yes	Yes	Yes
Data Flow Separation	Not directly	Not directly	Yes
False Positive	None	Numerous (Table 1.4)	Few (Section 5.1.3)

these requirements quite well. Finally, we describe a common issue, caused by automorphisms of signatures, which may affect the parameter localisation and, by extension, the mode of operation identification method.

5.1.1 Requirements

Given a segment of an execution trace, our mode of operation identification method needs the following information.

- The identity of each primitive executed within the trace segment. It includes the type of the primitives (for instance whether they are block ciphers, hash functions, stream ciphers, or key schedules) and ideally their name as well as their main characteristics (size of the key, number of rounds). This information is replicated in slices. It does not affect the inner computation of our mode of operation identification method. Nevertheless, it is of great importance for the value of slices. If primitive executions are not precisely identified, it will be harder for human analysts to interpret slices and to recognize modes of operation.

- The location of the input and output parameters of the primitives. Our solution summarizes the data transfers between the output and the input parameters of different primitive executions. Thus it is crucial to know the location of these parameters as a starting point for our solution. This information can either be expressed in terms of registers and memory locations in the execution trace or in terms of vertices in a DFG.

- The part of the data flow which corresponds to primitive executions. A trace segment is a sequence of dynamic instructions where every function call is inlined. In particular, large trace segments which are used to identify modes of operation, contain inlined primitive executions. It is essential to dissociate instructions (or edges in a DFG representation) belonging to primitive executions from those belonging to the mode of operation. Since we are only interested in the data transfers implemented by modes of operation, we must be able to exclude the data flow of primitives from our analysis.

Finally we assume that the primitive identification method is reliable. There is no filter or manual interaction between the output of the primitive identification method and the input of the mode of operation identification method. Thus, if the primitive identification method returns erroneous results they will be taken into account nonetheless by the mode of operation identification method.

5.1.2 Subgraph Isomorphism

A brief comparison of the three main primitive identification methods is given in Table 5.1. We choose to rely on DFG isomorphism to identify primitives. This method is relatively fast (the maximum execution time does not exceed a few seconds), efficient for non-obfuscated programs and it does not require heavy instrumentation. Furthermore, our mode of operation identification method uses the same DFG model. Thus, from an implementation perspective we can reuse part of the code that was developed for primitive identification, and from a performance perspective, we only have to create DFGs once for both methods. Finally, DFG isomorphism tells very precisely which vertices and edges belong to a primitive and which ones belong to the mode of operation.

DFG isomorphism returns augmented DFGs. In order to use augmented DFGs as input for our mode of operation identification method, we perform the following transformation. Let G be an augmented DFG.

1. We select all the matches of the signatures covering entire primitives.
2. For each selected match f of a signature S, we delete all the edges of $f_E(E_S)$ and all the vertices of $f_V(V_S \setminus I_S)$ in G. And for every vertex of $f_V(V_S \setminus I_S)$ labelled with a signature symbol resulting from the *push* transformation of a match f' of a signature S', we recursively delete $f'_E(E_{S'})$ and $f'_V(V_{S'})$ in G.
3. We delete all the vertices of G labelled with a signature symbol as well as the edges attached to them, including vertices resulting from *push* transformations of the selected matches.
4. For each selected match f, we mark every input parameter of f as final and label every output parameter of f with a variable symbol.

In Step 2, we delete part of the DFG corresponding to primitive executions. In Step 3, we delete irrelevant signature matches but also the selected matches. This last point is a technical detail. If we keep the vertices corresponding to the selected matches, there will be obvious paths going from their input parameter(s) to their output parameter(s) reflecting nothing but the internal data flow of primitives which will ruin the slice definition given in Section 5.2. Still, as explained at the beginning of Section 5.5, these vertices appear in the graphical representation of slices. Finally in Step 4 we try to preserve as much as possible the DFG syntax: input parameters may be left with no successor so they should be declared as final and output parameters may be left with no predecessor so they should be labelled as input variables.

As we can see in the last row of Table 5.1, DFG isomorphism sometimes produces false positives. They are caused by symmetries in signatures. This phenomenon is discussed in detail in the next section.

5.1.3 Detection Issues - Automorphism

An isomorphism from a DFG G to itself is called an automorphism. Let G be a DFG, S be a signature and f be an injective morphism from S to G. For every non-trivial automorphism h of S ($h \neq id$), $f \circ h$ is an injective morphism from S to G which differs from f. In other words, the number of matches of S in G is a multiple of the number of automorphisms of S.

Automorphisms are problematic with respect to the primitive and mode of operation identification problem, if they affect the input or output parameters. In that case, they generate signature matches with different mappings for the input or output parameters. Of course, in practice it is impossible to tell which detection corresponds to the correct mapping and which one is a false positive. An example of a signature with problematic automorphisms is given on the left hand side of Figure 5.1. This example is based on a simple toy cipher defined by: $C = S(M \oplus k_1) \oplus k_2$ where C is a ciphertext, S a public permutation implemented as a lookup table, M a plaintext and k_1 and k_2 two keys. The problematic automorphism maps M to k_1 and vice versa. Thus, it is impossible with this signature to dissociate the plaintext from the first key; this is going to be a serious problem for mode of operation identification.

Note that automorphisms often concern the input parameters since they lack operand constraints. A solution then, since we cannot modify the expressions of the primitive, is to add additional constraints on one of the parameter to break existing symmetries. This strategy is illustrated by the signature depicted in the middle of Figure 5.1. This enhanced signature works under the assumption that the keys are accessed through the same pointer. It has no symmetry any more. Yet, if we consider the DFG on right hand side of Figure 5.1, the enhanced signature will still produce false positive detections. In fact, all the three input parameters are read from the stack in a symmetric way. As a consequence there are two subgraphs which are isomorphic to the enhanced signature, each of them with different vertices for M and k_1. Again, we are unable to dissociate the plaintext from the key.

This issue happens for real cryptographic primitives such as AES for instance (because bitwise XOR are commutative, the first add round key has symmetries, and thus it is hard to correctly dissociate the plaintext from the first round key). In the case of AES, we applied the fix described

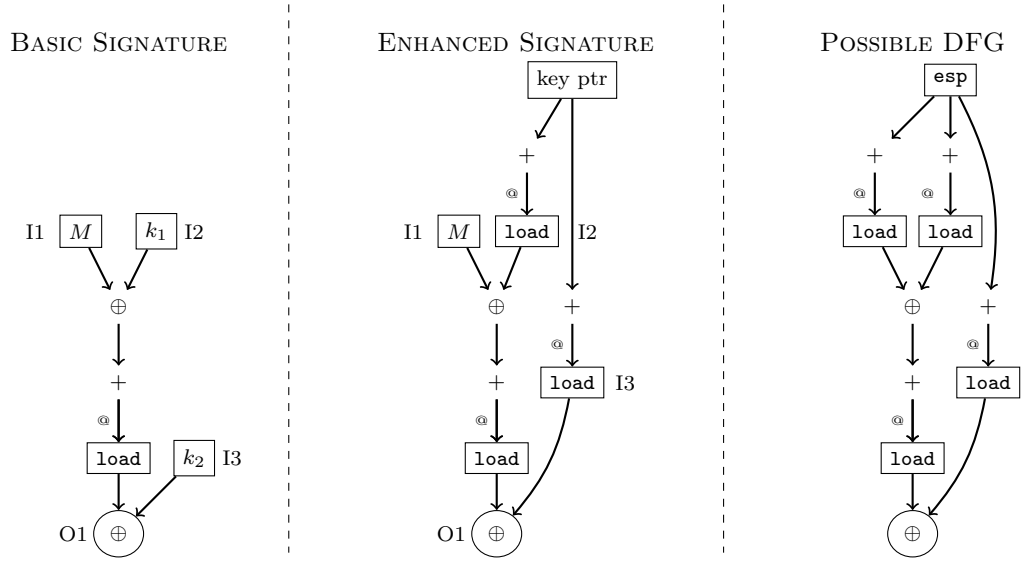

Figure 5.1: Two signatures and a possible DFG for the toy cipher $S(M \oplus k_1) \oplus k_2$. The graph on the left is a basic signature with a problematic automorphism. The graph in the middle is an attempt to fix the basic signature. And the graph on the right is a possible DFG which defeats this attempt.

above with moderate success. Another solution would be to filter incorrect parameters using the IOR method described in Chapter 1.

5.2 Slicing Definition

As motivated in the introduction, DFGs need to be simplified if they are to be manually interpreted by human analysts. To this end, we propose to extract pieces of DFGs which are connected to cryptographic parameters. Described as such, this step can be seen as a program slicing process. As in program slicing, our goal is to extract pieces of a program which are affected by or have an effect on points of interest (which are, in our case, the cryptographic parameters). But unlike usual definitions of program slicing [70], we do not impose the slice to maintain the semantics of the original program with respect to the points of interest. In fact, we favour readability over semantic equivalence. Thus, not every part of a DFG which is connected with a cryptographic parameter is included in the simplified graph. Because of the proximity to the program slicing domain, we borrow the terminology and call the simplified graph a slice. This section is structured as follows: first we give a formal definition of a slice and then we justify informally why this definition is a good compromise between completeness and readability.

5.2.1 Formal Definition

Given two vertices v and w in a directed graph G, an undirected path between v and w is defined by a sequence of edges $(e_i)_{1 \leq i \leq n}$ and a sequence of vertices $(u_i)_{0 \leq i \leq n}$ such that $v = u_0$, $w = u_n$ and the endpoints of e_i are equal to u_{i-1} and u_i for all $i \in \{1, ..., n\}$. We say that two vertices are connected if there is at least one undirected path between them. The length of an undirected path is equal to its number of edges. The distance between two connected vertices u and v in G, noted $d_G(u, v)$, is defined as the length of the shortest path between u and v. By extension, two sets of vertices V_1 and V_2 are connected if there is at least one vertex in each set $v_1 \in V_1$ and $v_2 \in V_2$ such that v_1 and v_2 are connected. And the distance between V_1 and V_2 is equal to $\min_{v_1 \in V_1, v_2 \in V_2}(d_G(v_1, v_2))$.

Definition 13 (Slice)**.** Given a DFG G and a set of cryptographic parameters PAR, a slice Γ is the smallest subgraph of G such that:

- every cryptographic parameter $P \in PAR$ has a least one vertex in Γ ;
- for any two cryptographic parameters P_1 and P_2 in PAR, if P_1 and P_2 are connected in G, then $P_1 \cap V_\Gamma$ and $P_2 \cap V_\Gamma$ are also connected in Γ and the distance between $P_1 \cap V_\Gamma$ and $P_2 \cap V_\Gamma$ in Γ is equal to the distance between P_1 and P_2 in G.

5.2.2 Completeness-Readability Trade-off

This section is an informal discussion to justify why the slice definition given above is a good trade-off between completeness and readability. A slice is complete if it is possible to accurately identify the mode of operation with the information it contains. A slice is readable if it does not contain significantly more information than what is strictly necessary to identify the mode of operation. If a slice is both complete and readable and on condition that the mode of operation is not too complex, a human analyst should have no trouble to interpret the slice and to identify the mode of operation.

Initial Approach

Initially slices were defined in a much less generic way. Instead of considering undirected paths between any two cryptographic parameters, we only considered paths with a predefined structure between specific cryptographic parameters. For instance, to correctly characterize CBC implementations, we first defined a slice Γ as the smallest subgraph of G which satisfies the following conditions.

- If there is a directed path from an output parameter O to an input parameter I in G, there is also a path from O to I in Γ and its length is equal to the length of the shortest directed path from O to I in G. With this condition we capture any chaining value between two executions of a block cipher such as in CBC.
- If two input parameters I_1 and I_2 have a common predecessor in G they also have a common predecessor in Γ and the distances between their lowest common predecessor and I_1, respectively I_2 are equal in G and in Γ. With this condition we capture any value which is shared by two input parameters such as the key in CBC.

But there is clearly a risk for this list of predefined connections to be incomplete and to become more and more complex as new modes of operation are considered. For instance in the DFG given in Figure 5.2, the part of data flow between the input parameter and the output of E_k cannot be summarized in terms of shortest directed paths or lowest common predecessors. To obtain complete slices without a priori knowledge of the types of connections that may be encountered, we adopt a more generic definition and we consider shortest undirected paths between any two cryptographic parameters. As a consequence, if two cryptographic parameters are connected in a DFG, they will also be connected in the slice. Unfortunately this is not sufficient to always obtain complete slices. For instance, if the relevant path to identify a given mode of operation is not a shortest path between a pair of cryptographic parameters, there is no guarantee that it will be reported in the slice. In the next paragraph this issue is illustrated by an example and a possible workaround is mentioned.

Connection-Preserving Subgraphs

It would make sense to define a slice as the smallest connection-preserving subgraph. The difference between distance-preserving and connection-preserving subgraphs is faint and only concerns a minority of modes of operation. Though, from a completeness perspective there is a slight advantage to prefer distance-preserving subgraphs over connection-preserving subgraphs. This advantage is illustrated in the example of Figure 5.3. This example is based on the `Rand_add` function of OpenSSL. This function is one of the principal component of the Random Number Generator (RNG) of OpenSSL. It is used to increase the entropy level of the RNG internal state given as input a memory buffer. The input memory buffer and the internal state are divided into 20-byte blocks noted respectively $b_0, ..., b_n$ and $s_0, ..., s_m$. c denotes a 32-bit constant. `Rand_add` mixes the input memory buffer with the internal state by iterating the following computation:

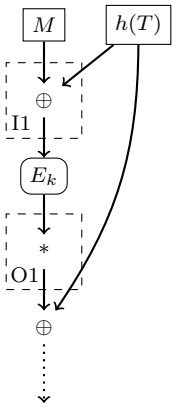

Figure 5.2: Example of a DFG where the data flow between the cryptographic parameters I1 and O1 cannot be reduced to a directed path nor to a common predecessor. The mode of operation which is depicted here, was introduced in [50] to construct a tweakable block cipher out of an ordinary block cipher. It behaves according to the following formula: $E_k(M \oplus h(T)) \oplus h(T)$, where E_k denotes a block cipher encryption under a key k, M a message bloc, h a hash function and T a tweak.

$$md_i \leftarrow SHA1(md_{i-1} \mid\mid b_i \mid\mid s_j \mid\mid c \mid\mid ...)$$
$$s_j \leftarrow s_j \oplus md_i$$

A simplified DFG of the `Rand_add` function is given on the left of Figure 5.3. In this example we only focus on the connections between the input and the output parameter of the SHA1 executions. These parameters are surrounded by dashed rectangles. If we define a slice as the smallest connection-preserving subgraph, the information regarding the counter c will be lost. This scenario is illustrated on the right hand side of Figure 5.3. Whereas if we define a slice as the smallest distance-preserving subgraph, the slice will contain both the internal state s_j and the counter c since the shortest path between the message and the output parameter of the first SHA1 execution goes through s_j and the shortest path between the message parameter of the two SHA1 executions goes through c. The smallest distance-preserving subgraph is sometimes larger than the smallest connection-preserving subgraph and thus it may be able to reveal connections that would have otherwise been concealed. But as discussed in Section 5.2.2, if the connections which are thereby revealed are redundant with already existing connections, there will be no completeness benefit but only a readability penalty.

