Outils pour l'analyse de code et de contre-mesures pour l'injection de fautes multiples

Etienne Boespflug

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : VERIMAG

Outils pour l'analyse de code et de contre-mesures pour l'injection de fautes multiples

Tools for code and countermeasures analysis against multiple faults attacks

Présentée par :

Etienne BOESPFLUG

Direction de thèse :

Marie Laure POTET Professeur des Universités, GRENOBLE INP	Directrice de thèse
Laurent MOUNIER Maître de conférences, UNIVERSITE GRENOBLE ALPES	Co-encadrant de thèse
Christian ENE Maître de conférences, UNIVERSITE GRENOBLE ALPES	Co-encadrant de thèse

Rapporteurs :

KARINE HEYDEMANN
Maître de conférences HDR, SORBONNE UNIVERSITE
JULIEN SIGNOLES
Ingénieur de recherche, CEA CENTRE DE PARIS-SACLAY

Thèse soutenue publiquement le **28 avril 2023**, devant le jury composé de :

MARIE-LAURE POTET Professeur des Universités, GRENOBLE INP	Directrice de thèse
KARINE HEYDEMANN Maître de conférences HDR, SORBONNE UNIVERSITE	Rapporteure
JULIEN SIGNOLES Ingénieur de recherche, CEA CENTRE DE PARIS-SACLAY	Rapporteur
VINCENT BEROULLE Professeur des Universités, GRENOBLE INP	Président
THOMAS JENSEN Directeur de recherche, INRIA CENTRE RENNES-BRETAGNE ATLANTIQUE	Examinateur

Invités :

FELIOT DAVID
Ingénieur, CEA CENTRE DE GRENOBLE
LAURENT MOUNIER
Maître de conférences, UNIVERSITE GRENOBLE ALPES

Remerciements

C'est la dernière page à rédiger pour ce manuscrit, et s'il ne s'agit probablement pas de la plus difficile à écrire, c'est pour autant loin d'être la plus simple. Je m'excuse par avance pour tous ceux que je vais fatalement oublier.

Tout d'abord, mes sincères remerciements à *Marie-Laure Potet*, ma directrice de thèse, qui m'a accompagné tout au long de ces quatre années de doctorat. Elle a su me guider et m'apporter son expérience du monde académique et son goût pour la recherche. Un grand merci aussi à mes co-encadrants, *Cristian Ene* et tout particulièrement *Laurent Mounier*, qui m'ont aidé et fait profiter de leur expertise dans les domaines de recherche de cette thèse.

Je tiens aussi à remercier *Karine Heydemann* et *Julien Signoles*, mes rapporteurs, qui ont pris le temps de lire soigneusement toutes les pages de ce manuscrit et de m'en faire un retour pertinent et détaillé. Merci tout particulièrement à *Julien* pour m'avoir accompagné depuis le début de la thèse en tant que référent pour le comité de suivi. Je remercie aussi les autres membres du jury, *Thomas Jensen*, ainsi que *Vincent Beroulle*, qui m'a fait l'honneur de présider le jury de thèse. Je souhaiterais aussi remercier *David Féliot*, qui m'a fourni des retours et des suggestions concernant l'outil Lazart et avec qui j'ai pu collaborer pour certaines publications.

Je voudrais aussi exprimer ma gratitude envers les membres permanents du laboratoire VERIMAG. Je n'ai pas eu la chance de tous vous connaître personnellement mais j'ai fait des rencontres enrichissantes et les discussions en salle de repos m'ont souvent été d'une grande aide sur le plan académique, technique et personnel. Je voudrais remercier tout particulièrement *Erwann Jahier* et *Patrick Fulconis*, qui m'ont sorti de situations techniques délicates et m'ont apporté de précieux conseils. Merci aussi aux différents doctorants et stagiaires que j'ai pu rencontrer au cours de ma thèse. Particulièrement mes collègues de bureau, *Vincent Werner* et *Maxime Lesourd*, pour les soirées bières et les discussions autour d'un café, *Guilhem Lacombe* pour son travail sur Lazart et les projets de recherche qui ont été menés par la suite, *Thomas Vigouroux* pour les moments de rire et son soutien sur la période difficile de la rédaction (le saumon fumé viendra un jour). Mes remerciements vont aussi à tous ceux que j'ai pu croiser que de façon occasionnelle, à *Louis Dureuil* et *Maxime Puys* (d'autant plus pour leur travail antérieur sur Lazart qui a été une excellente base pour le développement de l'outil), ainsi qu'à *Soline Ducousso*, *Jean-Baptiste Bréjon* et *Johnathan Salwan* et bien d'autres. Je remercie également mes collègues enseignants pour ce qu'ils m'ont apportés durant la période où j'ai enseigné ainsi que les étudiants des différentes promotions de licence et de master qui m'ont fait découvrir cet aspect de la recherche. Merci aussi aux chercheurs et intervenants de l'Université de Limoges, où j'ai effectué ma licence et mon master, qui ont su me donner goût à l'informatique, la sécurité et la recherche. Ainsi qu'à mes camarades de promotions, pour les moments de rigolades et de sérieux qu'on a pu passer ensemble. Merci à tous ceux du monde académique et industriel que j'ai pu rencontrer au cours de réunions, séminaires, écoles d'été ou colloques, et merci à ceux qui ont réalisé les différents travaux qui m'ont inspiré et sur lesquels je me suis basé pour les recherches présentées dans ce manuscrit.

À ma famille, ma mère *Magali*, mon père *Hervé* et mes deux frères, *Vincent* et *Arnaud*, qui ont été un soutien important pendant les moments difficiles, de doutes et de remise en question. Un remerciement tout particulier à mes amies et amis qui m'ont accompagné et soutenu pendant cette thèse, chacun à sa manière : *Audrey*, *Aurélie*, *Benjamin*, *Clément*, *Coraline*, *Élise*, *Étienne*, *Fabien*, *Florian*, *Hildéric*, *Jean-Baptiste*, *Joris*, *Julia*, *Julie*, *Mallaury*, *Mélisse*, *Morgan*, *Pierre-Olivier*, *Sophie*, les nombreux *Thomas*, *Quentin*, *Victor*, ainsi que tous ceux de chez Origin. Enfin, merci à tous ceux que je n'ai pas pu citer ou que

j'aurais oublié, ce manuscrit n'aurait pas vu le jour sans les conseils et le soutien qui m'ont été apportés au cours de ces quatre années de thèse.

Outils pour l'analyse de code et de contre-mesures pour l'injection de fautes multiples

Résumé : Les attaquants actifs sont capables d'intervenir sur le comportement du programme pendant son exécution. En particulier, les attaques par injection de fautes, qui ont émergées à l'origine en tant que méthode de test contre les fautes matérielles accidentelles, sont un vecteur d'attaque puissant dans lequel l'attaquant peut injecter des *fautes* pendant l'exécution à l'aide de techniques physiques telles que les faisceaux lasers [Roscian 2013, Colombier 2019] ou des glitches de tension [Bar-El 2006] ou de fréquence [Agoyan 2010, Yuce 2018]. Plus récemment, des méthodes d'injection de fautes logicielles sont apparues [Park 2014]. Cette thèse s'intéresse à l'analyse de programmes dans le contexte des attaques en fautes, et plus particulièrement en fautes multiples, qui implique que l'attaquant est capable d'effectuer des attaques combinant plusieurs fautes. Ces attaques combinées complexifient la tâche des outils d'évaluation automatique de robustesse en raison de l'explosion combinatoire des chemins d'exécution due aux fautes.

Ce manuscrit présente les différentes contributions de cette thèse. En premier lieu, l'outil d'analyse de robustesse Lazart a été repris et développé au cours de cette thèse et vise à aider la recherche d'attaques dans un programme dans le contexte de fautes multiples. L'évaluation des protections contre ce type d'attaques a aussi été un sujet important de cette thèse et des contributions sont présentées pour aider au placement de contre-mesures logicielles. Enfin, les outils de placement utilisés dans la littérature se basent souvent sur une approche essais / erreurs qui n'est pas adaptée au contexte d'attaques multiples et s'intéressent rarement à montrer que les protections sont effectivement utiles. Une méthodologie d'optimisation de programmes protégés a été développée, visant à déterminer quelles portions des protections peuvent être retirées, tout en maintenant le même niveau de sécurité.

Mots clés : Analyse de code, Injection de fautes, Fautes multiples, Contre-mesures logicielles, Évaluation de contre-mesures, Lazart

Tools for code and countermeasures analysis against multiple faults attacks

Abstract : Active attackers are able to modify the behavior of the program during its execution. In particular, fault injection attacks, which originally emerged as a method of testing against accidental hardware faults, are a powerful attack vector in which the attacker can inject faults during execution using physical techniques such as laser beams [Roscian 2013, Colombier 2019], or voltage [Bar-El 2006] or frequency glitches [Agoyan 2010, Yuce 2018]. More recently, software fault injection methods have been proposed [Park 2014].

This thesis focuses on analysing programs in the context of fault injection attacks, and more particularly multiple faults, which implies that the attacker is able to combine several faults to complete an attack. Combined attacks make more difficult the evaluation of the robustness of programs by tools because of the combinatorial explosion of the execution paths due to faults.

This manuscript presents the different contributions of this thesis. First, the robustness analysis tool Lazart has been developped during this thesis, aiming at helping the search of attacks on software in a multi-fault contex. The evaluation of protections against this type of attacks has also been an important topic of this thesis and contribution are presented to assist the placement of software countermeasures. Finally, the placement tools used in the literature are often based on a trial-and-error approach that is not adapted to multiple fault contexts, and they rarely address the verification that these protections are effectively useful in the program. A methodology to optimise the placementof countermeasures has been developed, able to determine which portions of the protections in a program can be removed, while maintaining the same level of security.

Keywords : Code analysis, Fault injection, Multiple faults, Software countermeasures, Countermeasure evaluation, Lazart

Table des matières

Table des figures

Liste des tableaux

Table des Listings

Chapitre 1

Contexte

Table des Matières

1.1 Garantir la sécurité d'un programme ou d'un système

La sûreté et la sécurité des programmes et des systèmes sont des enjeux majeurs de nos jours.