Minimality Property

A slice is the smallest subgraph to preserve a certain relation between cryptographic parameters. This minimality property ensures readability. It has two practical consequences. First a slice does not contain superfluous elements, that is to say, edges or vertices which are not connected to at least two different cryptographic parameters. Second a slice does not contain redundant elements, that is to say, edges or vertices which are involved in a connection which is somehow already represented in the slice.

However, the minimality property may also cause some perfectly relevant elements to be discarded. As mentioned earlier, if relevant elements are not located on a shortest path between a pair of cryptographic parameters, they will not be included in the slice. This is illustrated by an example in Figure 5.4. On the left there is a possible DFG of a CTR implementation and on the right its corresponding slice. The counter is implemented using two variables, as it could be the case for a 128-bit counter on 64-bit architecture. For a large majority of executions, including the one depicted in the example, only the least significant part of the counter is incremented. Of the two existing paths between the input parameter of the two executions of the block cipher E_k, only

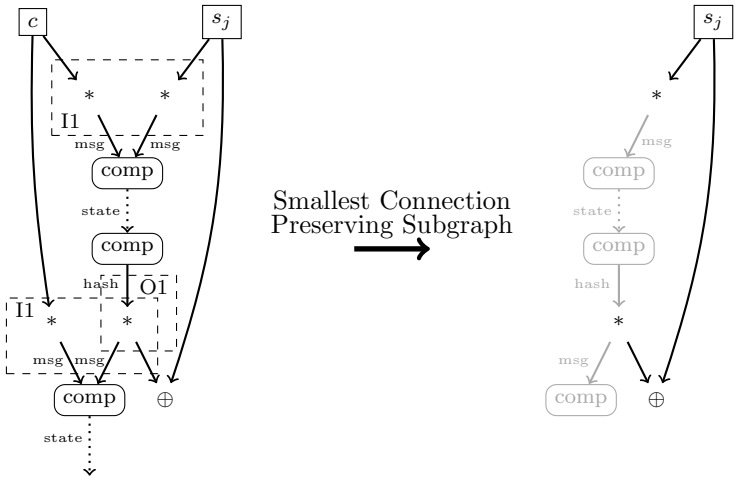

Figure 5.3: Simplified DFG of the loop body of the `Rand_add` function of OpenSSL and its smallest connection-preserving subgraph. The primitive noted comp is the SHA1 compression function. The smallest connection-preserving subgraph is depicted in black and part of the initial graph containing the primitive executions is recalled in light grey.

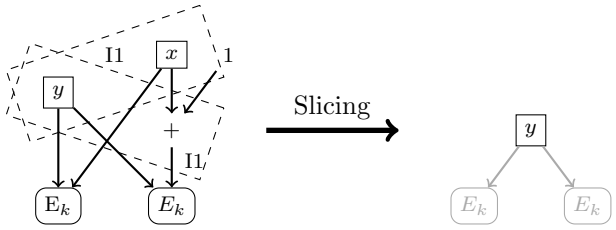

Figure 5.4: DFG of a CTR implementation and its corresponding slice. The counter is implemented using two variables: x is the least significant part and y is the most significant part. The input parameter of the two executions of E_k are surrounded with dashed rectangles.

the shortest one is included in the slice. Thus, the information about the addition, which would have been helpful to identify CTR, is lost.

Including every path and not only the shortest one is not a possible solution. In fact, some pairs of parameters are connected by numerous paths which are all strictly equivalent. For instance, the round key parameter of an AES-128 implementation is usually made of 44 32-bit words. Thus, there are at least 44 paths between two AES-128 executions which take as input the same round key buffer. To avoid redundant paths (representing the same information several times) we stick to the original slice definition. A possible workaround for the example of Figure 5.4 would be to split the input parameter of the block cipher into two parts. Since the two parts would be seen as independent parameters, both paths would be included in the slice (which as a matter of fact would be isomorphic to the original DFG). Obviously, this solution requires a priori knowledge of the mode of operation that is going to be identified. As such, it cannot be used directly in a first approach, but can be used during a refinement phase to get details about a particular connection.

Approximate Distance Preserving Subgraph

Distance preserving sometimes limits readability without providing any real completeness improvement in return. In the example of Figure 5.5 three cryptographic parameters P1, P2 and P3 depend on the same two input values. The directed paths from $\texttt{load}_1$ to P1 and from $\texttt{load}_2$ to P3 are altered by two operations: `movzx` and `part18`. These operations have no influence on the set of reachable values of P1 and P3 and thus are irrelevant to identify the mode of operation. But since they modify the distance between P1 and P3 they have an impact on the slice. P1 and P2 have two common predecessors in the slice. We consider that this slice is not perfectly readable since

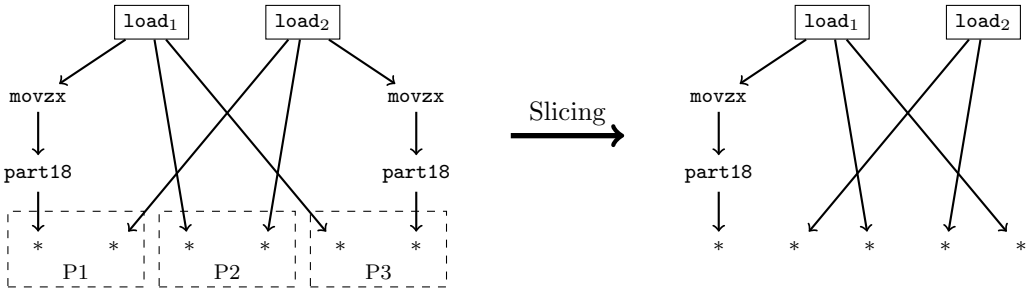

Figure 5.5: Example of a DFG where strict distance-preserving reduces the readability.

it contains duplicate information. A solution to this problem would be to consider approximate distance-preserving subgraphs instead of strict distance-preserving subgraphs. Given an approximation parameter α, a slice is α-distance-preserving if for any pair of cryptographic parameters P_1 and P_2, $|d_G(P_1, P_2) - d_\Gamma(P_1 \cap V_\Gamma, P_2 \cap V_\Gamma)| \leq \alpha$. Such a slice would probably be less impacted by noise. A typical noise example is the operations which are used to extract bytes of an architecture word or to swap endianness. These operations are not particularly interesting to identify modes of operation yet they slightly modify length of paths. Noise is frequent in DFGs used for mode of operation identification because they are not always completely normalized. In the example of Figure 5.5, a perfectly readable slice is obtained with an approximation parameter of two.

5.3 Adjustments Based on Semantics

This section is related to the notion of influence which was introduced in Section 3.5. As a reminder, given two vertices u and v, we say that u does not influence v, if for every assignment θ and every value $x \in \{0,1\}^{\text{size}(u)}$, $\theta(v)$ will be unaffected if we replace $\theta(u)$ by x. Conversely, we say that u influences v if there is an assignment θ and a value $x \in \{0,1\}^{\text{size}(u)}$ such that replacing $\theta(u)$ by x modifies $\theta(v)$. In practice influence is treated in a conservative way, meaning that if we cannot explicitly prove that u does not influence v we will consider that it influences v.

Our slice definition is purely based on the syntax of DFGs. In fact, it exclusively relies on the graph structure and does not contain any reference to the underlying operations. This definition has several advantages: it is concise and by putting constraints on the syntax it ensures maximal readability. Semantics is reduced to a syntactic concept according to the following statements:

- if a vertex u is a predecessor of a vertex v, then u influences v ;
- if a vertex u is not a predecessor of a vertex v, then u does not influence v.

These two statements which have been kept implicit so far, are fundamental to understand the slice definition. When we try to extract parts of a DFG which are close to the cryptographic parameters, we are actually interested in parts of this DFG which are related with the cryptographic parameters according to the symmetric transitive closure of the influence relation. According to these two statements the predecessor relation is equivalent to the influence relation. As a consequence it is possible to adopt a purely syntactical formulation for the slice definition which, as we said, has the advantage of homogeneously integrating the readability requirements.

Unfortunately none of these two statements is true. To repair the first statement, load-value influence needs to be taken into account in DFG syntax. This is done through a DFG transformation called concrete memory access simplification described in the second paragraph of Section 5.3.1. And to improve on the veracity of the first statement, we introduce in Section 5.3.2 a minor adjustment to the slice definition to filter paths that do not reflect actual dependencies (fined-grained influence tracking).

We also present, in Section 5.3.1, a particular type of influence relation, called address influence, which is not to be included in slices even though it complies with the two previous statements.

5.3.1 Memory Operations

Address Influence

Assuming realistic memory states (not all memory locations are equal to the same value) a memory access is influenced by its address operand. Yet we consider that a pointer and the variable pointed to are independent. There are two reasons to explain this choice.

- We suppose that modes of operation are limited to simple computations. Otherwise the most complex parts must be identified separately using for instance a primitive identification method. In particular, it seems reasonable to assume that parts of DFGs which are relevant to identify modes of operation are unlikely to contain any indirection. Thus, there is no absolute need to take address influence into account.
- If we consider that there is a dependency between a memory operation and its address operand, it will create many connections which do no reflect anything but implementation noise. For instance, given a function with the following prototype:

  ```
  void function(char* plaintext, char* ciphertext);
  ```

  If, according to the calling convention, arguments are pushed onto the stack, the plaintext and the ciphertext will automatically be connected through the stack pointer. Yet they may be perfectly independent and the fact that they are two arguments of the same function is not worth to figure in a slice. As a less radical solution one may suggest to limit the number of indirections along a path. For this idea to be effective, the maximum number of indirections must be strictly lower than two. In fact, if two cryptographic parameters are stored in the same structure or if the same offset is used to access two cryptographic parameters (as it might be the case for the plaintext and the ciphertext in a stream cipher for instance), there will be undirected paths between these parameters with only two indirections but which are nevertheless irrelevant to mode of operation identification. We did not implement this solution though.

To conclude a path cannot contain indirections, that is to say, edges labelled as an address operand and the destination vertex of which is labelled with a memory operation.

Load-Value Influence

The result returned by a `load` operation is either an input value, if it is the first time in the sequence of memory operations that its address is accessed, or is equal to the last value that was read or written at its address. In the latter case, there is no edge to explicitly reflect these equalities in our DFG model. Yet we absolutely need to consider them while searching for relations between cryptographic parameters, otherwise we will lose track of cryptographic parameters as soon as they leave the general purpose registers.

A solution to make load-value influence explicit is to use memory access simplification. This normalization mechanism (refer to Section 3.6) replaces `load` operations, the address of which was previously accessed in the sequence of memory operations, by the value that was read or written then. The sore point of memory access simplification is accurate pointer comparison. Two approaches have been presented: static and dynamic pointer comparison. Static pointer comparison is necessary and also sufficient for primitive identification. But modes of operation differ from primitives.

- As mentioned earlier, modes of operation are only supposed to contain simple computation. In particular the content of a cryptographic parameter is not supposed to be involved in address expressions. Thus, if it varies from one execution to another, it will have no effect on the relative ordering of addresses. The address of cryptographic parameters may also vary between two executions, but then, since they are probably obtained from a dynamic heap or stack allocator, if they do not overlap for one execution we can assume that they never will. Under these conditions, the loss of generality induced by dynamic pointer comparison seems acceptable.

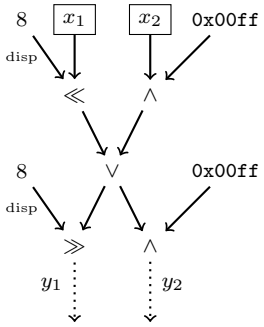

Figure 5.6: Example of a DFG where the influence relation is not preserved along certain paths. The size attribute of x_1 and x_2 is equal to 16 bits.

- Modes of operation usually manipulate several memory buffers (at least one for the plaintext, one for the ciphertext, one for the key and one for the nonce) with mixed `load` and `store` operations. Pointers to these buffers are generally defined outside the trace segment. Thus, it will be impossible to produce accurate static comparisons for these pointers. Because we have to deal with possible aliasing conservatively, if we cannot compare pointers accurately we will miss many simplifications, that is to say, many load-value influence relations.

For these two reasons we use dynamic pointer comparison to simplify memory accesses of modes of operation. We refer to this transformation as concrete memory access simplification.

5.3.2 Fine-Grained Influence Tracking

When we reduce the influence relation to a syntactic concept, we lose the ability to perform fined grained influence tracking. In fact, we are forced to admit that, for any operation, every bit of its input operand(s) influences every bit of its result. Hence, the granularity of the influence tracking depends on the size of vertices. This approximation can lead us to falsely conclude that a vertex influences another one based on the fact that there is a directed path between them. For instance in the example of Figure 5.6, there is a directed path from x_1 to y_2 but x_1 does not influence y_2. As a second example, y_1 and y_2 have a common predecessor but no bit of this common predecessor simultaneously influences y_1 and y_2. Including such connections in a slice will reduce its readability and could also be misleading by letting the analyst think that there is an influence relation between a pair of cryptographic parameters while in reality there is none.