La *sûreté* se réfère à l'ensemble des risques dont la cause est accidentelle. La *sécurité* concerne quant à elle les actes de malveillance. Si on prend pour analogie un aéroport, la panne d'un avion correspond à un risque de l'ordre de la sûreté tandis qu'un attentat relève de la sécurité. Une *attaque* correspond à une tentative d'un adversaire (appelé *attaquant*) d'obtenir un avantage sur un système. Cet avantage peut se traduire par une élévation de privilèges [Timmers 2016] ou l'accès à des données sensibles [Biham 1997] par exemple, mais dans un cadre général il peut s'agir de tout comportement qui n'a pas été prévu pour le système. Par *système* nous entendrons de manière générique un appareil informatique, un programme, un réseau de sous-systèmes et plus généralement, toute entité traitant de l'information répondant à des besoins de sûreté et susceptible d'être attaquée.

Des ordinateurs aux téléphones modernes, en passant par les appareils connectés ou encore les cartes à puces, le nombre de systèmes et de technologies disponibles continue de croître [Juniper 2018]. La surface d'attaque globale ne cesse donc de grandir, et la diversité des attaques qui doivent être prises en compte lors de la mise en place d'un système rend le processus de développement plus complexe.

Pour parer à ces menaces, des protections sont mises en oeuvre de manière à détecter ou bloquer les attaques. Ces protections peuvent prendre la forme de transformations du programme source, par exemple en modifiant l'algorithme [Aumüller 2002] ou en rajoutant du code visant à détecter et/ou empêcher les attaques [Reis 2005, Lalande 2014]. Les protections peuvent aussi être appliquées au niveau de la plateforme, comme par exemple la randomisation de l'espace mémoire par le système d'exploitation de manière à rendre l'exploitation des attaques plus difficiles [Shacham 2004]. Ou encore, au niveau matériel, en ajoutant des protections physiques dans le composant de manière à limiter la lecture des données stockées [Boneh 1997].

De nouvelles attaques sont découvertes et mènent à la recherche de nouvelles protections qui sont par la suite attaquées à leur tour. Ainsi, les attaquants se diversifient et un système

EAL	Description
EAL1	Testé fonctionnellement
EAL2	Testé structurellement
EAL3	Méthodiquement testé et vérifié
EAL4	Méthodiquement conçu, testé et revu
EAL5	Semi-formellement conçu et testé
EAL6	Conception semi-formellement vérifiée et testée
EAL7	Conception formellement vérifiée et testée

TABLE 1.1 – Les différents niveaux de certification des Critères Communs [Dureuil 2016a]

doit faire face à des menaces variées.

Problématique 1.1. *Comment faciliter le processus de développement d'un logiciel afin d'aider le développeur dans le choix et la mise en place des protections ?*

Plus encore, une nouvelle attaque peut être découverte après l'analyse de la sécurité d'un programme et ainsi mettre en lumière de nouvelles vulnérabilités. Les processus d'évaluation et de protection d'un système doivent donc être réexaminés à intervalles réguliers de manière à prendre en compte les nouvelles menaces.

Bien entendu, le niveau de sécurité dépend grandement du contexte d'utilisation d'un système. La confidentialité des communications d'un satellite militaire doit répondre à un niveau de sécurité plus élevé que celle d'une boite au lettre d'un particulier par exemple. A l'inverse, une serrure pour une résidence est une protection contre les tentatives d'intrusion dont la nécessité est discutable pour un satellite. Il est nécessaire d'évaluer le risque d'une attaque en fonction du niveau de sécurité qu'on souhaite atteindre et en fonction du système à protéger. La balance bénéfice/coût de l'attaquant est à prendre en compte par le défenseur pour évaluer les menaces du système.

Pour permettre de certifier la sûreté et la sécurité d'un système, des méthodologies ont été développées. La certification Critères Communs (CC) [ISO 2017] est un processus de certification standardisé par la norme International Organization for Standardization (ISO) 15408 permettant à des organismes spécialisés de garantir un certain niveau de sécurité sur un système. Les CC proposent un ensemble de procédures permettant d'obtenir un niveau d'assurance sur la sécurité du système en question. Ainsi, il est possible de viser un type de certification en fonction du niveau de sécurité voulu. La figure 1.1 présente les différents niveaux d'assurance des CC, chacun définissant un seuil de mise en place de moyens humains, d'expertise et de technologie par le fournisseur du produit et l'organisme de certification et des critères en fonction des problèmes découverts pour l'obtention ou non du certificat.

De la même manière, les audits de sécurité sont effectués pour les systèmes d'information et mettent en place des scénarios d'attaque variés de manière à s'assurer de la sécurité du système. Des tests d'intrusion (les *pentests*) sont mis en oeuvre afin de simuler différents types d'attaquant et de vérifier la réaction du système. Différents modèles d'attaquant sont considérés, en allant du test en boite noire où le système est inconnu de l'attaquant et qui se place à l'extérieur de celui-ci, au test en boite blanche où l'attaquant est modélisé comme connaissant les détails internes du système ou comme ayant un accès privilégié dans le système.

Problématique 1.2. *Comment faciliter le processus d'évaluation par l'expert d'un programme et de ses protections à l'aide de méthodologies et d'outils ?*

La sécurité a un coût, à la fois en termes de moyens humains et d'expertise mais également parce que les protections peuvent complexifier le système, ou altérer les performances (rapidité d'exécution inférieure, taille du code plus grande...). De plus, d'autres problématiques industrielles comme des délais de production ou des limitation matérielles viennent se combiner à celle de la sécurité. Par exemple, les processeurs modernes utilisent une optimisation appelée l'exécution spéculative qui vise à exécuter une branche d'un programme avant que la condition de branchement soit évaluée. Si la mauvaise branche a été choisie, la pipeline est déroulée pour revenir à son état précédent pour exécuter la bonne branche. Cette technique permet de fournir une performance accrue pour le processeur. Néanmoins, l'attaque Spectre, découverte en 2019 [Kocher 2019], a démontré qu'il était possible d'exploiter les effets de bord de cette optimisation de manière à retrouver des informations secrètes qui ne sont normalement pas accessibles. Un compromis est souvent nécessaire entre les différentes problématiques (sûreté, sécurité, performance, accessibilité etc.) liées au développement d'un système.

Chaque couche ou noeud d'un système pouvant être menacé, une attaque peut être composée de plusieurs attaques visant des parties différentes du système. Par exemple, l'attaque *Rowhammer* [Kim 2014, Park 2014], permet à un attaquant de tirer profit d'effets de bord dans le fonctionnement des cellules mémoires Dynamic Random-Access Memory (DRAM) pour augmenter le niveau de privilèges sur une machine sans nécessité d'accès physique. Ce fut aussi le cas lors du challenge *Wookey* [SSTIC 2020] où un chemin d'attaque hybride a été détecté dans lequel la protection contre le dépassement de tampon implémentée dans l'interface de carte à puce (ISO 7816) est contournée par une attaque physique. La combinaison des deux attaques permet d'obtenir une élévation de privilège.

Problématique 1.3. *Comment évaluer un programme et ses protections dans un contexte d'attaques multiples ?*

Cette thèse s'inscrit dans ce contexte et vise à proposer des solutions pour la protection et l'évaluation d'un programme et de ses protections. Ce chapitre présente des problématiques générales liées à l'analyse de programmes et de protections ainsi que les contributions apportées dans cette thèse.

L'exemple verify_pin

Le programme présenté dans le listing 1.1 est une version naïve d'un processus de vérification de code Personal Identification Number (PIN) en langage C. Ce programme servira d'exemple pour présenter les enjeux et les problématiques de la protection et l'analyse de programmes.

```
1  // PIN secret.
2  static uint8_t card_pin[5] = {'0', '1', '2', '3', '\0'};
3  static int try_counter = 3;
4
5  bool compare(uint8_t* a1, uint8_t* a2)
6  {
7      int i = 0;
8      while(*a1 != '\0') {
9          if(*a1 != *a2)
10             return false;
11
12         ++a1; ++a2;
13     }
14
15     return true;
16 }
```

```
17
18 bool verify_pin(uint8_t* user_pin) {
19     if(try_counter > 0) {
20         if(compare(card_pin, user_pin)) {
21             try_counter = 3;
22             return true;
23         } else {
24             try_counter--;
25             return false;
26         }
27     }
28
29     return false;
30 }
```

Listing 1.1 – Programme verify_pin « naif »

La figure 1.1 est un schéma d'utilisation fonctionnel de ce programme. L'authentification nécessite que le compteur d'essais soit strictement positif et que le PIN d'entrée soit identique à celui de la carte. Le programme `verify_pin` est décomposé en deux : la fonction `verify_pin` prend en argument le PIN utilisateur et appelle la fonction `compare` qui se charge de comparer les deux chaînes correspondant aux PINs.

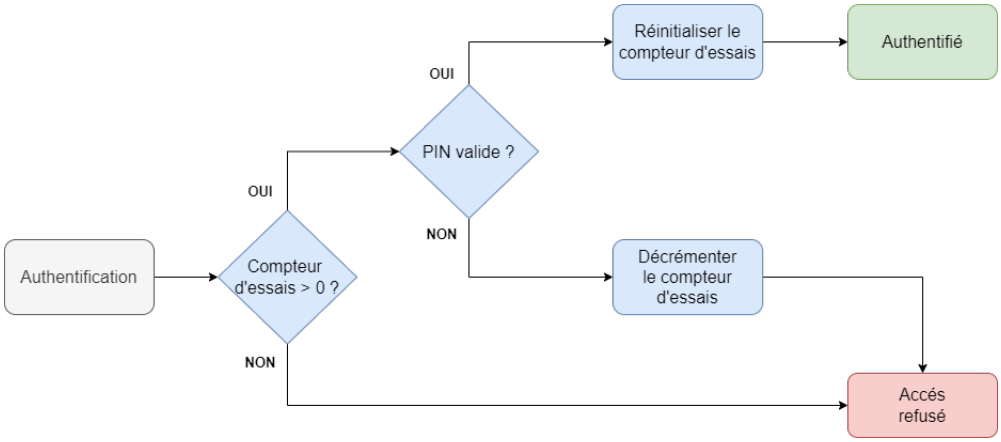

FIGURE 1.1 – Schéma fonctionnel du programme *verify_pin*

Dans la fonction principale, un compteur d'essai est vérifié de manière à limiter le nombre d'erreurs d'entrées successives autorisées. Le compteur est décrémenté en cas d'échec et remis à la valeur 3 lorsque l'authentification est un succès. Le PIN secret `card_pin` est modélisé par un tableau statique pour cet exemple (ce qui n'est pas le cas dans la réalité).