To solve this issue we perform fined grained influence tracking using influence masks. Given two vertices u and v, influence masks were used in Section 3.5 to determine which bits of u influence v. Here our objective is slightly different: we only want to consider the influence along a single path, and since we are interested in undirected paths, we also want to consider indirect influence. Let p be an undirected path characterised by a sequence of edges $(e_i)_{1\leq i\leq n}$ and a sequence of vertices $(u_i)_{0\leq i\leq n}$. The indirect influence mask on u_0 along p, noted $M^{\leftrightarrow}_{u_0|p}$, is a mapping from $\{u_i, 0 \leq i \leq n\}$ to $\{0,1\}^*$ recursively defined as follows:

$$M^{\leftrightarrow}_{u_0|p}(u_i) = \begin{cases} 11...1 & \text{if } i = 0 \\ I_{-e_i}(M^{\leftrightarrow}_{u_0|p}(u_{i-1})) & \text{if } u_i = src(e_i) \\ I_{+e_i}(M^{\leftrightarrow}_{u_0|p}(u_{i-1})) & \text{if } u_i = dst(e_i) \end{cases}$$

We associate to each edge e a function I_{-e} and a function I_{+e}. The function I_{-e} has already been presented in Section 3.5. It propagates an influence mask from the destination vertex of e to the source vertex of e. The function I_{+e} performs the opposite transformation, that is to say, it propagates an influence mask from the source vertex of e to the destination vertex of e. For instance, in the example of Figure 5.6, if e_1 denotes the edge between x_1 and $\ll$, then:

$$
\begin{array}{lccc}
I_{-e_1}: & \{0,1\}^{16} & \rightarrow & \{0,1\}^{16} \\
 & m & \mapsto & m \gg 8
\end{array}
$$

$$
\begin{array}{lccc}
I_{+e_1} & \{0,1\}^{16} & \rightarrow & \{0,1\}^{16} \\
 & m & \mapsto & m \ll 8
\end{array}
$$

We consider that p truly reflects an indirect influence relation between u_0 and u_n, if $M^{\leftrightarrow}_{u_0|p}(u_n) \neq 00...0$. With that in mind, we modify the notion of distance in the original slice definition: the distance between u and v is now equal to the length of the shortest path p between u and v such that $M^{\leftrightarrow}_{u|p}(v)$ contains at least one 1 bit.

5.4 Slice Construction

At the beginning of this section we briefly mention the light normalization process which takes place prior to slice extraction. Then we present the slice extraction algorithm. This presentation is divided in two parts: first we give a naive algorithm which returns optimal solutions but which is not tractable in practice, then we describe an approximation algorithm.

5.4.1 Light Normalization

The light normalization process is made of the following normalization mechanisms: concrete memory simplification (Section 5.3.1), common subexpression elimination (Section 3.3) and a subset of the rewrite rules presented in Section 3.11. Apart from concrete memory simplification which is specific to mode of operation identification, the other normalization mechanisms are also involved in the standard normalization process. Thus, they only affect DFGs which are not entirely normalized. A DFG is not entirely normalized if, for instance, it was built using pre-existing DFGs already annotated with primitive information. In that case, there is no need to search for primitives in the entire DFG and hence only the pre-existing parts are likely to be normalized. There are two reasons to apply common subexpression elimination and other miscellaneous rewrite rules to DFGs before extracting slices.

- To simplify DFGs. It is clear that the simpler DFGs are, the more readable slices will be.
- To replace constant expressions by constant terms. A constant expression does not depend on its input operand(s). It would be inappropriate to include in a slice a connection between a constant term and one of its operand(s). Thus it is important to detect constant expressions and to replace them by constant terms. A typical example of a constant term which is handled by the light normalization process is: $\oplus(x, x)$.

We did not include in light normalization all the normalization mechanisms that were devised for primitive identification. Most of them are not directly related to constant expression detection nor produce tangible simplifications for expressions which are usually reported in slices. For instance many normalization mechanism were introduced to simplify pointer expressions and to facilitate static pointer comparison. These normalization mechanisms are of no direct interest in our context. There is a second reason which is purely practical. Since light normalization is executed on augmented DFGs, we must pay special attention to vertices which break the DFG syntax. As a consequence implementation of normalization mechanisms needs to be slightly modified to run on augmented DFGs.

5.4.2 Naive Algorithm

Finding a minimum distance-preserving subgraph is a difficult task. A basic idea is to search for a shortest path for every pair of PAR and to take the union of these shortest paths. Since the length of a path is equal to its number of edges, a BFS algorithm can be used to compute the shortest path between two vertices. For a sparse graph with a number of edges linear in the number of vertices (as it is the case in our DFG model) the complexity of the BFS algorithm is linear in the number of vertices. Thus, the overall complexity of this simplistic algorithm is $\mathcal{O}(|V|.|PAR|^2)$.

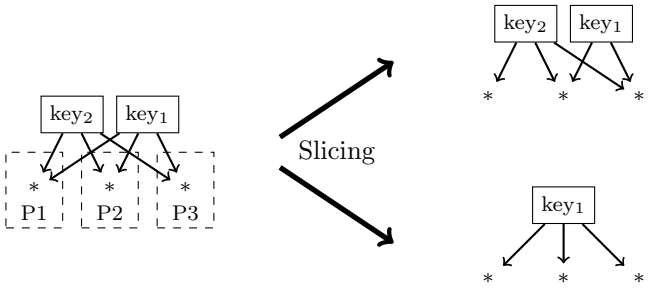

Figure 5.7: A data flow with two possible distance-preserving subgraphs

However, the resulting subgraph is not necessarily the smallest. If there are several shortest paths for a pair of vertices, the size of the union may depend on which one is chosen. This is illustrated by an example in Figure 5.7. We want to find a slice for the DFG on the left assuming a set of parameters $PAR = \{\mathrm{P1}, \mathrm{P2}, \mathrm{P3}\}$. By using the algorithm we just described, we may obtain the slice given on the top right which is equal to the union of $(\mathrm{P1}, \mathrm{key}_2, \mathrm{P2})$, $(\mathrm{P1}, \mathrm{key}_1, \mathrm{P3})$ and $(\mathrm{P2}, \mathrm{key}_2, \mathrm{P3})$. However, the slice given on the bottom right is smaller. This problem is common in practice. In fact, a cryptographic parameter is almost always defined by a set of vertices. For instance on a 32-bit architecture, a 128-bit plaintext is usually split into four 32-bit fragments. At least one shortest path for each of these fragments is to be expected. Back to the example of Figure 5.7, key_1 and key_2 could be two fragments of the same key parameter.

In the field of graph spanners, Coppersmith *et al.* [18] describe an approximate algorithm to compute pair-wise preservers. Given a graph $G = (V_G, E_G)$ and P a set of vertices, a pair-wise preserver of G with respect to P is a subgraph $G' = (V_G, E')$ that is distance-preserving for the elements of P. Their algorithm produces a pair-wise preserver the number of edge of which is bounded by $\mathcal{O}(|V_G| + \sqrt{|V_G|}.|P^2|)$. The idea behind their algorithm is to modify slightly the weight of the edges to enforce the uniqueness of the shortest paths. If this upper-bound is relevant from the graph spanner perspective, in our case it does not provide any guarantee at all. The DFG is already sparse. Thus, all its subgraphs are under that bound. Apart from this work, we have not been able to find any work or study addressing directly our problem.

An exact solution can be computed using the following algorithm. First, search for the set of shortest paths for every pair of parameters. Then, pick one path from each set, such that their union is minimum. This algorithms suffers from a very high complexity. The number of shortest paths can be exponential in the number of vertices. Evaluating every possible selection of paths to find the smallest union also has an exponential complexity.

5.4.3 Greedy Algorithm

To reduce the complexity of this former algorithm and to make it tractable in practice, we introduce the following adjustments. First, we limit to a fixed amount the number of paths returned by the BFS shortest path computation. And second, we use a greedy algorithm to find the set of paths with the smallest union. Iteratively, for each pair of parameters, we insert the shortest path which shares the largest number of edges with the current selection. A pseudo-code for this new algorithm is given in Algorithm 9.

The complexity of this greedy algorithm is $\mathcal{O}(|V_G|.|PAR^2|)$. Although there is no theoretical guarantee that the returned subgraph would be the smallest, it is often the case in practice. A list of remarks is given as follow to justify this observation. First, the fixed upper-bound on the number of shortest paths is almost never reached. In fact, as previously said, when several shortest paths are found it is often due to parameter fragmentation. Because fragments are rarely mixed together outside of the cryptographic primitives, the number of shortest paths is almost always linear in the number of fragments. Second, not every pair of parameters has several shortest paths. Thus, the greedy selection mechanism starts with a non empty set of edges. As a consequence, the first path is not chosen randomly. Finally, some sets of shortest paths are disjoints. For instance, in an usual mode of operation, the plaintext paths do not intersect the key paths. We define a

Algorithm 9 Greedy Algorithm

for all pairs (P_1, P_2) of PAR **do**
  $Path_{P_1,P_2} = shortestPath(P_1, P_2)$
end for
initialise Γ as an empty graph
repeat
  pick an unprocessed pair (P_1, P_2) such that $|Path_{P_1,P_2}|$ is minimal
  pick a path $p \in Path_{P_1,P_2}$ such that $|E_\Gamma \cup p|$ is minimal
  add p to Γ and mark (P_1, P_2) as processed
until all pairs of PAR have been processed
return Γ

binary relation $\sim$ on PAR^2, such that given $Q_1, Q_2 \in PAR^2$, $Q_1 \sim Q_2$ if the shortest paths of Q_1 intersect the shortest paths of Q_2. The transitive closure of $\sim$ is an equivalence relation on PAR^2 and its equivalence classes form a partition of PAR^2. Each equivalence class of PAR^2 can be processed separately. This mitigates the influence of the selection algorithm on the solution. If a suboptimal solution is returned for an equivalence class it will not deteriorate the results returned for the other equivalence classes. Moreover, we insert an extra clustering phase after computing the sets of shortest paths and prior to the selection loop. During this phase we compute the equivalence classes of P^2. For each class, either the exhaustive search or the greedy algorithm is used depending on its cardinality and on the cardinality of its elements.

5.5 Experimental Evaluation

In this section we describe the experiments we conducted to evaluate the mode of operation identification method. In a first time, we give a batch of results obtained on a set of synthetic samples covering very basic modes of operation. As in Chapter 4, the idea is to extensively test different compilers, optimization levels and well known cryptographic libraries. In a second time, we present in detail two slices that were obtained for more challenging modes of operation.

Graphical Representation of Slices. Slices are depicted using the following convention. Let Γ be a slice and e be an edge in E_Γ. If $src_\Gamma(e)$ is labelled with an operation symbol and has a single outgoing edge and if $dst_\Gamma(e)$ is also labelled with an operation symbol and has a single ingoing edge, then the endpoints of e are condensed into a single vertex. This vertex is labelled with a sequence of operation symbols: first the label of $src_\Gamma(e)$, then the label of $dst_\Gamma(e)$. Accordingly, a directed path with no intersection is compacted into a single vertex, in the same way a basic block is depicted by a single vertex in traditional CFG representations. Even though slices do not contain any vertex labelled with a signature symbol, we use some of them to mark the emplacement of the cryptographic primitives. If a cryptographic parameter is not connected to anything apart from the vertex marking the emplacement of its primitive, it will not be depicted. The vast majority of edge labels are not depicted. The only edges with a visible label connect a cryptographic parameter to the vertex marking the emplacement of its primitive. These labels specify the identify of the parameters: plaintext, ciphertext or key for a block cipher for instance.

5.5.1 Extensive Evaluation on Basic Modes of Operation

The set of synthetic samples covers three modes of operation: CBC (encryption and decryption), CTR and HMAC. It contains implementations taken from five well known cryptographic libraries: Botan, Crypto++, Nettle, OpenSSL and TomCrypt. CBC and CTR were tested with AES and HMAC was tested with MD5. For CBC and CTR we also recompiled the implementation of Gladman using different compilers and optimization levels (arbitrarily we used configuration V_1 but this choice has no impact on the implementation of the different modes of operation).

Methodology

For each synthetic sample we collect an execution trace including runtime address values. Depending on the difficulty to identify primitive executions in large trace segments, we either extract a single trace segment (corresponding to the complete execution of the mode of operation) or one trace segment per primitive execution. For instance due to constant folding the first execution of the MD5 compression function cannot be identified if the trace segment also contains the initialisation of the chaining value. Thus, for all HMAC MD5 synthetic samples, we identified the different executions of the MD5 compression function separately. Once primitive executions are identified, we perform the transformation described in Section 5.1.2. If primitive executions were identified separately, we extract a trace segment containing the execution of the whole mode of operation and we construct its DFG using DFG composition as explained in Section 2.3.3. Then we perform light normalization and finally we compute the slice.

To save some space, we do not detail all the slices that we obtained. Instead, to assess their usability by a human analyst, we provide measurements of their completeness and of their readability. We define these two notions with respect to an ideal slice. Let Γ be a slice, Γ_{opt} be the ideal slice for the targeted mode of operation, and *Msc* be a function which returns, for a pair of graphs, their maximum common subgraph. The completeness *Cp* and the readability *Rd* are defined as follows:

$$Cp(\Gamma) = \frac{|Mcs(\Gamma, \Gamma_{opt})|}{|\Gamma_{opt}|} \qquad Rd(\Gamma) = \frac{|Mcs(\Gamma, \Gamma_{opt})|}{|\Gamma|}$$

Here, the size of a graph (denoted by $|.|$) is equal to its number of edges. If their is an injective morphism from Γ_{opt} to Γ then the completeness is equal to one, otherwise it is less than one. If their is an injective morphism from Γ to Γ_{opt} then readability is equal to one, otherwise it is less than one. Ideally Γ and Γ_{opt} are isomorphic and both the completeness and the readability are equal to one. During our experiments, the completeness and the readability were evaluated manually.

In Figure 5.8, we give what we consider to be the ideal slice for CBC encryption, CBC decryption, CTR and HMAC. The * label refers to any path which does not intersect the rest of the graph. In CBC encryption and in CTR the primitive noted *enc* is a block cipher encryption function. It has two input parameters (noted *pt* for plaintext and *key*) and one output parameter (noted *ct* for ciphertext). In CBC decryption the primitive noted *dec* is a block cipher decryption function. In HMAC the primitive noted *comp* is a compression function. It has two input parameters (noted *state* and *msg* for message block) and one output parameter (noted *hash*). These ideal slices only contain the minimal number of executions of the primitives allowing identification. The dotted edges mark the emplacement where the slices must be extended if larger trace segments were to be analysed.

Results

Completeness and readability measurements are given in Table 5.2. The completeness is always equal to one. It means that the slicing process did not miss any important connection specified in the ideal slices. The majority of the readability values are also equal to one, except for HMAC MD5. We detail below the synthetic samples with a readability value lower than one.

HMAC MD5. For HMAC MD5 synthetic samples, slices contain an additional connection between the message parameter of the last compression function of the inner hash function and the message parameter of the last compression function of the outer hash function. It appears that this additional connection is caused by the size of a message block. In fact, according to the specifications the size of $k \oplus \texttt{opad}$ and $k \oplus \texttt{ipad}$ are both equal to the size of a message block. Thus, the size of $k \oplus \texttt{opad} \,||\, H(k \oplus \texttt{ipad} \,||\, m)$ and $k \oplus \texttt{ipad} \,||\, m$ depend on the size of a message block. And since the message padding includes the length of the message, it is perfectly legitimate for the content of the last block to depend on the size of a message block. Note that for trace segments limited to the HMAC execution, it is generally impossible to determine that the size of a message block is a constant. In fact, HMAC implementations are generally hash function-agnostic. Thus the size

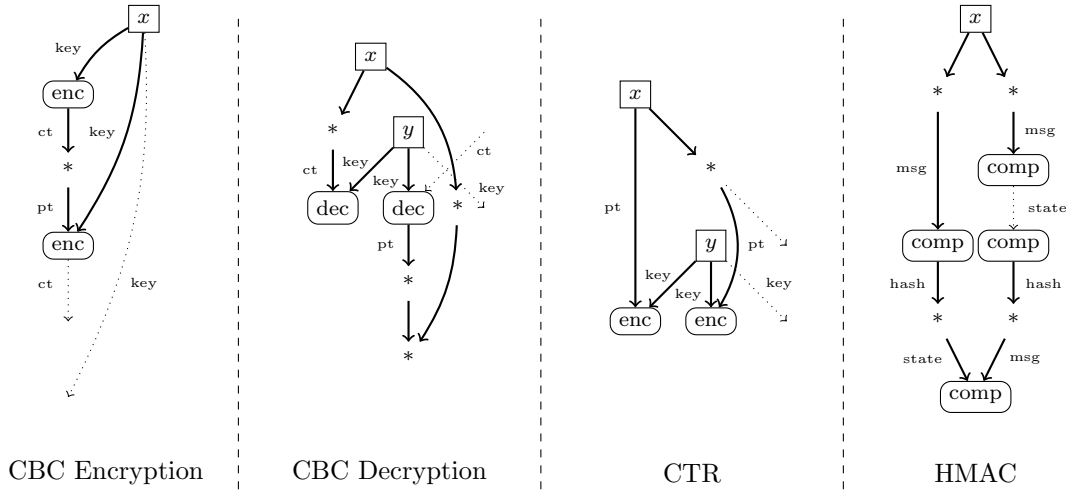

Figure 5.8: Ideal slice for CBC encryption, CBC decryption, CTR and HMAC.

of a message block is an input parameter for HMAC implementations. We count this additional connection as superfluous since it can be misleading for inexperienced human analysts. It decreases the readability score from 1 to 0.8. The readability score obtained for the HMAC MD5 synthetic samples based on Botan and Nettle are even lower. In the case of Botan, after computing an authentication tag the internal HMAC data structure is reinitialised to be ready to compute a new authentication tag. As part of this reinitialisation process the compression function is executed on $k \oplus \texttt{ipad}$. Thus the slice contains an additional execution of the compression function which is connected to the key parameter. In the case of Nettle, the key parameter is manually realigned. In our synthetic samples the message and the key parameter are declared as C strings. As such they were not aligned by the compiler and they overlap over an architecture word. In this context, the manual memory realignment computation creates a connection between the key and the message that seriously spoils the readability of the slice.

CBC AES. The readability score is lower than one for some specific compilation conditions of Gladman V_1 CBC AES. At the beginning of the AES primitive, Clang `-O2` and `-O3` produces a rather complex sequence of SIMD instructions to read the plaintext memory buffer. It includes a 128-bit mask which is stored in the `.rodata` segment. This mask is a common predecessor for every plaintext parameter. This issue does no affect Gladman V_1 CTR AES since there is a shorter path between the plaintext parameters. The easiest way to solve this issue would be to establish that the 128-bit mask is a constant value. As future work one can extend the constant expression detection mechanism to replace by a constant term any `load` operation the address of which is a constant and which accesses read only memory segments.

5.5.2 Examples on More Complex Modes of Operation

The objective of this section is to present real slices obtained for more challenging modes of operation. In the first example we apply our mode of operation identification method to an OCB AES implementation. OCB is an authenticated encryption mode of operation. With this example we show that our solution can scale to complex modes of operation. In the second example we apply our solution to a malicious CBC implementation containing a back door. With this example we demonstrate that our solution can provide valuable information even for modes of operation that would have been difficult to anticipate.

Authenticated Encryption: OCB

There are three versions of OCB. This example is based on the implementation of OCB given in the TomCrypt library which corresponds to the first version, described in [64]. The slice given in

Table 5.2: Mode of operation identification for CBC, CTR and HMAC synthetic samples. For cells highlighted in yellow, primitives were identified in small trace segments limited to a single primitive execution, whereas for the other cells, primitives were identified directly in the trace segment containing the whole mode of operation execution.

			CBC AES Enc.	CBC AES Dec.	CTR AES	HMAC MD5
Gladman V_1	GCC	00	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	
		01	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	
		02	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	
		03	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	
	Clang	00	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	
		01	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	
		02	$Cp = 1, Rd = 0.4$	$Cp = 1, Rd = 0.4$	$Cp = 1, Rd = 1$	
		03	$Cp = 1, Rd = 0.4$	$Cp = 1, Rd = 0.4$	$Cp = 1, Rd = 1$	
	MSVC	00	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	
		02	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	
Botan			Primitive not detected			$Cp = 1, Rd = 0.7$
Crypto++			Primitive not detected			$Cp = 1, Rd = 0.8$
Nettle			$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 0.6$
OpenSSL[a]			$Cp = 1, Rd = 1$	Prim. not detected	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 0.8$
TomCrypt			$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 1$	$Cp = 1, Rd = 0.8$

[a]Compared to the experiments conducted in Chapter 4, here we used a specific signature to detect the AES implementation of OpenSSL. Due to the *MixColumns* operation, different signatures are required for the encryption and for the decryption. Unfortunately we did not have time to create the decryption signature.

Figure 5.9, was obtained after encrypting a 34-byte message with OCB AES. We used the set of signatures described at the end of Chapter 4 to detect the AES primitive.

To justify why this slice correctly reflects the algorithm and to underline some of its imprecisions we divide the slice into four parts.

- The first part is coloured in blue and is located at the top of the figure. It computes the first offset which is equal to: $E_k(N \oplus E_k(00...0))$, where N denotes a nonce and E_k an AES encryption under a key k. The two AES executions and the bitwise XOR in between are visible in the slice.

- The second part is coloured in orange and is located at the bottom left of the figure. It encrypts the two first message blocks by evaluating the expression: $E_k(M[i] \oplus Z[i]) \oplus Z[i]$, where $M[i]$ denotes the i^{th} message block and $Z[i]$ the i^{th} offset (random mask). Here again the slice perfectly transcribes the algorithm specifications. The two message blocks correspond to the two `load` vertices at the centre of the slice. The offset $Z[i]$ is XORed two times, before and after the encryption. The bitwise OR and `part18` operations are due to size changes from 32-bit to 8-bit variables and conversely.

- The third part is coloured in violet and is located on the right of the figure. It corresponds to the last block encryption defined by: $E_k(len(M) \oplus L(-1) \oplus Z[m]) \oplus M[m]$. The last message block $M[m]$ does not appear in the slice. $M[m]$ is read only once for the whole scheme. Thus, it does not belong to any path, and consequently it was not reported in the slice.

- The last part is coloured in green and is located at the bottom right of the figure. It computes the authentication tag defined by: $E_k(M[1] \oplus ... \oplus M[m-1] \oplus (C[m] \,||\, 0^*) \oplus Y[m] \oplus Z[m])$, where $C[m] \,||\, 0^*$ is the last encrypted block padded with zeros and $Y[m] = E_k(len(M) \oplus L(-1) \oplus Z[m])$. As previously said, the message used for this slice is 34-bytes long. Thus, the size of $C[m]$ is 2 bytes. These two bytes are obviously not on any shortest path, since they involved an additional XOR operation compared to $Y[m]$. With this remark in mind, the slice appears to contain the right dependencies: the two message blocks, $Y[m]$ and $Z[m]$ are XORed together and the result is encrypted.

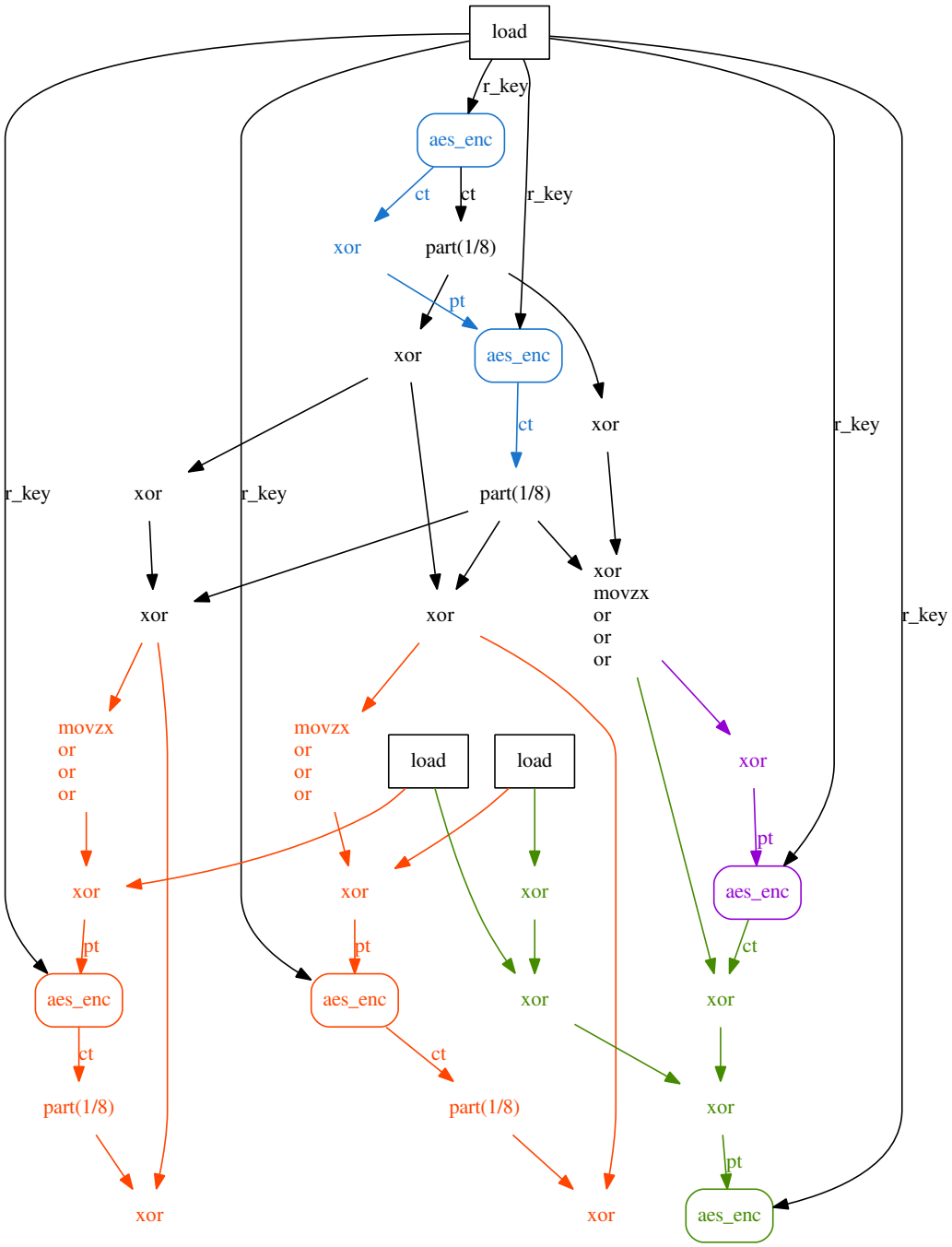

Figure 5.9: Experimental slice obtained for the OCB AES implementation of the TomCrypt library executed on a 3-block message. Colours were added manually to point out the different parts of the algorithm:

- — : first offset computation ;
- — : encryption of the first two blocks ;
- — : encryption of the last block ;
- — : authentication tag computation.

This slice was generated in 0.5 s on a Pentium Dual-Core T4200 processor out of a DFG with 14187 vertices and 19630 edges.

For brevity, we will not dig into how the different offsets are generated. As far as we conducted our inspection, no inconsistency between the slice and the specifications was discovered. To conclude, the slice contains most of the interesting connections even though some are missing (the bitwise XOR with $C[m] \mid\mid 0^*$ for instance). Obviously the complexity of this mode of operation reduces the advantage of a graph representation for a human analyst. However, as demonstrated above, it is still possible to understand it with the help of the specifications.