La fonction `compare` effectue une comparaison de deux chaînes en bouclant sur chaque caractère jusqu'à atteindre un caractère terminant par le caractère nul '\0'. Si deux caractères sont différents, la boucle se termine prématurément en retournant *faux*. Cela correspond à ce qui est fait par des fonctions C standards telles que `strcmp`. Les deux fonctions retournent des valeurs booléennes *vrai* et *faux* respectivement représentées par les entiers 1 et 0.

Ce programme illustre une problématique d'authentification qui est présente dans un grand nombre d'objets sécurisés, par exemple les cartes Subscriber Identification Module (SIM) des téléphones ou les cartes à puces.

1.2 Modèles d'attaquant et objectif d'attaque

Lorsqu'on s'intéresse à la sécurité d'un programme, il est nécessaire de définir l'adversaire contre lequel on souhaite se protéger, ce qui se traduit par le *modèle d'attaquant*.

On aimerait dans l'idéal pouvoir rester le plus général possible, mais en pratique les outils et méthodes d'analyse restreignent le contexte dans lequel le programme est étudié. Cela est fait d'une part, pour maintenir des temps de calcul réalistes et d'autre part, parce qu'il n'y a pas de limite théorique à la puissance d'un attaquant, qui va dépendre du contexte et des moyens mis en oeuvre. Si un programme est sécurisé face à un modèle d'attaquant précis, rien n'empêche qu'un autre type d'attaque soit découvert par la suite et remette en question la sécurité du programme.

Problématique 1.4. *Comment définir et modéliser un attaquant dans le contexte d'une analyse de sécurité ?*

Le *modèle d'attaquant* peut être vu comme un ensemble contenant : l'objectif d'attaque et le pouvoir de l'attaquant, comme présenté dans la figure 1.2.

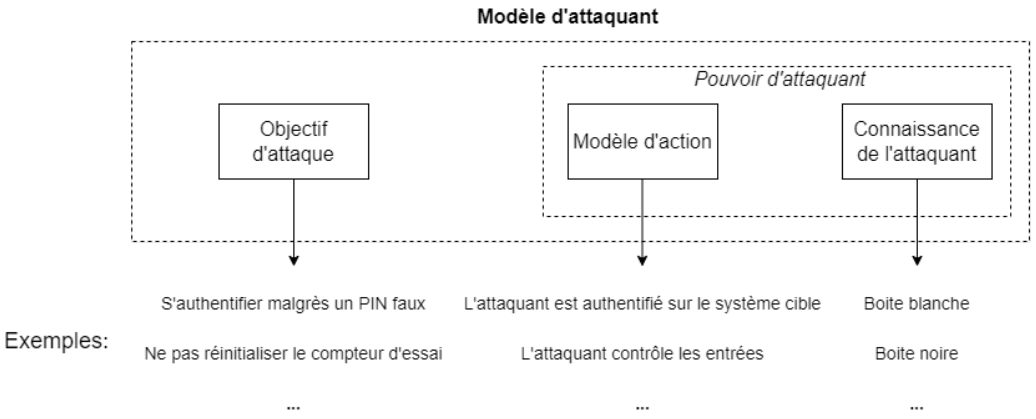

FIGURE 1.2 – Modèle d'attaquant

On appelle *objectif d'attaque* la propriété de sécurité visée par un adversaire. Pour l'exemple *verify_pin*, un objectif d'attaque naturel consiste à s'authentifier sans connaître le code de la carte. Un second objectif peut être de ne pas décrémenter le compteur d'essais lors d'un échec d'authentification.

On appelle *modèle d'action*, l'ensemble des actions qu'un attaquant est capable d'effectuer sur le système cible. La *surface d'attaque* correspond à l'ensemble des points du système sur lesquels l'attaquant effectue ses actions et est donc directement liée au modèle d'action considéré.

Une *vulnérabilité* est une faiblesse, dans une implémentation ou un système, pouvant être exploitée par un adversaire. Une vulnérabilité peut être un système mal configuré, non mis à jour, une erreur dans un logiciel, l'usage d'algorithme cryptographique non sûr ou même d'ordre organisationnel comme la présence d'un adversaire au sein du personnel de l'entreprise par exemple. Une vulnérabilité est intrinsèquement liée à un modèle d'attaquant, par exemple un programme peut être vulnérable aux dépassements de tampon ou à un défaut d'alimentation.

On parle d'*exploitation* pour désigner un programme ou une méthode permettant d'exploiter une vulnérabilité dans un système (en obtenant une élévation de privilège par

exemple). La présence d'une vulnérabilité n'implique pas forcément qu'une exploitation soit possible dans l'immédiat. Par exemple, l'attaque *Rowhammer* présentée précédemment a été d'abord théorisée [Kim 2014] puis exploitée plus tard [Seaborn 2015, Gruss 2016]. Une vulnérabilité peut potentiellement être exploitée bien après sa découverte. Dans l'exemple *verify_pin*, une vulnérabilité sur la ré-initialisation du compteur de boucle en cas d'échec pourrait être exploitée pour obtenir un accès à la carte à l'aide d'une attaque par force brute.

Enfin, le modèle d'attaquant inclus aussi la connaissance qu'un adversaire a sur sa cible, la *connaissance de l'attaquant*. C'est ce que représente la diversité des tests en certification par exemple avec des attaquants agissant sur un système en boite noire ou en ayant un accès et une connaissance plus ou moins étendue sur le système. La Join Interpretation Library (JIL) [JIL 2020] fait la distinction entre différents niveaux d'expertise reflétant la capacité d'un attaquant à utiliser des outils d'audit ou à créer de nouvelles attaques. Cette modélisation de la connaissance de l'adversaire sur le système permet aussi d'évaluer le risque que présente une vulnérabilité sur le système lors des processus d'audit. Un attaquant ayant un accès et des connaissances privilégiés sur le système est plus dangereux mais moins probable.

En allant d'un attaquant ne contrôlant que les entrées du programme à un adversaire infiltré en tant d'administrateur sur la machine cible, le modèle d'attaquant peut radicalement varier en fonction du cadre d'usage du programme et des techniques d'attaques connues. Les exploitations peuvent mettre en oeuvre plusieurs types d'attaques, comme dans l'exemple *Wookey* évoqué précédemment, et demande donc de prendre en compte des objectifs d'attaque variés.

Dans le cadre de l'analyse logicielle, les outils et les techniques d'analyses doivent faire face à la diversité des modèles d'attaquants, bien que actuellement ils soient souvent spécifiques à des classes d'attaques particulières.

Problématique 1.5. *Comment faciliter l'analyse et la protection de programme pour des modèles d'attaquants qui évoluent et se complexifient ?*

1.3 Analyse de programmes

Le problème de l'arrêt énoncé par Alan Turing [Turing 1937] a montré qu'il n'existe pas de méthode d'analyse automatique permettant de savoir si un programme termine ou non. Lorsqu'on s'intéresse à la vérification de propriétés non triviales sur un programme, on se confronte au problème de l'indécidabilité [Bradley 2006]. Toutefois, il existe des méthodes pour étudier le comportement d'un programme malgré cette limitation.

La fonction `foo` présentée dans le listing 1.2 prend en entrée les variables a, b et c et calcule $a * b * c$. Même dans un cas simple comme celui-ci, il n'est pas trivial de s'assurer que la fonction se comporte toujours correctement.

```
1 int foo(int a, int b, int c) {
2     int total = 0;
3     for(int i = 0; i < c; ++i)
4         total += a * b;
5     return total;
6 }
```

Listing 1.2 – Fonction foo

La fonction *foo* possède un nombre fini d'*exécutions* différentes possibles. Les entrées sont constituées seulement des trois variables a, b et c, mais il reste difficile de tester par force brute toutes les combinaisons d'entrées (2^{96} possibilités dans le cas d'entiers codés

sur 32 bits). Plus encore, certains programmes peuvent potentiellement avoir un nombre d'exécutions infinies. Le système peut attendre des entrées utilisateurs ou un évènement réseau sans limite finie, ce qui rend impossible l'exploration exhaustive des exécutions dans le cas de boucles ou de cycles infinis.

Même si on pouvait effectuer une analyse exhaustive des exécutions, la question de la modélisation du système est aussi une problématique majeure. En effet, si on veut par exemple s'intéresser aux effets d'un dépassement de capacité du type entier dans la fonction `foo`, il faut prendre en compte l'encodage utilisé et les spécificités des architectures cibles. Pour pallier ce problème, il est possible d'effectuer ces tests sur une véritable machine ce qui apporte des garanties concernant la précision de l'analyse mais nécessite la mise en place d'un banc de test physique et rend l'analyse spécifique au système de test choisi [Faurax 2009].

De nombreuses techniques d'analyses ont été développées pour la vérification de programmes, que ce soit dans le domaine de la sécurité ou de la sûreté. Les méthodes d'analyses peuvent être classées en quatre catégories, présentées dans la figure 1.3, en fonction de leur approximation de l'ensemble des exécutions du programme étudié.

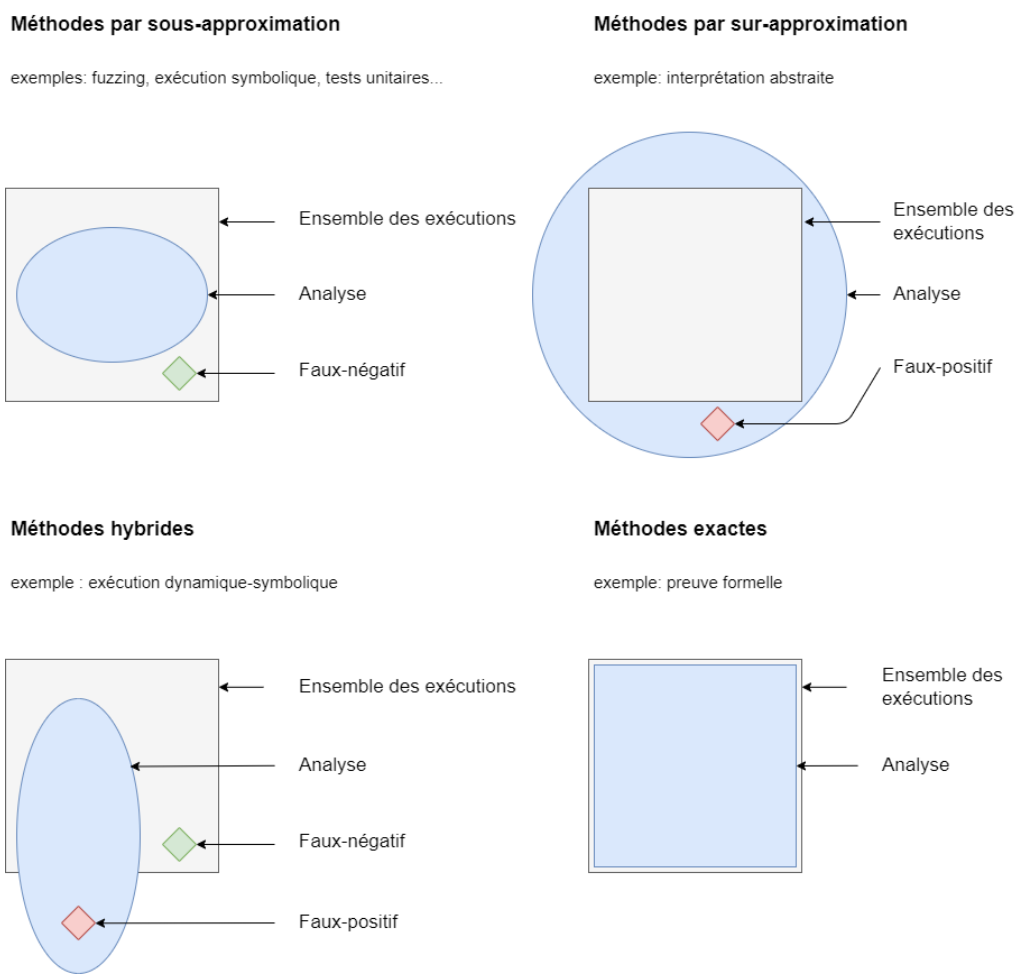

FIGURE 1.3 – Méthodes d'analyses statiques

Les *méthodes par sous-approximations* étudient un sous-ensemble des exécutions possibles d'un programme. L'exemple naturel est le test. Les tests unitaires sont très utilisés en développement logiciel et consistent à tester indépendamment des fonctions et des modules d'un programme en se concentrant souvent sur les cas qui sont à priori susceptibles d'entraîner des erreurs (telles que les bornes du type des entrées ou des valeurs interdites par la spécification par exemple). Le fuzzing [Manès 2019] est une autre méthode de génération de test où les entrées sont choisies aléatoirement (ou semi-aléatoirement à l'aide d'heuristiques). L'exécution symbolique [King 1976] (qui sera détaillé dans la section 3.1.2) est aussi une méthode de génération de tests qui effectue une sous-approximation de l'ensemble des exécutions.

A l'inverse, les *méthodes par sur-approximation* produisent des faux-positifs puisqu'elles s'intéressent à un sur-ensemble des exécutions possibles. L'interprétation abstraite [Cousot 1977, Cousot 2014] est un exemple de méthode par sur-approximation. Le plugin Evolved Value Analysis (EVA) de Frama-C [CEA 2008a] utilise l'interprétation abstraite au niveau C. C'est aussi le cas des analyses faites par les compilateurs pour savoir si certaines transformations ou optimisations sur un programme peuvent être effectuées sans changer la sémantique d'un programme.

Les *méthodes exactes* permettent de caractériser les exécutions d'un programme de manière exacte. La preuve formelle par exemple, consiste à prouver formellement qu'un programme est correct, à l'aide de théories mathématiques. Cela étant, ces techniques ne sont pas entièrement automatiques puisque le problème est indécidable en général. Les assistants de preuves tels que Coq [INRIA 1989], Isabelle [Nipkow 2002] ou encore le plugin WP de Frama-C [CEA 2008b] assistent l'utilisateur qui peut être amené à effectuer une partie du travail d'analyse.

Certaines méthodes présentent à la fois des faux-positifs et des faux-négatifs. L'exécution concolique [Baldoni 2018] est un exemple de *méthode hybride*. Là encore, de nombreux outils existent comme DART [Godefroid 2005], KLEE [Cadar 2008a] ou encore Angr [Shoshitaishvili 2016] pour n'en citer que quelques uns. Plus généralement, les méthodes combinant différentes techniques d'analyse tendent à se classer parmi les méthodes hybrides. Le typage dans certains langages de programmation peut aussi entrer dans cette catégorie.

Cette classification n'est pas stricte et certaines techniques se rangent différemment en fonction du contexte. Le model-checking par exemple considère le programme comme un automate à état finis et permet de vérifier des propriétés sur ce modèle. En fonction de la manière dont est défini le modèle, l'analyse pourra appartenir à différentes classes.

Ces méthodes d'analyse disposent chacune d'avantages et d'inconvénients, que ce soit en termes de précisions, de complétude ou de temps d'exécution. Ces caractéristiques dépendent aussi du niveau de représentation auquel l'analyse se situe. Les processus d'évaluation de programmes dans le cadre de la sécurité incluent le plus souvent plusieurs méthodes.

1.4 Protéger un programme

De manière à protéger un système contre des attaquants, des protections sont mises en oeuvre. Ces protections, qu'on appellera aussi *contre-mesures*, visent à détecter ou prévenir les attaques. Celles-ci peuvent être mises en place à différents niveaux d'un système : sur le code source, sur le code machine, au niveau de la plateforme (machine virtuelle, système d'exploitation) ou encore au niveau physique. Dans le cas de protections ajoutées au niveau du code source d'un programme, on parle de protections logicielles.

Le ver de Morris [Morris 1979], considéré comme le premier « ver » informatique, ex-

ploitait une vulnérabilité dans l'utilitaire *finger* à l'aide d'un dépassement de tampon. Les attaques par dépassement de tampon constituent une classe d'attaques communément étudiée. Il s'agit de dépasser l'espace mémoire prévu (pour une chaîne de caractères ou un tableau par exemple) de manière à accéder à des segments de mémoire non prévus initialement. Cela peut par exemple permettre d'écraser l'adresse de retour de manière à rediriger le flot de contrôle vers un code malicieux, appelé *shellcode*.

Le programme `verify_pin` présenté précédemment souffre d'une vulnérabilité de ce type, notamment à cause de l'utilisation d'un caractère nul pour la fin de chaîne. Le listing 1.3 présente une variante de la fonction `compare` dans laquelle la taille est passée en paramètre ce qui constitue une première protection contre les attaques par dépassement de tampon. Les fonctions C `strcmp` et `memcmp` ont elles aussi vu naître leurs homologues `strcmp_s` et `memcmp_s`, prenant en paramètre la taille des chaînes à comparer pour s'assurer que la taille des tableaux n'est pas dépassée (si tant est que la bonne taille est passée en argument).

```
1  bool compare(uint8_t* a1, uint8_t* a2, size_t size)
2  {
3      for(size_t i = 0; i < size; i++) {
4          if(a1[i] != a2[i]) {
5              return false;
6          }
7      }
8
9      return true;
10 }
```

Listing 1.3 – verify_pin avec la taille passée en paramètre

Les canaris constituent une autre solution de protection logicielle contre les attaques par dépassement de tampon. Il s'agit de valeurs particulières d'octets qui sont placées entre les tableaux de données (ou le plus souvent entre les parties sensibles, à savoir les valeurs de retour sur la pile). Il est alors possible de vérifier l'intégrité du canari pour détecter certains dépassements de mémoire et prendre les mesures nécessaires (invalidation de la donnée corrompue, arrêt du programme, signalement...). Des variantes existent mais cette méthode, qui est implémentée dans tous les compilateurs actuels, implique une surcharge de la mémoire pour les canaris et des mécanismes de vérification à l'exécution.

Des techniques comme l'utilisation d'une pile non exécutable visent à empêcher l'exécution d'un shellcode. Néanmoins, l'attaquant peut se servir de *gadget*, c'est-à-dire des portions du programme cible (ou d'une bibliothèque) de tailles généralement assez courtes. Ces portions étant déjà dans la section instruction de la mémoire, il sera possible de les exécuter. L'attaquant va alors écraser les adresses de retour de manière à exécuter plusieurs gadgets à la suite et obtenir le comportement souhaité, ce qu'on appelle Return Oriented Programming (ROP) [Roemer 2012].

L'Adress Space Randomization Layout (ASRL), effectuée au niveau de la plateforme, repose sur la randomisation de l'espace mémoire de manière à rendre plus difficile pour un attaquant de trouver les canaris ou des gadgets par exemple. Dans le même registre, les système d'exploitation peuvent mettre en place un réordonancement aléatoire des adresses mémoires manipulées.