IV-Replacement Attack

An Algorithm Substitution Attack (ASA) consists in replacing the original encryption algorithm by a malicious one containing backdoor capabilities. There has been a renewed attention in the past few years for ASA, as shown by recent publications in that domain [12, 11]. Closed source implementations of symmetric cryptography are attractive targets for ASA. Thus, while evaluating binary software, security experts could be interested in detecting ASA. This example shows that our mode of operation identification method can automatically discloses an IV-replacement attack.

An IV-substitution attack is a simple ASA that was first described in [12]. It can be used against any encryption scheme that surfaces its IV, such as CBC or CTR. Two keys are used: the legitimate encryption key k defined by the user and a second key k' known only by the attacker. The IV is replaced by k encrypted under k'. Anyone with the knowledge of k' can decrypt the IV, recover k and finally decrypt the data.

For this experiment, we implemented a very simple CBC AES encryption subject to an IV-replacement attack. The encryption key k is also encrypted using AES. To start the analysis, we located the AES key schedule and the AES encryption, using our primitive identification method. The slice that was returned by our mode of operation identification method is given in Figure 5.10. It is easy to recognize three CBC patterns in the middle: encryption executions chained by bitwise XORs. Notice that encryptions depend on both the result of the key schedule and the key (vertices labelled with `load`). It is perfectly correct since the first four round keys are equal to the key. The key schedule is executed two times: once for k' and once for k. The IV generation happens on the top left corner: we notice that the first AES encryption takes as a plaintext parameter a value read from the memory which is later used as an input by a key schedule execution. This is the encryption key k. The IV-substitution pattern is thus clearly visible.

5.5.3 Conclusion

We demonstrate through experimental results on CBC, CTR and HMAC that, in practice, slices produced by our method are complete and also readable apart from a few corner cases which were clearly identified. We hope that the two slices given at the end of this section have convinced the reader that security analysts can take advantage of the results returned by our solution to quickly identify modes of operation and to get a good understanding of their internal structure. We believe that our solution is highly profitable for black box audits and any other activities which require to reverse engineer binary implementations of modes of operation.

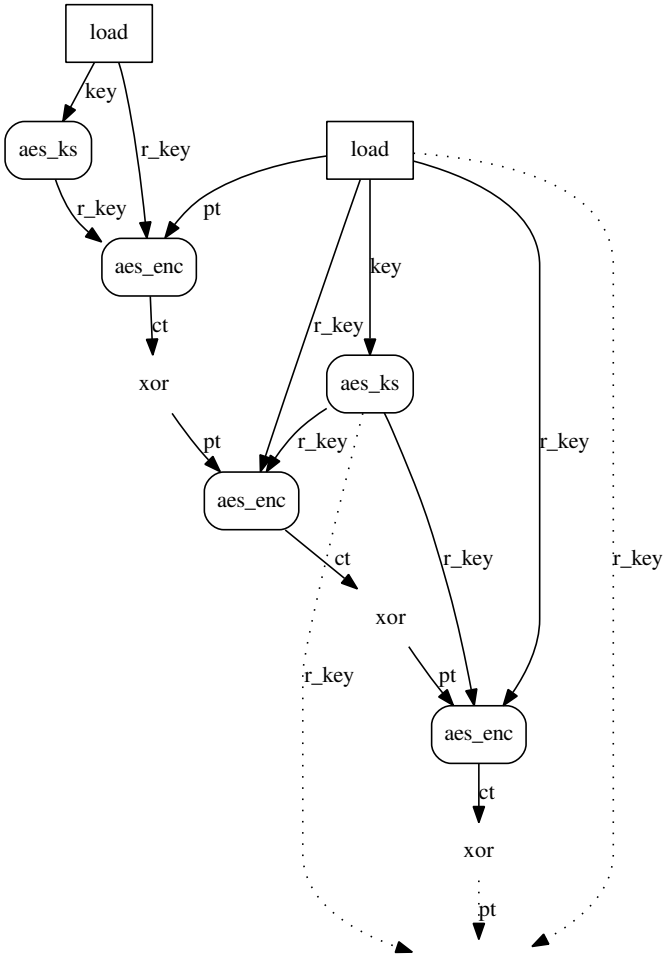

Figure 5.10: Experimental slice obtained for an CBC AES encryption subject to an IV-replacement attack. This slice was generated in 4 ms on a Pentium Dual-Core T4200 processor out of a DFG with 675 vertices and 968 edges.

Chapter 6

Detailed Use Cases

In this chapter we present two practical examples of automated reverse engineering of cryptographic algorithms. In the first example we explain how one can make use of our primitive identification method to run test vectors in an automated way. In the second example, we detail a complete analysis of mode of operation in an instant messaging application. With these two examples we confront our solution with real life programs.

Contents

6.1 Automatic Test Vectors Verification 124
6.1.1 Execution Trace 125
6.1.2 Fragment Selection 125
6.1.3 Primitive Identification 125
6.1.4 From Vertices to Instruction Operands 126
6.1.5 Test Vectors 127
6.1.6 Conclusion 128
6.2 Complete Analysis of an Instant Messaging Application 128
6.2.1 Execution Trace 128
6.2.2 Basic Heuristics for Segment Selection 128
6.2.3 Primitive Identification 129
6.2.4 Block Cipher Mode of Operation 129
6.2.5 Key and IVs Derivation Function 131
6.2.6 Conclusion 131

6.1 Automatic Test Vectors Verification

The most common way to check that a given cryptographic implementation truly behaves according to its specifications is to run test vectors. A test vector is a couple of values (x, y) such that any implementation f of some given specifications must verify $f(x) = y$. As a first step to check whether or not this relation holds in practice, security experts have to locate the algorithm along with its input and output parameters. With this knowledge, they should be able to selectively execute the algorithm on any chosen inputs and to monitor the output. In this section, we propose to use our primitive identification method to automatically retrieve the location of the input and output parameters.

In this example we analyse 7-Zip [59]. It is a well known and widely used file archiver and compression tool for Windows. In our context, we got interested in 7-Zip because it allows users to encrypt compressed data. In the remainder of this section we describe step by step how, with only minimal human interactions, we can identify the encryption primitive used by 7-Zip and check the correctness of its implementation.

The materials presented in this section have been published in a blog post [46].

6.1.1 Execution Trace

The binary that was used in this example is 7-Zip version 16.04 compiled for a 32-bit architecture. As a first step we need to collect an execution trace which contains at least one execution of the encryption primitive. It was done using the following command line:

```
./pin.exe -t lightTracer.dll -w whl.lst -- ./7z a -p123passwd file.7z file.txt
```

We used a pintool of our own making, called `lightTracer.dll`, that has briefly been introduced in Section 2.2.2. The `-w` option specifies the name of the file, here `whl.lst`, which contains a list of white-listed shared libraries. Shared libraries white-listing is a simple but efficient technique to reduce the tracing overhead. Only the shared libraries mentioned in `whl.lst` are instrumented by `lightTracer.dll`. For this example the only white-listed shared library was `7z.dll`. The other libraries imported by `7z.exe` are related to the operating system and thus, can rapidly be excluded from our analysis. 7-Zip is invoked with the `a` and `-p` options to create an encrypted archive. For simplicity, we add to the archive a single file called `file.txt` which contains 12 bytes of data.

The tracing process took a few seconds and returned an execution trace of 185 MB. A total of four threads were executed. The first thread executed 376 million dynamic instructions, the second thread 770 million dynamic instructions and the last two threads around ten thousand dynamic instructions each.

6.1.2 Fragment Selection

We know from the information available on the 7-Zip website, that the encryption primitive should be AES. Our initial idea was to search for constants because many implementations of AES use large lookup tables. FindCrypt2 did not find any constant related to AES, but a manual search revealed an AES substitution box at virtual address `0x100d5e28` in `7z.dll`. The reason why FindCrypt2 failed is that it only looks for the large lookup tables, which are neither contained within `7z.exe` nor within `7z.dll`. Using IDA [38], we found that the AES substitution box is used by four different functions.

- The first function F_1 starts at virtual address `0x100bfba0`. It was executed once by the second thread after 770755330 dynamic instructions.
- The second function F_2 starts at virtual address `0x100bfcd0`. It was not executed.
- The third function F_3 starts at virtual address `0x100bfd60`. It was executed once by the second thread after 770757133 dynamic instructions.
- The fourth function F_4 starts at virtual address `0x100c08d0`. It was executed once by the first thread after 202452 dynamic instructions.

Functions F_1, F_3 and F_4 are possible implementations of the AES encryption algorithm. From our experience we know that the length of a typical AES encryption algorithm does not exceed 1000 dynamic instructions. Thus we selected the three trace segments of 1000 dynamic instructions each, starting from the entry point of F_1, F_3 and F_4 respectively. These trace segments are given below:

- `[770755330:770756330]` for the second thread ;
- `[770757133:770758133]` for the second thread ;
- `[202452:203452]` for the first thread.

6.1.3 Primitive Identification

To detect AES we used the set of signatures that is described in Section 4.3.2 plus two additional signatures called AL_V9 and AF256. AL_V9 is the 9th version of the last round signature. It targets implementations that use the AES substitution box and $\vee(x_1, \ll (\vee(x_2, \ll (\vee(x_3, \ll (x_4, 8)), 8)), 8))$ as the merge expression. AF256 stands for **A**ES **F**ull **256**. It is a wrapping of A14 which includes the first *AddRoundKey*. To break the symmetries of the first *AddRoundKey*, that is to say to dissociate the plaintext from the first round keys, the signature also includes the `load`

Table 6.1: Signatures detected for the second trace segment `[770757133:770758133]`. Execution time measurements were performed on a Pentium Dual-Core T4200.

Signature	Number of Occurrence	Time (s)
AL_V9	4	0.003
AT4	312	0.096
A10	100	0.153
A12	52	0.030
A14	4	0.001
AF256	4	0.032

operations that access the round keys and their address expressions. This workaround is discussed in Section 5.1.3.

We did not find any signature in the first and in the third trace segment. But we detected an AF256 signature in the second trace segment. The exact signatures, their number of match and the time spent for their detection are given in Table 6.1. Note that AT4 has 24 automorphisms and AL_V9 has four automorphisms. It explains why we detected four matches of the AF256 signature when the trace segment contains at most only one complete execution of the encryption primitive. Interestingly, we observe that even if the four large lookup tables are not contained within the binaries, 7-Zip still uses a table implementation of AES. We can guess that the large lookup tables are dynamically computed.

A manual analysis later revealed that F_1 implements the key scheduling for the encryption, F_2 the key scheduling for the decryption and F_4 the large lookup table computation.

6.1.4 From Vertices to Instruction Operands

For each signature match, he subgraph isomorphism algorithm returns a mapping from parameter fragments to vertices. To be usable by security experts, this result needs to be converted back to the assembly language representation. To do so, during DFG construction, vertices are associated with the index of the dynamic instruction they are originated from. We assume that a vertex corresponds, in the assembly language representation, to the destination operand of the dynamic instruction it is associated with. This naive scheme has several flaws though.

First, an instruction may have several explicit and implicit destination operands. For instance, the instruction `div ebx` modifies the value of both `eax` (quotient) and `edx` (remainder). If a vertex is associated with this instruction, how can we tell to which register it corresponds? Second, not all the vertices that are used to represent an instruction correspond to their destination operand(s). For instance in the DFG that represents `xor eax, [esp + 0x10]`, there is one vertex labelled with a variable symbol to represent the initial value of `eax` and one vertex labelled with a + to represent the address computation. But none of them corresponds to the final value of `eax`. Third, during the normalization phase, vertices that do not correspond to any particular dynamic instruction may be inserted. There is no easy way to convert those vertices back to the assembly language representation. In the context of test vectors, we should either characterize those vertices by their direct predecessors or by their direct successors depending on whether we need to write or to read their assembly counterparts. We proceed recursively until vertices that are associated with dynamic instructions are eventually reached.

The first two points are not addressed by this work and the last point only received a partial solution.

Fortunately, none of these complications happened in our 7-Zip example. The following lines are the exact result that was returned by our analysis tool about the location of the AF256 parameters.

```
Parameter I0 = {IMEM@15 IMEM@10 IMEM@11 IMEM@9}
Parameter I1 = EAX[16:256]@1
Parameter O0 = {@868 @891 @911 @845}
```

There is one line per parameter. The first line is about the plaintext parameter. This parameter is divided into four 32-bits fragments. The first fragment corresponds to the **I**nput **MEM**ory

operand of the 15th dynamic instruction[1], the second fragment corresponds to the input memory operand of the 10th dynamic instruction, and so on. The second line is about the round keys parameter. This parameter is divided into sixty 32-bit fragments. Luckily we were able to employ a condensed formulation to report their location. When all the fragments of a parameter are mapped to memory operations (either `load` or `store`), we try to find whether they belong to the same memory buffer. That is to say, we try to find if their addresses are equal to $x + c_i$, where x is the same term for every address and c_i is a constant term such that $c_{i+1} = c_i + s_i$, where s_i denotes the size of the i^{th} fragment. If this is the case, the parameter is simply reported as $x[min(c_i), max(c_i + s_i)]$. Back to our example, the round keys parameter is located in a memory buffer that starts at address $\overline{\texttt{eax}_1} + 16$ and ends at address $\overline{\texttt{eax}_1} + 256$, where $\overline{\texttt{eax}_1}$ denotes the value of `eax` before the 1st dynamic instruction. The third line is about the ciphertext parameter. This parameter is divided into four 32-bit fragments. The first fragment corresponds to destination operand of the 868th dynamic instruction, the second fragment to the destination operand of the 891st dynamic instruction, and so on.

Note that our analysis tool reports only one possible location for the parameters even though four matches of the AF256 signature were found. As previously explained, the four matches of the AF256 signature are caused by symmetries in the underlying AT4 and AL_V9 signatures. The only difference between these four concurrences is the ordering of the parameter fragments. For simplicity we have presented only the first possible ordering. As we will see in the next paragraph, for this example it is relatively easy to determine the right ordering in practice.