Ces contre-mesures visent à endiguer ou à limiter l'impact des attaques par débordement de tampon. Celles-ci peuvent prendre la forme de transformations du programme source (en ajoutant un argument de taille comme dans l'exemple précédent de *compare*), être appliquées par un outil automatique (par exemple au moment de la compilation comme l'ajout de canaris), être des protections assurées par l'environnement (comme l'ASRL qui est effectuée par le système d'exploitation, la mise en place de machines virtuelles ou de

système de détection d'intrusion (IDS)) ou encore être mises en place au niveau physique (en chiffrant la mémoire de la machine par exemple).

Un autre exemple de protection concerne les attaques par canaux auxiliaires. Celles-ci visent à récupérer des informations à partir de données physiques telles que la consommation d'un composant ou le temps d'exécution. Pour la fonction `compare` précédente, un attaquant pourrait mesurer le temps d'exécution de la boucle de manière à savoir quel chiffre du code PIN était invalide et ainsi simplifier la recherche du secret [1].

On peut se protéger de ce type d'attaque en transformant la boucle de la manière à la rendre en *temps constant*, comme présenté dans le listing 1.4 où la boucle n'est pas terminée prématurément en cas d'échec et où la condition est remplacée par une opération bit à bit. Un large éventail de protections logicielles et matérielles a été développé pour ce type d'attaque également [Witteman 2008, Veyrat-Charvillon 2012].

```
1 bool compare(uint8_t* a1, uint8_t* a2, uint8_t size)
2 {
3     bool result = true;
4     for(size_t i = 0; i < size; i++)
5         result |= a[i] ^ b[i];
6
7     return result;
8 }
```

Listing 1.4 – verify_pin en temps constant

Problématique 1.6. *Étant donné un modèle d'attaquant et un programme à protéger, comment aider au choix des contre-mesures ?*

La recherche de la meilleure contre-mesure implique la nécessité d'être capable d'analyser la robustesse d'un programme protégé ou de comparer différents programmes protégés. Des méthodes et des outils doivent donc être mis en place pour aider à l'analyse des contre-mesures.

1.5 Analyser les contre-mesures

L'analyse de contre-mesures a plusieurs objectifs : en amont du processus de développement, l'aide au développeur pour la sélection et la mise en place des protections, et en aval, l'aide à l'auditeur pour la recherche de vulnérabilités et la comparaison de différents moyens de protection. Cette analyse se fait en fonction d'un modèle d'attaquant défini.

En plus de renforcer la sécurité, d'autres paramètres concernant les contre-mesures sont à étudier. Il peut s'agir de métriques de performances (vitesse d'exécution, taille du programme, consommation en mémoire / courant), de contraintes matérielles ou encore de préoccupations telles que la facilité de déploiement ou d'automatisation (par exemple si la contre-mesure nécessite des primitives cryptographiques qui ne sont pas disponibles sur toutes les plateformes). Le plus souvent il est nécessaire de faire un compromis entre ces différents paramètres. Le problème de la performance est d'autant plus présent dans le monde de l'Internet des objets (Internet of Things (IoT)) ou des composants sécurisés tels que les cartes à puces. En effet, la puissance du matériel est limité mais ces appareils contiennent potentiellement des données sensibles comme des clefs de chiffrement ou des certificats par exemple.

L'analyse des contre-mesures d'un programme peut être faite de différentes manières. L'utilisation d'ensembles de cas de tests, appelés *benchmark*, permet aux concepteurs de

1. Un PIN à 4 chiffre comporte 10^4 combinaisons, si un attaquant peut savoir quel chiffre est incorrect il devient alors capable de casser le PIN en $10 * 4$ essais au maximum.

contre-mesures de comparer leurs résultats à ceux obtenus par des contre-mesures existantes. L'évaluation de la protection dépend alors de la représentativité du benchmark choisi, ce qui peut être difficile à estimer. La comparaison des versions protégées d'un programme peut alors être réalisée en comparant des métriques d'analyse. Il peut s'agir, entre autres, du nombre d'attaques trouvées ou du nombre d'attaques pondéré par la taille du programme. La *vérification formelle* peut aussi être utilisée pour effectuer une preuve que le programme protégé n'est pas vulnérable en cas d'attaque [Rauzy 2014, Moro 2014a]. Les contre-mesures sont généralement pensées pour protéger le programme contre un modèle précis et la question de savoir si toutes les protections sont effectivement utiles est plus rarement considérée. De plus, les contre-mesures étant parfois placée automatiquement, par la chaîne de compilation par exemple, la nécessité d'outils capables d'évaluer automatiquement la pertinence des protections est d'autant plus importante.

Problématique 1.7. *Comment s'assurer que les protections ajoutées à un programme sont effectivement utiles dans le modèle d'attaquant considéré ? Peut-on en enlever ?*

Les modèles d'attaquants puissants complexifient d'autant plus l'analyse que le nombre d'attaques et de chemins d'exécution à étudier peuvent devenir importants. Les attaques composées de plusieurs attaques, appelées *attaques multiples*, comme c'est le cas dans l'exemple Wookey [ANSSI 2020] cité précédemment, permettent d'attaquer les contre-mesures elles-mêmes : la première attaque neutralise la contre-mesure et une seconde exploite cette absence de protection.

Problématique 1.8. *Comment faciliter la mise en place de contre-mesures dans un contexte d'attaques multiples ?*

1.6 Contributions et organisation du manuscrit

1.6.1 Contributions

Cette thèse s'intéresse aux attaques multiples dans lesquelles l'attaquant est actif, et peut donc intervenir sur l'exécution du programme. Les attaques par injection de fautes, présentées plus en détail dans le chapitre 2, sont un type d'attaques où l'attaquant peut intervenir sur l'état du programme pendant l'exécution. L'attaquant peut par exemple modifier les instructions du programme exécuté, sauter d'un point du programme à l'autre ou encore modifier les données manipulées par le programme, en fonction du modèle d'attaquant considéré. Ces attaques sont de plus en plus d'actualité pour différents facteurs. Premièrement, la baisse du coût du matériel pour réaliser ces attaques a permis de les rendre plus grand public [Breier 2019]. De plus, ces attaques impliquent une grande puissance de l'attaquant qui est actif pendant l'exécution du programme.

L'extension de l'outil *Lazart*, développé depuis 2014 au sein du laboratoire *VERIMAG*, a été réalisée tout au long de cette thèse. Lazart est un outil d'analyse de code travaillant au niveau de la représentation intermédiaire Low-Level Virtual Machine (LLVM) et qui utilise l'exécution concolique, à l'aide de l'outil KLEE, afin de trouver des chemins d'attaques en fautes multiples. L'outil Lazart est destiné à aider au développement de programme robustes (problématique 1.1) et à l'évaluation de ces programmes (problématique 1.2).

L'outil a été porté sur des versions plus récentes de LLVM et de KLEE et a été enrichi avec notamment de nouveaux modèles d'attaquants, de nouvelles analyses et une nouvelle interface en fonction des besoins énoncés par des utilisateurs issus de la recherche et de l'industrie. Les problématiques de définition de modèles d'attaquant (problématiques 1.4

et 1.5) et d'analyse dans le cadre d'attaques multiples (problématiques 1.3) sont également adressées, en partie, par l'outil Lazart.

L'étude et l'analyse de contre-mesures a été le sujet principal de cette thèse. Cette thèse s'est orientée vers l'étude de l'application automatique de contre-mesures contre les fautes multiples et propose des solutions pour la sélection des portions d'un programme à protéger en fonction d'un scénario d'attaque et d'un ensemble de contre-mesures locales (problématiques 1.6). Enfin, une méthodologie d'analyse de contre-mesure a été proposée, permettant de détecter les protections qui ne sont pas utiles pour un modèle d'attaquant donné dans le cadre d'attaques multiples (problématiques 1.7 et 1.8). Cette méthodologie a été implémentée au sein de l'outil Lazart.

1.6.2 Organisation du manuscrit

Le chapitre 2 présente les pré-requis de cette thèse, notamment ce qui concerne les attaques physiques qui constituent le domaine d'étude principal de nos expérimentations. Il s'agira de présenter les attaques en fautes et proposer un aperçu des modèles de faute et outils d'analyse existants.

Le chapitre 3 traite le fonctionnement général et l'architecture de l'outil Lazart, la plateforme LLVM et l'exécution symbolique avec KLEE. Il se concentre sur l'utilisation de l'outil, la description d'une analyse et l'exploitation des résultats. Il présente les résultats obtenus sur un ensemble de programme et discute de l'aspect méthodologique notamment en ce qui concerne l'utilisation de Lazart dans une chaîne d'outils. Le chapitre 4 se concentre sur les choix d'implémentation des différents modules de l'outil.

Le chapitre 5 aborde les problématiques de l'analyse de contre-mesures logicielles et présente diverses approches qui ont été proposées dans la littérature. Il propose aussi différents algorithmes visant le placement de contre-mesures afin de produire un programme protégé résistant à n attaques, à partir d'un programme d'entrée et d'un modèle d'attaquant. Le chapitre 6 présente une méthodologie permettant de déterminer les portions du code protégé pouvant être retirées sans introduire de nouvelles attaques, en fonction d'un scénario d'attaque fourni par l'utilisateur. Ce chapitre décrit cette méthodologie, présente son implémentation et discute des résultats obtenus sur un ensemble de programmes d'exemples.

Enfin, le chapitre 7 présentera des perspectives de recherche pouvant faire suite aux contributions de cette thèse.

Chapitre 2

Analyse de robustesse dans le cadre d'injection de fautes multiples

Table des Matières

2.1 Attaques physiques et injection de fautes

2.1.1 Fautes matérielles aléatoires

En 1954, Isaac Azimov, un écrivain américain réputé pour ses écrits en science-fiction, publie le livre « Les cavernes de l'acier » (dans son titre original « The Caves of Steel » [Asimov 1954]), dans lequel un robot se voit désactivé par le rayonnement d'une particule alpha [1].