6.1.5 Test Vectors

In order to check the 7-Zip AES implementation, we relied on the test vector of the NIST for AES 256 [1]. Using a debugger, we can dynamically assign a new value to the input parameters. Then, once the algorithm has been executed, we can check that we obtained the expected results. We wrote a simple WinDbg [56] script to perform these actions. This script was written manually, but writing it in an automated way should not raise any significant difficulty. It is given below.

```
ff Set the plaintext
bp 0x100bfd78 "ed @edi 0x33221100; g";     ff @11: 0x100bfd78 xor edx, [edi]
bp 0x100bfd72 "ed @edi+0x4 0x77665544; g"; ff @9 : 0x100bfd72 xor esi, [edi+0x4]
bp 0x100bfd83 "ed @edi+0x8 0xbbaa9988; g"; ff @15: 0x100bfd83 xor ecx, [edi+0x8]
bp 0x100bfd75 "ed @edi+0xc 0xffeeddcc; g"; ff @10: 0x100bfd75 xor ebx, [edi+0xc]

ff Set the round keys @1: 0x100bfd63 mov edx, [eax+0x10]
bp 0x100bfd63 "ed @eax+0x10 0x03020100 0x07060504 [...] 0x36de686d; g"

ff Check the ciphertext
bp 0x100c018e "r @edx; g"; ff @868: 0x100c018b xor edx, [eax+0x14]
bp 0x100c01e6 "r @ebp; g"; ff @891: 0x100c01e3 xor ebp, [eax+0x18]
bp 0x100c0237 "r @ecx; g"; ff @911: 0x100c0234 xor ecx, [eax+0x1c]
bp 0x100c0136 "r @edx; g"; ff @845: 0x100c0133 xor edx, [eax+0x10]
```

For each dynamic instruction index, we recall in the comments the address and the value of the instruction. Even though our analysis tool failed to detect it as such, the plaintext parameter is located in a memory buffer: $\overline{\texttt{edi}_9}[0, 16]$, where $\overline{\texttt{edi}_9}$ denotes the value of `edi` before the 9th dynamic instruction. That being said, the correct ordering of the plaintext parameter is easy to guess. The first fragment (test vector value: `0x33221100`) goes to `[edi]`, the second fragment (test vector value: `0x77665544`) goes to `[edi + 0x4]` and so on. The round keys parameter is quite long and for brevity most of it has been omitted in the code above. This script returned the following output:

[1]For clarity, we use here a sub-index to count dynamic instructions. It is initialised to zero at the beginning of the current trace segment, that is to say, after 770757133 dynamic instructions if we count from the beginning of the execution trace of the second thread.

```
edx=cab7a28e, edx=bf456751, ebp=9049fcea, ecx=8960494b
```

Fragments of the NIST test vector are all found back. The AES implementation used in 7-Zip is compliant with the NIST standard.

6.1.6 Conclusion

The first objective was to demonstrate that our solution is able to cope with real life software. The 7-Zip example is well suited since the traditional approach based on constant detection gives poor results. A manual search revealed the encryption function but mixed with three other functions. The inverse substitution box is computed dynamically. Thus the decryption algorithm could not have been found using static constant detection. A second objective was to illustrate how one can take advantage of precise parameter location to evaluate the correctness of an implementation by running test vectors. Translation from vertices back to assembly instructions still suffers from a couple of imprecisions but we have never experienced them in practice.

6.2 Complete Analysis of an Instant Messaging Application

For simplicity reasons, most of the experiments presented in Chapter 4 and 5 were limited to synthetic samples. In this section, we apply our solution to a more substantial program: the Telegram client for Linux. Telegram is an instant messaging service that uses a custom encryption scheme called MtProto. Brief specifications of this protocol can be found on the editor's website [69]. Official client applications are available for several operating systems and they are all open source. Thus, it will be easy to check the validity of our findings.

6.2.1 Execution Trace

This example is based on the official Telegram client for Linux 32-bit, version 32.1.0.29. The size of the binary is 74 MB. It is partially stripped. Based on the remaining symbols, it is clear that it includes several statically linked libraries such as Qt.

We used our pintool to collect an execution trace. The only white-listed shared library was the C standard library. We know from our experience that many implementations of modes of operation use *memcpy*. We preventively decided to trace the C standard library to cover completely any modes operation even when they use *memcpy*. To be able to analyse modes of operation, we also saved runtime address values in the execution trace. During the recorded execution, we simply launched the program, waited for the main window to appear and closed the main window which caused the program to return.

The tracing process was rather long and took approximatively 30 minutes. More than a dozen of threads were executed. The size of the trace is around 6 GB and 90% of it consists of runtime address values. In this example we limit our analysis to the first thread. More than 1 billion dynamic instructions were recorded for the first thread.

6.2.2 Basic Heuristics for Segment Selection

We used four simple heuristics to rapidly discover the location of possible cryptographic primitives. The first three are based on the notion of basic block and the last one on the notion of function execution.

- We discard basic blocks which contains fewer than 40 instructions. Symmetric cryptography algorithms have very few conditional statements resulting in large basic blocks.
- We discard basic blocks which contain less than 25% of logical bitwise instructions.
- We discard basic blocks which were executed fewer than five times. This threshold was fixed rather arbitrarily. The general idea is to consider only basic blocks which are executed a large number of times. In fact, if a basic block belongs to a round function, it will be executed several times as part of the primitive and the primitive itself is likely to be executed several times depending on the amount of data to be processed.

Table 6.2: List of functions returned by the segment selection method described in Section 6.2.2. In the last column we specify for each of them the name of primitive that was detected.

Start Address	Number of Execution	Result
0x09abfe00	131	∅
0x09abe820	40	AES-256 Encryption OpenSSL
0x0846b2a0	54	Compression Function MD5
0x099e1db0	54	Compression Function MD5
0x09ab8cc0	379	Compression Function SHA1

- We discard function executions which contain function calls. Symmetric cryptographic algorithms are usually implemented in a single function that does not call any sub-function. Note that compiler tricks to retrieve the program counter value such as the function given below, do not count as sub-functions.

```
mov ebx, dword ptr [esp]
ret
```

First we select all basic blocks which simultaneously satisfy the first three heuristics. From the sixteen thousands basic blocks originally included in the execution trace, only 42 survive this first selection process. Then, we select every function all executions of which satisfy the last heuristic and contain at least one of the previously selected basic block. Function executions are determined based on the count of `call` and `ret` instructions. At the end we obtain four functions.

6.2.3 Primitive Identification

The four functions returned by the segment selection method described above are given in Table 6.2. For each of them we randomly extracted one of their executions and we analysed it with our primitive identification method. Results are given in the last column of Table 6.2. No primitive was detected for the first function. It is probably the AES-256 decryption implementation of OpenSSL, but this specific implementation was not covered by our set of signatures. If this hypothesis is correct, the heuristics used to rapidly locate the cryptographic primitives returned no false positive. To analyse the mode operation we used the location of the AES-256 encryption primitive as a starting point.

6.2.4 Block Cipher Mode of Operation

The AES-256 encryption primitive is executed 40 times in the execution trace. Each execution is separated by exactly 71 dynamic instructions. This regularity suggests that AES-256 is executed as part of a mode of operation. The function that calls the AES-256 encryption primitive is probably just a wrapper since it contains very few instructions of its own and only executes AES-256 once. The function that calls the wrapper may correspond to the encryption mode of operation since it contains all the 40 executions of AES-256. The execution trace of this latter function is quite large (72 thousand dynamic instructions), thus we only analysed a piece of it containing five executions of AES-256. The slice that was obtained is given in Figure 6.1.

It is not easy to recognize this mode of operation at first glance. We can make two observations though. First the key seems to be the same for every execution of AES-256 and second, the same data is XORed with the input of the i^{th} encryption and with the output of the $(i+1)^{\text{th}}$ encryption.

As stated in the specifications of MtProto this mode of operation should be Infinite Garble Extension (IGE). IGE has practically never been used apart by Telegram. A message block $M[i]$ is encrypted in a ciphertext block $C[i]$ according to the following formula:

$$C[i] = E_k(M[i] \oplus C[i-1]) \oplus M[i-1]$$

One should have no trouble to recognize IGE in the slice of Figure 6.1. The completeness and the readability are optimal.

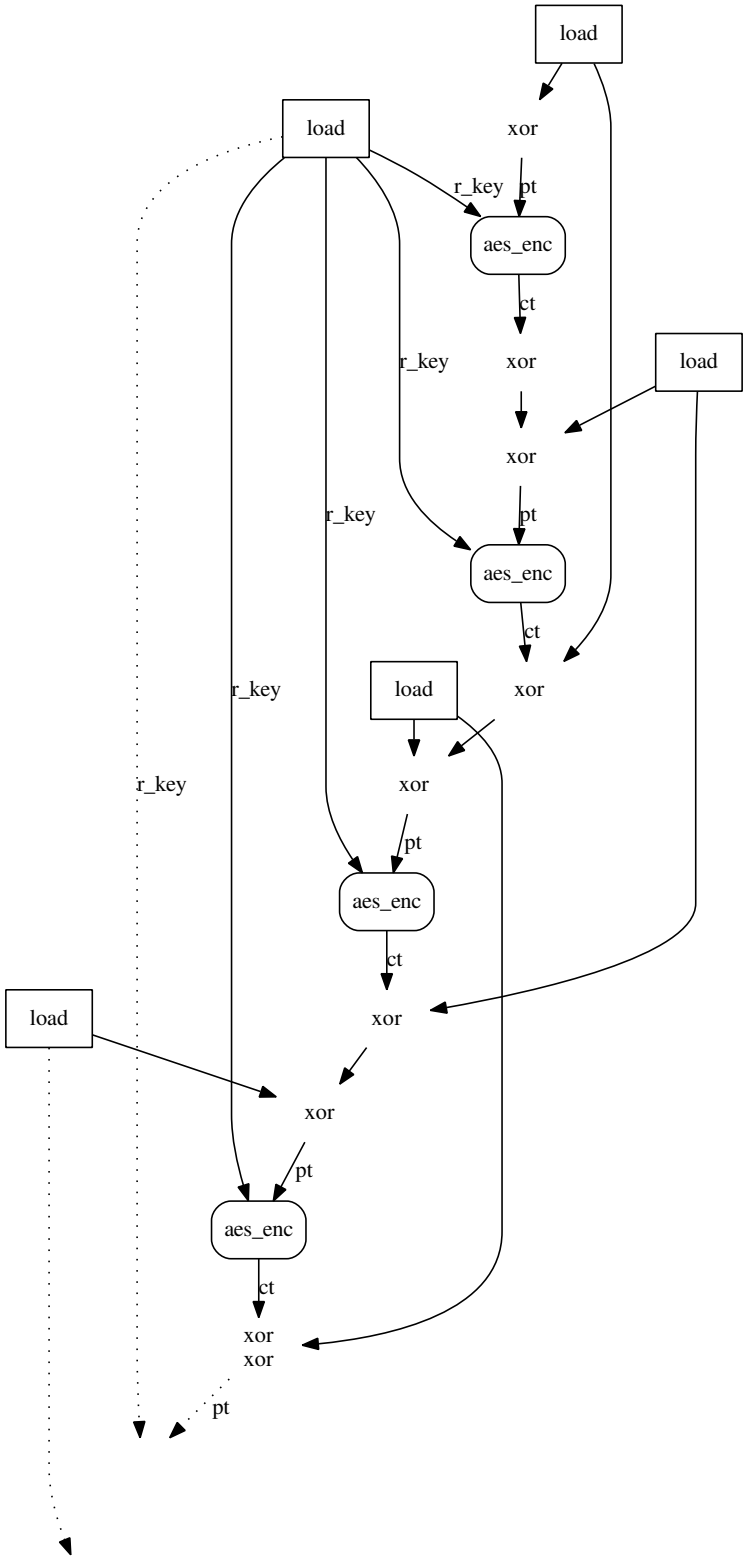

Figure 6.1: Experimental slice obtained for a trace segment containing five executions of the AES_{256} encryption primitive. The bottom-most edges are depicted with dotted lines to symbolize how one could extrapolate the current slice for a larger trace segment. This slice was generated in 8 ms on a Pentium Dual-Core T4200 processor out of a DFG with 531 vertices and 853 edges.

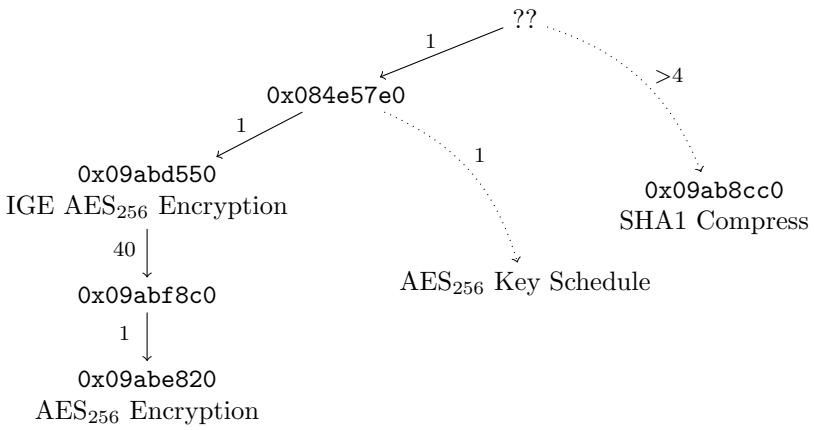

Figure 6.2: Synthesis of the call stack. Edge labels specify the number of times sub-functions are called. We did not investigate precisely the number of intermediate functions along the dotted edges.