Vingt ans plus tard, en 1975, Binder et al. [Binder 1975] indiquent que les anomalies électroniques observées sur les satellites de télécommunications pourraient être liées aux rayonnements ionisants issus du soleil provoquant le déclenchement de bascules. En 1978, May et al. [May 1978] découvrent que des erreurs de valeurs dans la mémoire dynamique DRAM, appelées *soft errors*, peuvent être causées par le rayonnement de particules alpha de l'environnement immédiat des cellules mémoires.

La problématique de la tolérance aux fautes (*fault tolerance)* face aux corruptions silencieuses de la mémoire (Silent Data Corruption (SDC)) liées à des phénomènes physiques

1. Les particules alpha sont des rayonnements issus de la radioactivité.

tels que les rayonnements cosmiques [Ziegler 1979], est d'autant plus présente de nos jours en raison de la forte augmentation des tailles de stockages [Charyyev 2019], ou des systèmes hautes performances (High Performance Computing (HPC)) [Di 2016].

Cela a donné lieu à la mise en place d'outils d'analyse de tolérance des programmes aux fautes matérielles [Segall 1988, Kanawati 1992] ainsi que des mécanismes de protection au niveau logiciel ou matériel (comme le déploiement de codes correcteurs d'erreur ou de redondance des données par exemple [Wu 2017]). Les années 90 voient émerger l'injection de fautes [Karlsson 1994, Clark 1995], une technique visant à injecter volontairement des fautes dans un système afin d'observer leurs effets sur un composant. Ces injections sont réalisées en induisant un stress sur le système visé à l'aide de radiations d'ions lourds ou de perturbations de l'alimentation par exemple.

2.1.2 Attaques par canaux auxiliaires

En 1996, Korsher et al. [Kocher 1996] présentent une attaque sur certains crypto-systèmes tels que RSA [Rivest 1978] et Diffie-Hellman [Diffie 1976]. Ils parviennent à trouver des secrets via l'information que laisse fuiter l'implémentation (sans attaque). Ils confirment ainsi qu'il est possible de profiter des différences dans le temps d'exécution de ces algorithmes pour retrouver les clefs de chiffrement secrètes. Ces attaques visant à obtenir de l'information secrète dans une implémentation à l'aide de l'étude de paramètres physiques sont nommées *attaques par canaux auxiliaires* (*side-chanel attacks*), ou attaques par canaux cachés.

Par la suite, les attaques par canaux auxiliaires se sont étendues à l'analyse de la consommation électrique (Differencial Power Analysis (DPA)) [Kocher 1999, Kocher 2011], du cache du processeur [Tiri 2007], des rayonnements électromagnétiques [Gandolfi 2001] ou encore des ondes sonores telles que celles émises par un clavier [Asonov 2004, Gupta 2018]. L'existence de telles attaques implique la nécessité que l'analyse de sécurité d'un système prenne en compte ce type d'objectifs d'attaque.

2.1.3 Attaques par injection de fautes

En 1997, Boneh et al. [Boneh 1997] présentent un modèle théorique permettant de casser des implémentations de crypto-systèmes. Prenant pour appui l'existence de fautes matérielles, ils proposent d'injecter volontairement des fautes pour casser l'implémentation des algorithmes cryptographiques Rivest Shamir Adleman (RSA), Fiat-Shamir [Fiat 1986] et Schnorr [Schnorr 1991]. L'année suivante, Biham et al. [Biham 1997] utilisent le terme de Differencial Fault Analysis (DFA) pour qualifier une attaque reposant sur l'injection volontaire de fautes afin de comparer une exécution normale à des exécutions fautées. Ils parviennent à récupérer la clef secrète d'une implémentation sur carte-à-puce de l'algorithme de chiffrement Data Encryption Standard (DES) à l'aide de l'étude comparative d'un chiffré obtenu à partir d'un clair inconnu et de différents chiffrés dont le calcul a été fauté par la modification d'un bit. Ils proposent également une méthodologie pour appliquer ce type d'attaque. La DFA est une attaque par canaux auxiliaires *active* (où l'attaquant interagit avec le matériel cible).

Dans les décennies suivantes, plusieurs techniques d'injection de fautes sont apparues : modification de la fréquence de l'horloge [Agoyan 2010, Yuce 2018], impulsion électromagnétiques [Poucheret 2011], rayon de lumière [Skorobogatov 2002] et faisceaux lasers [Roscian 2013, Colombier 2019] par exemple. Chaque méthode possède des caractéristiques spécifiques en ce qui concerne la difficulté et le coût de mise en place, la précision, la reproductibilité et l'effet des fautes [Bar-El 2006].

Les exploitations de l'injection de fautes ont montré la possibilité d'obtenir des données secrètes d'un algorithme cryptographique [Boneh 2001a, Biham 1997], l'élévation de privilèges [Timmers 2016], l'introduction de vulnérabilité par dépassement de tampon [Nashimoto 2017, SSTIC 2020] ou bien encore l'évitement d'un boot sécurisé [Nashimoto 2017] ou du module d'exécution de confiance [Nashimoto 2022]. Des techniques d'injection au niveau logiciel ont aussi été développées, comme par exemple *RowHammer* [Park 2014] qui vise à induire un changement d'une cellule DRAM en actualisant de façon répétée les lignes adjacentes.

2.1.4 Plan du chapitre

Cette thèse vise l'aide à la conception d'applications robustes contre les attaques par injections de fautes, qui constituent une menace importante pour la sécurité des composants sensibles. Il s'agit à la fois de l'aide au développeur et à l'auditeur qui évalue la robustesse de l'application. C'est pourquoi les contributions de cette thèse se concentrent majoritairement sur le niveau logiciel (code source, représentation intermédiaire et assembleur), pour s'intéresser aux fautes mettant en cause la logique algorithmique des implémentations. L'objectif de ce chapitre est d'introduire les problématiques liées aux attaques par injection de fautes multiples et à leurs évaluation. La section 2.2 définit les concepts de fautes et de modèles de faute et présente un panorama des modèles considérés afin de présenter les enjeux liés à la modélisation des fautes et de leur propagation au sein d'un système. La section 2.3 discute brièvement de la problématique de la mise en place de contre-mesures dans le cadre d'attaques en faute, ce qui sera détaillé dans le chapitre 6. La section 2.4 se penche sur les problématiques posées par les fautes multiples, c'est-à-dire lorsque l'attaquant est capable d'injecter plusieurs fautes lors de son attaque. La section 2.5 discute des différentes méthodes et outils utilisés pour l'analyse de robustesse dans le contexte de l'injection de fautes.

2.2 Fautes et modèles de faute

Un *modèle de faute* correspond au *modèle d'action* du modèle d'attaquant (section 1.2), instancié au contexte des attaques par injection de fautes. Il représente l'ensemble des fautes qu'un attaquant est capable de réaliser dans le modèle. La notion de *faute* est directement liée au niveau de représentation considéré. Au niveau du code source d'un programme par exemple, les modèles de faute considérés correspondent à des abstractions des effets qu'une faute à plus bas niveau peut avoir sur le programme (par exemple modifier une valeur dans une variable ou changer un branchement).

Cette section vise à présenter les modèles de faute communément considérés aux différents niveaux de représentation en se concentrant sur les modèles au niveau logiciel. La section 2.2.1 présente une classification des niveaux de représentation et discute des caractéristiques d'une faute et des problématiques de mise en relation des modèles de faute à des niveaux variables. La section 2.2.2 donne un aperçu des modèles de faute issus des attaques par injection de fautes et de l'analyse de robustesse. Enfin, la section 2.2.3 discute de la classification des modèles de faute.

2.2.1 Niveaux de représentation et caractéristiques des fautes

Une *faute* est une modification de l'état du système entraînant un effet sur son comportement. Celle-ci peut-être induite par une injection physique, une injection logicielle ou encore une erreur aléatoire. Différents niveaux de représentation sont considérés lorsqu'il

s'agit d'analyser un programme ou d'appliquer des protections : au niveau logiciel (langage C, représentation intermédiaire ou assembleur), niveau micro-architectural (Register Transfer Language (RTL), VHSIC Hardware Description Language (VHDL)), au niveau matériel (technique d'injection physique utilisée) ou encore l'analyse d'un automate représentant le programme par exemple.

La figure 2.1 présente la terminologie qui sera utilisée dans la suite de ce manuscrit en ce qui concerne les niveaux de représentation. Les niveaux sont ici organisés en partant du niveau le plus bas (niveau physique) au niveau le plus haut (niveau algorithmique). Le *niveau physique* correspond à la perturbation physique déclenchant le comportement fauté, que celle-ci soit liée à une erreur aléatoire, une injection de faute physique ou encore déclenchée par des moyens logiciels. Cette injection se traduit au *niveau du circuit* sur les portes logiques, les bascules et latches ou les cellules mémoires et se manifeste en fonction de la partie du système visé, au niveau *micro-architectural* en ciblant une modification des données, des instructions ou de parties spécifiques du processeur par exemple. A plus haut niveau, formant le *niveau logiciel*, les fautes sont observables au niveau binaire, au niveau assembleur et au niveau source qui se rapprochent davantage de la logique algorithmique ou des propriétés logiques attendues par la spécification.

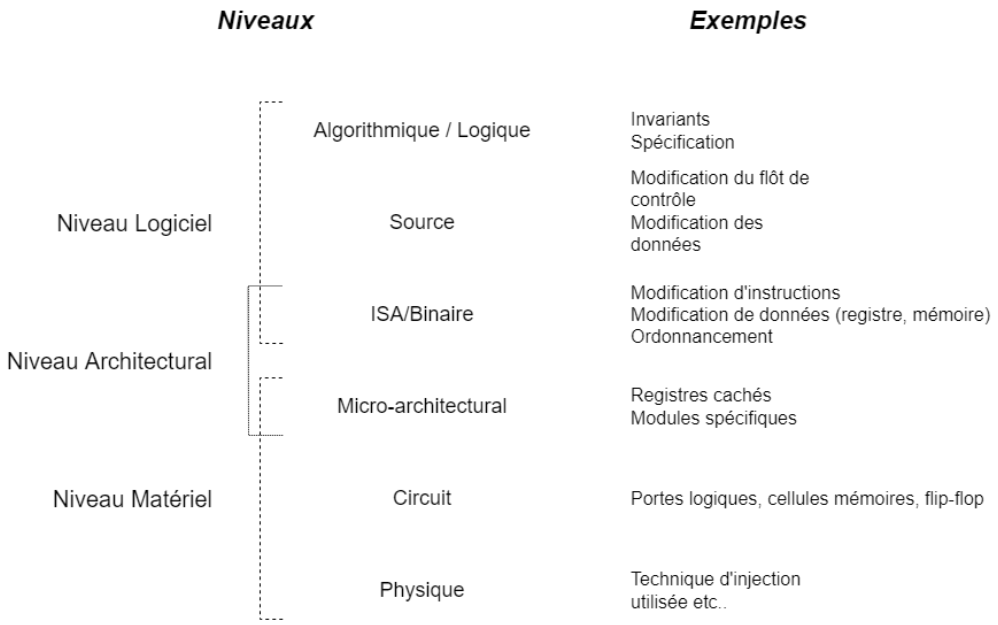

FIGURE 2.1 – Classification et terminologie des niveaux de représentation

Chaque niveau de représentation dispose d'avantages et d'inconvénients en ce qui concerne la représentativité du modèle étudié, la facilité de mise en place ou la facilité avec lesquels les résultats peuvent être observés. Un modèle plus haut niveau sera plus générique, plus proche des propriétés logiques de la spécification, tandis qu'un modèle bas niveau sera plus proche de l'effet des techniques d'injections de fautes.

La manière dont les fautes sont propagées aux travers des différents niveaux n'est pas toujours bien comprise. Observer l'effet d'une faute à un niveau supérieur n'est pas forcément évident. L'effet de la faute peut se perdre, par exemple une modification de donnée qui aurait lieu juste avant une réécriture de cette variable passerait inaperçue (schéma Write Write Read (WWR), aussi appelé *faute inactive*), et les techniques d'in-

jection de fautes peuvent avoir des taux de reproductibilité assez bas. Observer l'effet des fautes est donc une problématique à part entière et la caractérisation des modèles de faute vise à comprendre l'effet des fautes et leur propagation aux niveaux suivants [Balasch 2011, Dureuil 2015, Werner 2020].

Le *modèle de faute* est donc une spécification de l'ensemble des fautes qu'on autorise à l'attaquant et revient à décrire quelles sont les caractéristiques des fautes qui sont acceptées par le modèle. L'*effet de la faute* est une caractéristique des fautes mais d'autres caractéristiques peuvent être considérées. En particulier la *persistance temporelle* (*combien de temps la faute a un impact sur le programme ?*) et la *position spatio-temporelle* (*où et quand la faute est injectée ?*). Pareillement, ces caractéristiques ont un sens qui dépend du niveau de représentation considéré.

2.2.1.1 Position spatio-temporelle d'une faute

Au niveau physique, une faute peut être caractérisée par sa position spatiale (*où la faute est injectée ?*) et sa position temporelle (*quand la faute est injectée ?*). En fonction de la technique d'injection utilisée, la position peut correspondre sur le composant à une entrée physique (tension d'alimentation ou fréquence d'horloge) ou à des bits dans des cellules mémoires ou un bus par exemple. La position temporelle correspond au moment où l'injection est effectuée, par exemple compté depuis le démarrage de l'exécution.

A plus haut niveau, ces deux notions peuvent être plus difficiles à distinguer. Au niveau source par exemple, la position de la faute peut correspondre à un point de contrôle du programme, appelé alors *point d'injection* (Injection Point (IP)). Le listing 2.1 présente les points d'injection dans une fonction `compare` avec un modèle de faute permettant à l'attaquant d'inverser le résultat d'une condition. Durant l'exécution, $IP1$ et $IP2$ peuvent se déclencher, c'est-à-dire qu'une faute peut y être injectée à chaque exécution de boucle.

```
1  bool compare(uint8_t* a1, uint8_t* a2, size_t size)
2  {
3      bool result = true;
4      for(size_t i = 0; i < size; i++) { // IP1
5          if(a1[i] != a2[i]) { // IP2
6              result = false;
7          }
8      }
9      return result;
10 }
```

Listing 2.1 – Points d'injection pour la fonction `compare`

On appellera l'*espace de faute* l'ensemble des points d'injection sur un programme pour un modèle de faute donné. La figure 2.2 présente l'espace des fautes pour le modèle d'inversion de test logiciel, ainsi que l'espace des fautes d'un modèle au niveau physique permettant d'obtenir un comportement d'inversion de test, par exemple en ciblant le registre contenant le résultat de la condition lors des évaluations des tests. Les fautes sont ici les mêmes mais sont caractérisées par un ensemble de points d'injection dans le cas du modèle au niveau logiciel et par les paramètres de l'injection au niveau physique.

2.2.1.2 Persistance temporelle

BarEl et al. [Bar-El 2006] indiquent qu'un circuit peut être sujet à deux types de fautes : les fautes **permanentes** (destructives) et les fautes **transitoires**. Dans le cas de fautes *transitoires* (ou *provisoires*), la faute a un effet immédiat sur le système en provoquant une mauvaise interprétation d'un signal. Lorsque la faute cesse, le circuit revient ensuite à son

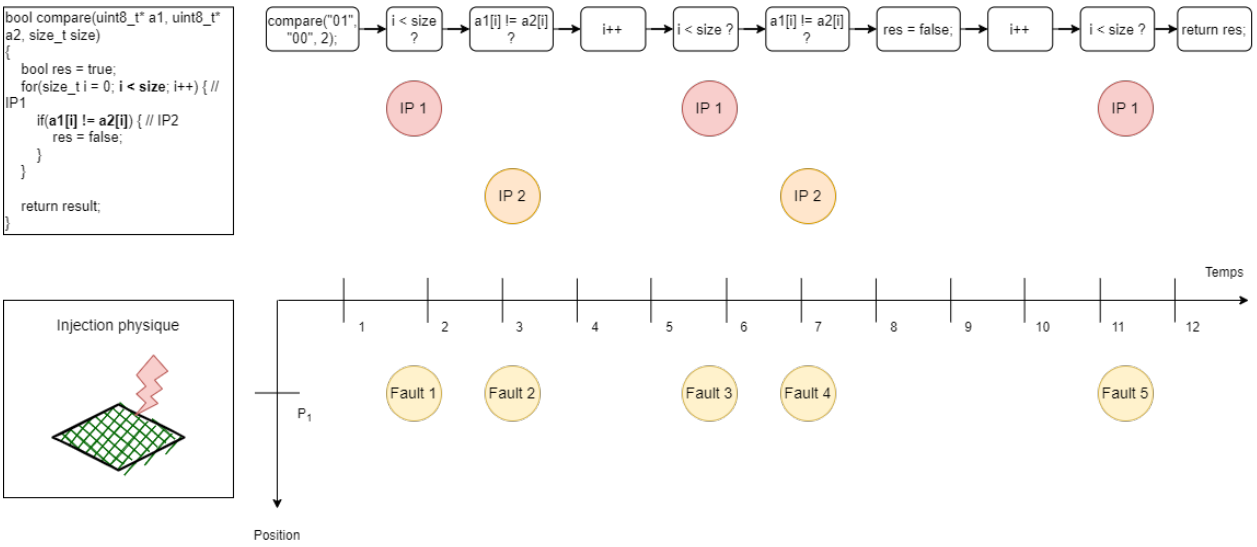

FIGURE 2.2 – Comparaison des fautes sur la fonction compare en inversion de test

état initial. A l'inverse, les fautes *destructives* ont un effet permanent sur le système en altérant l'intégrité d'un registre, d'une cellule mémoire ou du circuit par exemple.

En pratique, cette taxonomie au niveau physique est difficile à transposer à un niveau d'abstraction plus élevé. Une faute transitoire peut avoir un effet *permanent* ou *semi-permanent*, sur le programme. Par exemple, si une faute transitoire modifie une valeur dans un registre qui est ensuite écrite en mémoire vive (Random-Access Memory (RAM)), alors l'effet de la faute persistera jusqu'à la fin de l'exécution du programme. Dans sa thèse [Bréjon 2020], J.-B. Bréjon parle de **faute persistante** lorsque qu'une faute transitoire a un effet qui persiste au delà de son effet physique sur le circuit. Il cite aussi le cas où une faute transitoire implique une modification d'une instruction du programme chargée en RAM. Dans un cas plus extrême, une faute transitoire peut avoir un effet permanent sur un système, par exemple si la valeur fautée est ensuite utilisée dans un système de mise à jour de micro-programme (firmware updater). La faute aura alors un impact sur le système même après la fin de l'exécution du programme, voire provoquera l'arrêt du système, la faute ayant été répercutée dans la mémoire non volatile. On parle aussi de fautes *transitoires répétitives* [Berthomé 2012] lorsqu'une faute transitoire peut être répétée, par exemple en fautant un bus mémoire plusieurs fois pour introduire plusieurs valeurs invalides successivement.

2.2.2 État de l'art des modèles de faute

Cette sous-section propose un aperçu des modèles de faute considérés dans la littérature, à la fois par les attaques physiques et par les outils d'analyse. Cet état de l'art est organisé en fonction du niveau de représentation du programme. La section 2.2.2.1 présente les modèles au niveau matériel, la section 2.2.2.2 les modèles de faute au niveau de la micro-architecture et du jeu d'instructions, ainsi que des modèles au niveau assembleur et finalement la section 2.2.2.3 se concentre sur les modèles au niveau source.

2.2.2.1 Modèles au niveau matériel

Au niveau physique, l'injection d'une faute introduit une perturbation qui va modifier le fonctionnement du circuit. Le circuit peut consister en des portes logiques, des cellules

mémoire ou encore des bascules.