6.2.5 Key and IVs Derivation Function

In this section we analyse how the key and the two IVs required by IGE are generated. To find a relevant trace segment we proceed as we did for IGE, that is to say, we go upward in the call stack. Our starting point is the IGE function that we uncovered in the previous section. The caller function executes 750 instructions before the IGE function and 4 instructions after. We extracted in a trace segment the 750 instructions which precede the IGE function, and with our primitive identification method we found out that they implement an AES_{256} key schedule. This function is related to the AES encryption but does not contains what we are looking for, so we move to the caller function. We had difficulties to estimate the size of the caller function because we reached pieces of code which belong to black-listed shared libraries. The one thing that is certain though, is that the function contains several executions of the SHA1 compression function which was previously identified in Section 6.2.3. A synthesis of the call stack that has been described so far is given in Figure 6.2.

According to the specification of MtProto four SHA1 hashes are computed from a *message key* and different parts of a so called *shared key*. These four hashes are noted respectively `sha1_a`, ..., `sha1_d`. For none of them the input of the SHA1 function exceed 447 bits, thus the SHA1 compression function is only called once for each of these hashes. The AES key and the two IGE IVs derive from these four hashes according to the following formulae:

$$
\begin{aligned}
Key &\leftarrow \texttt{sha1\_a}[0:7] \,||\, \texttt{sha1\_b}[8:19] \,||\, \texttt{sha1\_c}[4:15] \\
IV1 &\leftarrow \texttt{sha1\_a}[8:19] \,||\, \texttt{sha1\_b}[0:3] \\
IV2 &\leftarrow \texttt{sha1\_b}[4:7] \,||\, \texttt{sha1\_c}[16:19] \,||\, \texttt{sha1\_d}[0:7]
\end{aligned}
$$

To capture this computation in a slice, we extracted a trace segment containing the first AES_{256} encryption (to easily visualize the two IVs), the AES_{256} key schedule and the last four executions of the SHA1 compression function. Each primitive was identified separately. The resulting slice is presented in Figure 6.3. It correctly reflects the specifications. To facilitate its interpretation we have pinpointed the location of the two IVs and we have identified each hash function with respect to the notation introduced above. The completeness and the readability of this slice are optimal.

6.2.6 Conclusion

We show with this example that our solution is usable even on large programs. With a few basic heuristics and a little bit of human interaction it is easy to find the right trace segments. We do not deny that having access to the specifications of MtProto was of great help to both, select the right trace segment (especially for the key and the IVs derivation function) and to interpret the

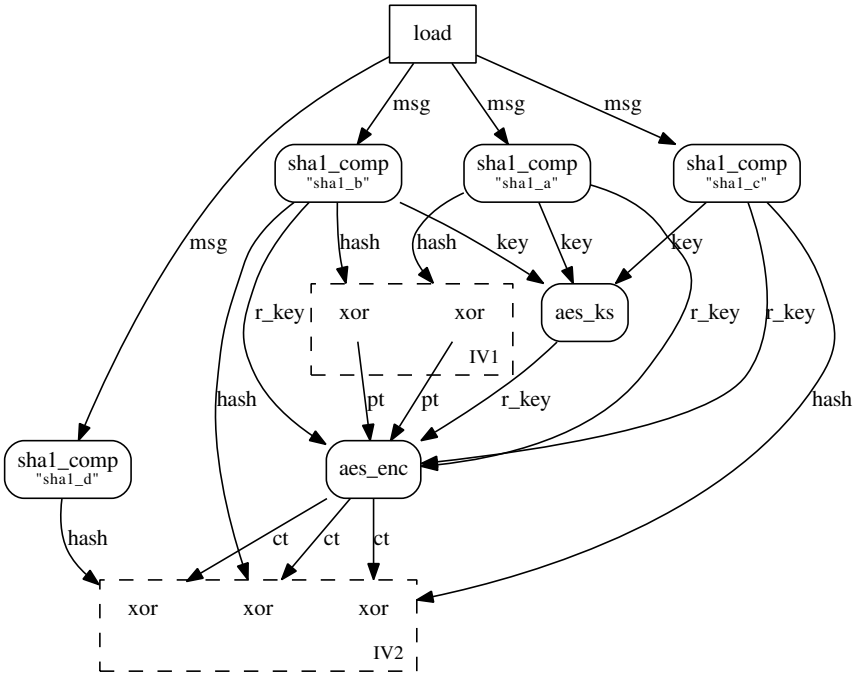

Figure 6.3: Experimental slice obtained for a trace segment containing the first AES_{256} encryption, and the last four executions of the SHA1 compression function. This slice was generated in 9 ms on a Pentium Dual-Core T4200 processor out of a DFG with 2535 vertices and 2818 edges. The two dashed rectangle boxes as well as the mentions of `sha1_a`, ..., `sha1_d` were added manually to clarify the figure.

slices. With respect to this last point, we acknowledge that without any prior knowledge on the mode of operation it is relatively difficult to draw reliable conclusions from the manual study of a slice. As future work to improve on this aspect, we plan to add a size attribute to the edges of a slice. The size of an edge could be defined as the minimal operation size along a path sum over all the parallel paths (that is to say all the paths of equal length which connect the same two parameters but do not intersect). With this kind of information, one would have known from the study of the key derivation slice (Figure 6.3) how many bits of the AES key come from each hash function. The idea is to increase the accuracy of the information reported in slices without inserting additional connections which would probably ruin their readability. Here, instead of only reporting that a connection exists between two parameters, we want to specify the number of bits which are concerned.

Appendix A

Cryptographic Algorithms Background

In this appendix, we give a short description of the main cryptographic algorithms mentioned in this document.

A.1 Primitives

A.1.1 Description of AES

The Advanced Encryption Standard [21] is a Substitution Permutation Network (SPN) that can be instantiated using three different key lengths: 128-bit, 192-bit, and 256-bit. The 128-bit plaintext initialises the internal state viewed as a 4×4 matrix of bytes seen as elements of the finite field $GF(2^8)$, which is defined via the irreducible polynomial $x^8+x^4+x^3+x+1$ over $GF(2)$. Depending on the version of Advanced Encryption Standard (AES), N_r rounds are applied to that state: $N_r = 10$ for AES-128, $N_r = 12$ for AES-192 and $N_r = 14$ for AES-256. Each of the N_r AES round applies four operations to the state matrix (except the last one where the *MixColumns* is omitted):

- *AddRoundKey*: adds a 128-bit round key to the state ;
- *SubBytes*: applies the same 8-bit to 8-bit invertible S-Box S sixteen times in parallel on each byte of the state ;
- *ShiftRows*: shifts the i^{th} row left by i positions ;
- *MixColumns*: replaces each of the four columns C of the state by $M \times C$ where M is a constant 4×4 maximum distance separable matrix over $GF(2^8)$.

After the N_r^{th} round has been applied, a final round key is added to the internal state to produce the ciphertext. A key expansion algorithm is used to produce the $N_r + 1$ round keys required for all AES variants.

Implementation Remarks. The *ShiftRows* and the *MixColumns* can be combined with the *SubBytes* resulting in four lookup tables of one kilobyte each. We introduce the following notations: A_i is the state at round i, divided in four 32-bits word and T_i $(0 \leq i \leq 3)$ is a function that given a 32-bit word, extracts the i^{th} most significant byte and returns the associated 32-bit word in the i^{th} lookup table. For each full round $(1 \leq i \leq N_r - 1)$, the combination of the *ShiftRows*, *MixColumns* and *SubBytes* can be implemented using the following pseudo-code (we have omitted the *AddRoundKey* for brevity):

$$
\begin{aligned}
A_{i+1}[0] &= T_0(A_i[0]) \oplus T_1(A_i[1]) \oplus T_2(A_i[2]) \oplus T_3(A_i[3]) \\
A_{i+1}[1] &= T_0(A_i[1]) \oplus T_1(A_i[2]) \oplus T_2(A_i[3]) \oplus T_3(A_i[0]) \\
A_{i+1}[2] &= T_0(A_i[2]) \oplus T_1(A_i[3]) \oplus T_2(A_i[0]) \oplus T_3(A_i[1]) \\
A_{i+1}[3] &= T_0(A_i[3]) \oplus T_1(A_i[0]) \oplus T_2(A_i[1]) \oplus T_3(A_i[2])
\end{aligned}
$$

This implementation is the most widespread and it is usually referred to as the table implementation. However, it is not the only way to efficiently implement AES. Matsui *et al.* [54] and Käsper *et al.* [41] proposed two bitsliced implementations. In bitsliced modes, several blocks are processed in parallel taking advantage of the SIMD architecture. Nevertheless bitsliced can only be used in parallel modes of operation (such as CTR for instance). Hamburg [37] demonstrated that it is feasible to implement a single block AES encryption with vector permute instructions. Finally recent CPUs have dedicated AES instructions to reach the best performance and the highest security levels. These alternative implementations have been mentioned here for completeness. This work only covers the table implementation.

A.1.2 Description of MD5

MD5 [62] is a cryptographic hash function which produces 128-bit hashes. At a high level, MD5 is structured according to the Merkle-Damgård construction [23, 55]. The message is divided into 512-bit blocks with a padding being applied to the last block. Each message block M_i is processed by a compression function f which also takes as input argument a 128-bit chaining value. The chaining value, noted H_i, is updated as follows:

$$H_{i+1} = f(M_i, H_i)$$

H_0 is initialised with a fixed constant. The last value H_i is the output of the hash function. The structure of the compression function follows a Davies-Meyer construction. Its underlying block cipher is a four-branch Feistel network that operates on a 128-bit state. The step function is executed 64 times. Steps are regrouped in four rounds of 16 consecutive steps each. The internal state is divided in four 32-bit words A_i, B_i, C_i and D_i. At each step i, a 32-bit word of the message block, noted W_i, is used to update the internal state according to the following formulae:

$$
\begin{aligned}
A_{i+1} &\leftarrow D_i \\
B_{i+1} &\leftarrow B_i + \circlearrowleft (A_i + \phi_i(B_i, C_i, D_i) + W_i + K_i), S_i) \\
C_{i+1} &\leftarrow B_i \\
D_{i+1} &\leftarrow C_i
\end{aligned}
$$

K_i and S_i are fixed constants and ϕ_i denotes one of the four following boolean functions:

$$
\begin{array}{ll}
\text{for } i \in [1, 16] & \phi_i(x, y, z) = (x \wedge y) \vee (\neg x \wedge z) \\
\text{for } i \in [17, 32] & \phi_i(x, y, z) = (x \wedge z) \vee (y \wedge \neg z) \\
\text{for } i \in [33, 48] & \phi_i(x, y, z) = x \oplus y \oplus z \\
\text{for } i \in [49, 64] & \phi_i(x, y, z) = y \oplus (x \wedge \neg z)
\end{array}
$$

Implementation Remarks. Usually the 64 steps are directly unrolled in the source code. This is for instance the case of the reference implementation given in the RFC [62]. Many MD5 implementations use alternative formulae for the boolean function ϕ_i used in the first and in the second round. These alternative formulae take on less operator and therefore, are supposed to be more efficient. They are given below:

$$
\begin{array}{ll}
\text{for } i \in [1, 16] & \phi_i(x, y, z) = z \oplus (x \wedge (y \oplus z)) \\
\text{for } i \in [17, 32] & \phi_i(x, y, z) = y \oplus (z \wedge (y \oplus x))
\end{array}
$$

A.1.3 Description of RC4

RC4 is a stream cipher also known as ARC4 (for Alleged RC4). Its internal state consists of a permutation of all the 256 possible bytes noted S and two 8-bit index pointers noted respectively i and j. The initial state of the permutation derives from a variable-size key. This first part of the primitive is called the Key Schedule Algorithm (KSA) and is not covered by this work. The key-stream is generated by the second part of the primitive called the PRGA. Each iteration of the PRGA updates the permutation S and outputs one byte of key-stream according to the pseudo code given in Algorithm 10. The PRGA is executed as many times as necessary to obtain a key-stream of the desired size.

Algorithm 10 RC4 PRGA

$i \leftarrow i + 1$
$j \leftarrow S[i] + i$
swap $S[i]$ and $S[j]$
output $S[S[i] + S[j] \pmod{256}]$

A.1.4 Description of SHA1

SHA1 [4] is hash function which produces 160-bit hashes. It shares many characteristics with MD5. At a high level it is also structured according to the Merkle-Damgård construction and its compression function also follows a Davies-Meyer framework. Its underlying block cipher is a five-branch Feistel network that operates on a 160-bit state divided into five 32-bit words A_i, B_i, C_i D_i and E_i. The step function is executed 80 times. Steps are regrouped in four rounds of 20 consecutive steps each. At each step i, a 32-bit expanded message word W_i derived from the message block is used to update the internal state according to the following formula:

$$
\begin{aligned}
A_{i+1} &\leftarrow \circlearrowleft (A_i, 5) + \phi_i(B_i, C_i, D_i) + E_i + W_i + K_i \\
B_{i+1} &\leftarrow A_i \\
C_{i+1} &\leftarrow \circlearrowleft (B_i, 30) \\
D_{i+1} &\leftarrow C_i \\
E_{i+1} &\leftarrow D_i
\end{aligned}
$$

K_i is a fixed constant and ϕ_i denotes one of the four following boolean functions:

$$
\begin{aligned}
&\text{for } i \in [1, 20] && \phi_i(x, y, z) = (x \wedge y) \vee (\neg x \wedge z) \\
&\text{for } i \in [21, 40] && \phi_i(x, y, z) = x \oplus y \oplus z \\
&\text{for } i \in [41, 60] && \phi_i(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) \\
&\text{for } i \in [61, 80] && \phi_i(x, y, z) = x \oplus y \oplus z
\end{aligned}
$$

The expanded message words W_i derive from the 512-bit message block. The message block is divided into sixteen 32-bit words, W_1 is assigned to the first word, W_2 to the second word and so on up to W_{16}. The remaining expanded message words: W_{17} to W_{80}, are computed recursively according to the following formulae:

$$W_i \leftarrow \circlearrowleft (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}, 1)$$

Implementation Remarks. Unlike MD5, the 80 steps are generally not unrolled in the source code. To reduce memory consumption the message expansion can be done gradually during the 80 steps. Only the sixteen last expanded message words are required to compute the next expanded message word. Thus, the space needed to store the expanded message words can be reduced from eighty 32-bit words to sixteen 32-bit words. Another possibility regarding message expansion is to use SIMD instructions as described in [51]. Finally the exact boolean expression used by the ϕ_i functions may vary from one implementation to another. For instance in OpenSSL the majority

function (the third boolean function used from step 41 to step 60) is either implemented using $(x \wedge y) \vee (z \wedge (x \oplus y))$ or $((x \oplus z) \wedge (y \oplus z)) \oplus z$ depending on the version.

A.1.5 Description of TEA & XTEA

Tiny Encryption Algorithm (TEA) [57] is a 64-bit block cipher with a 128-bit key. It is based on a two-branch Feistel network that operates on a 64-bit state. Rounds are usually regrouped in pairs forming cycles. The recommended number of cycles is 32. TEA suffers from related-key attacks [42]. To solve this weakness Needham and Wheeler proposed an extended version of TEA named XTEA. XTEA has a different key scheduling and a different round function.

Implementation Remarks. Both TEA and XTEA have been designed as small C programs performing simple operations on 32-bit words. The only implementation variation we are aware of concerns the key scheduling. Since the key scheduling is extremely simple some implementations do not compute the round keys separately but they do it directly in each round.

A.2 Modes of Operation

A.2.1 CBC

The CBC mode of operation is one of the most-used chaining mode. It is used to encrypt large messages using a block cipher. To encrypt a message M, it is split into blocks M_i, the size of which is equal to the input length of the block cipher. If the length of the message is not a multiple of the block size, the message is padded. The most-used padding scheme simply consists in adding a bit 1 and as many bits set to 0 as needed. Then the idea consists in randomizing the input of the block cipher using a random value. The first message block is randomized using an initialisation vector, while block M_i is randomized using the output of M_{i-1}. We choose $C_0 = IV$ uniformly and we compute iteratively:

$$C_i \leftarrow E_k(M_i \oplus C_{i-1})$$

This makes the scheme non-parallelizable. But it has the advantage of being self-synchronizing: an error in one block C_i only affects two blocks M_i and M_{i-1}. The decryption is parallelizable and it is done according to the following formula:

$$M[i] \leftarrow C_{i-1} \oplus D_k(C_i)$$

A.2.2 CTR

The counter mode, abbreviated CTR, is used to encrypt arbitrarily long messages using a block cipher. The idea is to use a block cipher to encrypt a counter. It generates a bit-string which is indistinguishable from a random bit-string up to the birthday bound. Then we perform a one-time pad with this bit-string and the message. More precisely, a message M is divided in blocks M_i the size of which is equal to the input length of the block cipher, except for the last block which might be shorter (no padding is necessary). The first ciphertext block C_0 is set with the initial counter value. Then:

$$C_i \leftarrow M_i \oplus E_k(C_0 + i)$$

The decryption is exactly the same as the encryption. This mode is parallelizable.

A.2.3 HMAC

HMAC, standing for keyed-Hash Message Authentication Code, was defined by Bellare, Canetti and Krawczyk in 1995 and it has been extensively used in many RFCs [10] since. It is a message authentication code based on a cryptographic hash function. It is compliant with the cryptographic

hash functions following the Merkle-Dangård construction such as MD5 and SHA1. The Merkle-Dangård construction is prone to length extension attack. In particular if H is a hash function of the Merkle-Dangård family, the construction $H(k \mid\mid M)$ is not secure. The idea to avoid length extension attack consists in hashing the output of this former expression with another key. Given a hash function H, a message M and a key k, HMAC outputs:

$$H(k \oplus \texttt{opad} \mid\mid H(k \oplus \texttt{ipad} \mid\mid M))$$

A.2.4 OCB

The Offset Codebook Mode (OCB) has been defined by Rogaway [64]. It is an authenticated encryption mode of operation. It ensures confidentiality and authentication through a single pass over the message. The idea consists in XORing a random mask Z_i over the plaintext M_i and on the ciphertext:

$$C_i \leftarrow E_k(M_i \oplus Z_i) \oplus Z_i$$

The mask evolves between each call based on a linear relation $Z_i \leftarrow \gamma_i \cdot L \oplus R$, where $L = E_k(0^n)$ and $R = E_k(N \oplus L)$ and N is a random nonce and the multiplication is performed in some finite field of 2^{128} elements. There is a much more efficient way of computing Z_i from Z_{i-1} but we do not need such details. Finally the authentication tag T is equal to:

$$T \leftarrow E_k((\oplus_{i=1}^{m} M_i) \oplus Z_m)$$

In Section 5.5.2 we describe more precisely how the last message block is encrypted. There are some technicalities in order to make a ciphertext stealing mode, so that the output length of the ciphertext part is as long as the plaintext.

Bibliography

[1] Advanced encryption standard (AES). Technical report, NIST, 2001.

[2] Hex-rays IDA FLIRT technology: In-depth. Technical report, Hex-Rays, 2011.

[3] PKCS #1 v2.2: RSA cryptography standard. Technical report, October 2012.

[4] FIPS 180-4: Secure hash standard. Technical report, National Institute of Standards and Technology, 2015.

[5] CVE-2016-3995. Available from MITRE, CVE-ID CVE-2016-3995., April 2016.

[6] Donald E. Eastlake 3rd and Paul Jones. US secure hash algorithm 1 (SHA1). RFC 3174, September 2001.

[7] ASProtect. `http://www.aspack.com/asprotect32.html`. Executable Packer.

[8] Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer, 2004.

[9] Gogul Balakrishnan and Thomas W. Reps. WYSINWYX: what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, 2010.

[10] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[11] Mihir Bellare, Joseph Jaeger, and Daniel Kane. Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1431–1440. ACM, 2015.

[12] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In Garay and Gennaro [29], pages 1–19.

[13] Alex Biryukov. The design of a stream cipher LEX. In *Selected Areas in Cryptography*, pages 67–75, 2006.

[14] Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[15] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Xiaodong Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 621–634. ACM, 2009.

[16] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. Aligot: cryptographic function identification in obfuscated binary programs. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 169–182. ACM, 2012.

[17] Christophe Clavier and Kris Gaj, editors. *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*. Springer, 2009.

[18] Don Coppersmith and Michael Elkin. Sparse sourcewise and pairwise distance preservers. *SIAM J. Discrete Math.*, 20(2):463–501, 2006.

[19] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, pages 149–159, 2001.

[20] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, Francesco Tortorella, and Mario Vento. Graph matching: a fast algorithm and its evaluation. In Anil K. Jain, Svetha Venkatesh, and Brian C. Lovell, editors, *Fourteenth International Conference on Pattern Recognition, ICPR 1998, Brisbane, Australia, 16-20 August, 1998*, pages 1582–1584. IEEE Computer Society, 1998.

[21] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[22] Joan Daemen and Vincent Rijmen. The Pelican MAC function. *IACR Cryptology ePrint Archive*, 2005:88, 2005.

[23] Ivan Damgård. A design principle for hash functions. In Brassard [14], pages 416–427.

[24] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.

[25] The Dot language. `http://www.graphviz.org/doc/info/lang.html`.

[26] Noé Lutz. Towards revealing attackers' intent by automatically decrypting network traffic. Master thesis, ETH Zurich, 2008.

[27] EasyLock. `http://www.endpointprotector.com/products/easylock`. File Encryption Software.

[28] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 303–317. USENIX Association, 2014.

[29] Juan A. Garay and Rosario Gennaro, editors. *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*. Springer, 2014.

[30] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Garay and Gennaro [29], pages 444–461.

[31] Brian Gladman. AES implementation. `http://www.gladman.me.uk/AES`.

[32] Felix Gröbert. Automatic identification of cryptographic primitives in software. Master thesis, Ruhr-University Bochum, 2010.

[33] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automated identification of cryptographic primitives in binary programs. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *RAID*, volume 6961 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2011.

[34] Ilfak Guilfanov. FindCrypt2. http://www.hexblog.com/?p=28, February 2006. IDA Plugin.

[35] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 45–60. USENIX Association, 2008.

[36] Shai Halevi, William Eric Hall, and Charanjit S. Jutla. The hash function "Fugue". *IACR Cryptology ePrint Archive*, 2014:423, 2014.

[37] Mike Hamburg. Accelerating AES with vector permute instructions. In Clavier and Gaj [17], pages 18–32.

[38] Hex-Rays. IDA. https://www.hex-rays.com/products/ida/index.shtml. Interactive Disassembler.

[39] Berthold Hoffmann and Detlef Plump. Implementing term rewriting by jungle evaluation. *ITA*, 25:445–472, 1991.

[40] Intel. XED. https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library. Intel X86 Encoder Decoder Software Library.

[41] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Clavier and Gaj [17], pages 1–17.

[42] John Kelsey, Bruce Schneier, and David Wagner. Related-key cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA. In Yongfei Han, Tatsuaki Okamoto, and Sihan Qing, editors, *Information and Communication Security, First International Conference, ICICS'97, Beijing, China, November 11-14, 1997, Proceedings*, volume 1334 of *Lecture Notes in Computer Science*, pages 233–246. Springer, 1997.

[43] Wei Ming Khoo, Alan Mycroft, and Ross J. Anderson. Rendezvous: a search engine for binary code. In Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors, *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 329–338. IEEE Computer Society, 2013.

[44] Makoto Kobayashi. Dynamic characteristics of loops. *IEEE Trans. Computers*, 33(2):125–132, 1984.

[45] Tímea Làszló and Àkos Kiss. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica*, 30:3–19, 2009.

[46] Pierre Lestringant. Automated reverse engineering of cryptographic algorithms. http://blog.amossys.fr/Automated%20Reverse%20Engineering%20of%20Cryptographic%20Algorithms.html, May 2015.

[47] Xin Li, Xinyuan Wang, and Wentao Chang. CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution. *IEEE Trans. Dependable Sec. Comput.*, 11(2):101–114, 2014.

[48] Nettle. http://www.lysator.liu.se/~nisse/nettle/. Cryptographic Library.

[49] LibTomCrypt. http://www.libtom.net/LibTomCrypt/. Cryptographic Library.

[50] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In Moti Yung, editor, *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2002.

[51] Max Locktyukhin. Improving the performance of the secure hash algorithm (SHA1). Technical report, Intel, March 2010.

[52] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 190–200. ACM, 2005.

[53] Carsten Maartmann-Moe, Steffen E. Thorkildsen, and André Arnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. *Digital Investigation*, 6:S132 – S140, 2009. The Proceedings of the Ninth Annual DFRWS Conference.

[54] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on Intel Core2 processor. In *CHES*, pages 121–134, 2007.

[55] Ralph C. Merkle. One way hash functions and DES. In Brassard [14], pages 428–446.

[56] Microsoft. WinDbg. `https://msdn.microsoft.com/fr-fr/library/windows/hardware/ff551063%28v=vs.85%29.aspx`. Debugger for Windows.

[57] Roger M. Needham and David J. Wheeler. TEA, a tiny encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 363–366. Springer, 1994.

[58] Kenneth Oksanen. Detecting algorithms using dynamic analysis. In Leonardo Mariani and Xiangyu Zhang, editors, *Proceedings of the International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2011), WODA 2011, Toronto, ON, Canada, July 18, 2011.*, pages 1–6. ACM, 2011.

[59] Igor Pavlov. 7-Zip. `http://www.7-zip.org/`. Open-source file archiver.

[60] Detlef Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51:277–289, 2001.

[61] Radare2. `http://radare.org/r/`. Reverse Engineering Framework.

[62] Ron L. Rivest. The MD5 message-digest algorithm. RFC 1321, April 1992.

[63] Juliano Rizzo and Thai Duong. BEAST: Surprising crypto attack against https. EKOPARTY security conference 7th edition, 2011.

[64] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: a block cipher mode of operation for efficient authenticated encryption. In Michael K. Reiter and Pierangela Samarati, editors, *CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, USA, November 6-8, 2001.*, pages 196–205. ACM, 2001.

[65] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel J. Quinlan, and Zhendong Su. Detecting code clones in binary executables. In Gregg Rothermel and Laura K. Dillon, editors, *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 117–128. ACM, 2009.

[66] Michael Steil. 17 mistakes Microsoft made in the Xbox security system. In *Der Chaos Communication Congress*, 2005.

[67] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA1. *IACR Cryptology ePrint Archive*, 2017:190, 2017.

[68] Petr Svenda, Matús Nemec, Peter Sekan, Rudolf Kvasnovský, David Formánek, David Komárek, and Vashek Matyás. The million-key question - investigating the origins of RSA public keys. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 893–910. USENIX Association, 2016.

[69] Secret chats, end-to-end encryption. `https://core.telegram.org/api/end-to-end`. Technical description of mtProto used in Telegram.

[70] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[71] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[72] Julian R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[73] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002.

[74] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Steal this movie: Automatically bypassing DRM protection in streaming media services. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 687–702. USENIX Association, 2013.

[75] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 497–512. IEEE Computer Society, 2010.

[76] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. ReFormat: Automatic reverse engineering of encrypted messages. In Michael Backes and Peng Ning, editors, *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, volume 5789 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2009.

[77] A. F. Webster and Stafford E. Tavares. On the design of s-boxes. In Hugh C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 523–534. Springer, 1985.

[78] David J. Wheeler and Roger M. Needham. TEA extensions. Technical report, Computer Laboratory, University of Cambridge, 1997.

[79] Tao Xie, Fanbao Liu, and Dengguo Feng. Fast collision attack on MD5. *IACR Cryptology ePrint Archive*, 2013:170, 2013.

[80] Ruoxu Zhao, Dawu Gu, Juanru Li, and Hui Liu. Detecting encryption functions via process emulation and IL-based program analysis. In Tat Wing Chim and Tsz Hon Yuen, editors, *Information and Communications Security - 14th International Conference, ICICS 2012, Hong Kong, China, October 29-31, 2012. Proceedings*, volume 7618 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2012.

[81] Ruoxu Zhao, Dawu Gu, Juanru Li, and Ran Yu. Detection and analysis of cryptographic data inside software. In Xuejia Lai, Jianying Zhou, and Hui Li, editors, *Information Security, 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings*, volume 7001 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2011.