Le modèle de faute considéré à bas niveau est lié à la technique d'injection de fautes utilisée. Certaines techniques comme celles basées sur les ondes électromagnétiques [Maistri 2014, Dumont 2019] ou les impulsions laser [Van Woudenberg 2011, Dutertre 2014b] ont une haute précision à l'inverse des glitchs de température [Hutter 2013] par exemple. Les caractéristiques des fautes injectées vont alors dépendre des paramètres expérimentaux choisis (moment et position de l'injection, durée et intensité d'un rayonnement etc.).

Valeur / Granularité	Niveau bit	Niveau byte	Autres plages
mise à 0	bit-set	byte set	set
mise à 1	bit-reset	byte reset	reset
inversion	bit flip	byte flip	flip
aléatoire	bit ran	byte ran	rand

TABLE 2.1 – Modèle de faute sur les données

Au niveau d'une donnée dans le circuit (dans un bus ou une cellule mémoire par exemple), on distingue souvent les modèles en fonction de la granularité de la plage de bits pouvant être visée et la façon dont la valeur peut être modifiée, comme indiqué dans la table 2.1. La granularité de la faute en donnée peut consister en un seul bit, un octet, une plage contiguë quelconque ou une plage non contiguë de bits. Concernant la valeur fautée, on peut distinguer les modèles de mise-à-zéro et de mise-à-un ainsi que l'inversion des bits. Les modèles de type aléatoire permettent de simuler les injections en données sur des mémoires chiffrées où l'attaquant a un contrôle limité sur la valeur qui pourra être injectée.

2.2.2.2 Modèles au niveau architectural et ISA

Au niveau architectural, les fautes peuvent être distinguées en fonction de la partie de l'architecture qui est ciblée comme présenté dans la figure 2.3 :

— fautes sur les instructions,
— fautes sur les données (registres, cache, mémoire),
— fautes sur la micro-architecture : pipeline, registres cachés...

Lorsqu'on se place au niveau du jeu d'instructions (Instruction Set Architecture (ISA)), une modification de bits induit une transformation des instructions ou des données. Une faute peut avoir pour effet la modification de l'opérande (les paramètres de l'instruction), de l'opcode (le type de l'instruction) ou les deux.

La figure 2.4 présente la spécification de l'encodage des trois versions de l'instruction *MOV* dans le jeu d'instructions *ARMv7-M thumb2* [ARM 2010]. Les bits correspondant à l'opcode dans l'encodage sont indiqués avec une valeur fixe binaire tandis que les opérandes sont nommées. Les opérandes de la forme `immX` correspondent à la valeur immédiate (constante) qui sera déplacée et `rd` correspond au registre de destination. Si ces bits sont changés par une faute, le type de l'instruction restera le même (MOV-TX), mais avec des paramètres différents [Colombier 2019]. Un cas particulier du remplacement d'instruction est le saut d'instructions [2] très largement étudié [Balasch 2011, Kelly 2017, Moro 2013], dans lequel la faute permet d'ignorer l'exécution d'une ou plusieurs instructions (par exemple, en transformant une instruction en instruction No-OPeration (NOP))). Dans le cas de jeu d'instructions à taille d'instruction variable comme c'est le cas pour *thumb*, la mutation au

2. D'autres types de fautes peuvent toutefois induire un saut d'instruction, comme par exemple la modification du compteur du programme (Program Counter (PC)).

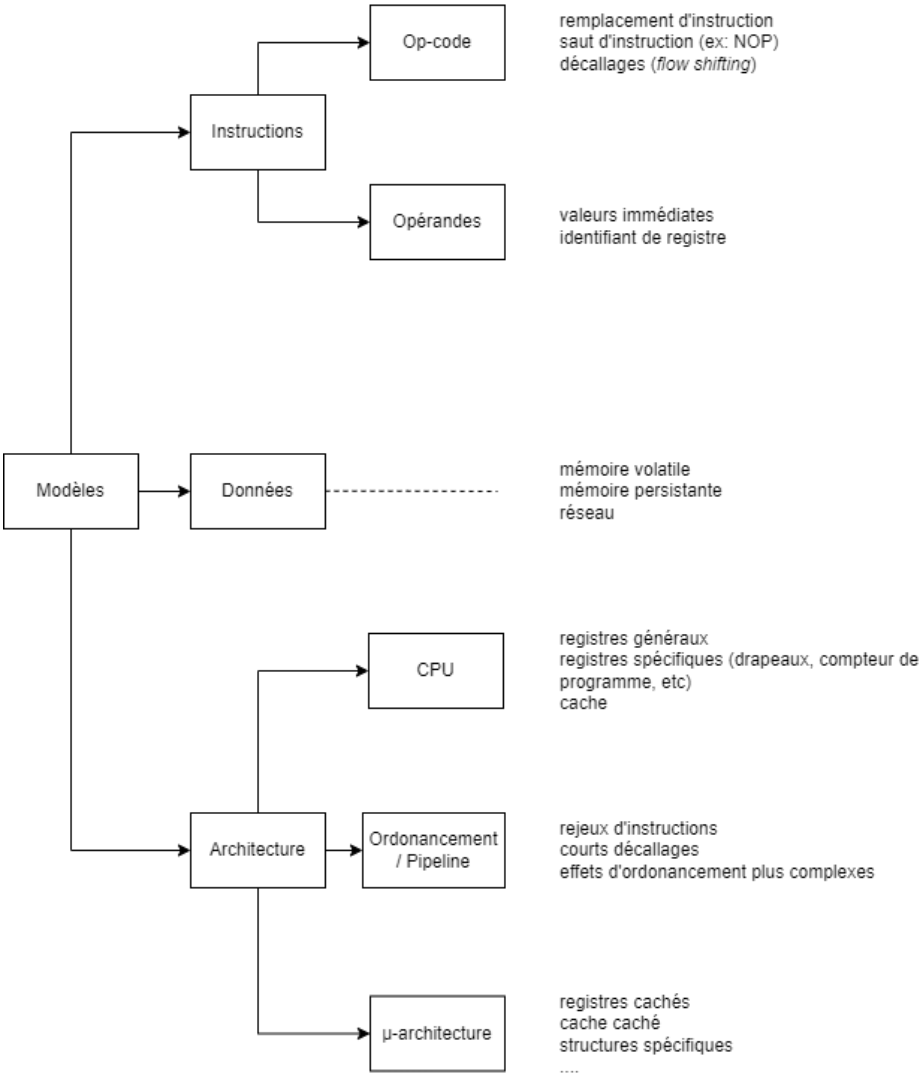

FIGURE 2.3 – Classification des modèles au niveau architectural

niveau de l'*op-code* peut jouer sur la taille de l'instruction (en passant d'une instruction de deux à quatre octets ou réciproquement) et peut entraîner une mutation en cascade des instructions interprétées par la suite, et donc un impact significatif sur la structure du reste du programme au niveau binaire. Cette suite de décodage erronés *flow shifting* [Berthomé 2012] est généralement assez courte en raison d'instructions invalides amenant à un crash du processeur [Shacham 2007].

La modification des données consiste à changer les bits dans un registre du processeur, sur un bus mémoire pendant son transit ou dans la mémoire. La corruption de registre est également un modèle très étudié dans l'analyse au niveau binaire [Blömer 2003, Verbauwhede 2011]. De la même manière qu'au niveau matériel, différentes valeurs et plages de bits peuvent être considérées en fonction du modèle de faute comme indiqué dans la figure 2.1. Dans le domaine de la tolérance aux fautes, les modèles étudiés se restreignent souvent à une inversion de bit simple sur la donnée [Lu 2015a, Sharma 2013, Van Der Kouwe 2014,

Encoding T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

Encoding T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	0	S	1	1	1	1	0	imm3			Rd				imm8							

Encoding T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	1	0	0	imm4				0	imm3			Rd				imm8							

FIGURE 2.4 – Encodage de l'instruction MOV sur l'architecture ARMv7-M thumb2

Georgakoudis 2017, Le 2018a]. Les modèles considérant la modification de plusieurs bits contiguës voire de fautes indépendantes existent [Kelemen 2007]. La modification des données peut aussi correspondre à la modification d'une adresse ou de son calcul (comme un index de tableau par exemple).

Au niveau micro-architectural, c'est-à-dire en tirant parti des spécificités de l'architecture interne du processeur et des composants, des modifications plus subtiles peuvent être observées. Rivière et al. [Rivière 2015] montrent qu'il est possible d'exploiter le tampon de pré-chargement du processeur (*prefetch buffer*) pour annuler son remplissage et ainsi répéter une instruction en ignorant une partie des instructions suivantes. Le rejeu d'instruction [Rivière 2014] consiste à exécuter deux fois une instruction, par exemple en attaquant la valeur du PC, ou de la pipeline du processeur. Lorsqu'il s'accompagne du saut de l'instruction suivante, il peut s'assimiler à un remplacement de l'instruction. Plus récemment, Laurent et al. proposent des exploitations de fautes sur la micro-architecture, avec notamment des effets sur le processeur non-observables sur le jeu d'instructions [Laurent 2018, Laurent 2019]. Par ailleurs, les modèles de faute visant les caches du processeur correspondent à des modèles ciblant la micro-architecture du processeur.

2.2.2.3 Modèle au niveau source

A plus haut niveau, les modèles de faute s'intéressent davantage aux modifications de la logique algorithmique du programme. En fonction de si on se place au niveau du code source ou d'un niveau intermédiaire, la granularité des fautes peut varier. La figure 2.5 présente une classification des modèles de faute au niveau source. Deux grandes classes de modèles de faute se dégagent de la littérature : l'*altération des chemins d'exécution* du programme et la *modification des données*.

L'altération des chemins d'exécution concerne les fautes ayant un effet sur l'intégrité du flot de contrôle [Abadi 2009, Sayeed 2019]. Il s'agit de détourner le flot d'exécution de son comportement normal. Le *graphe de flot de contrôle* (Control Flow Graph (CFG)) est une représentation du flot de contrôle du programme correspondant à un graphe orienté. Les nœuds sont des blocs de base qui correspondent à une suite d'instructions atomiques (toujours exécutées en séquence). Les arêtes correspondent aux branchements conditionnels

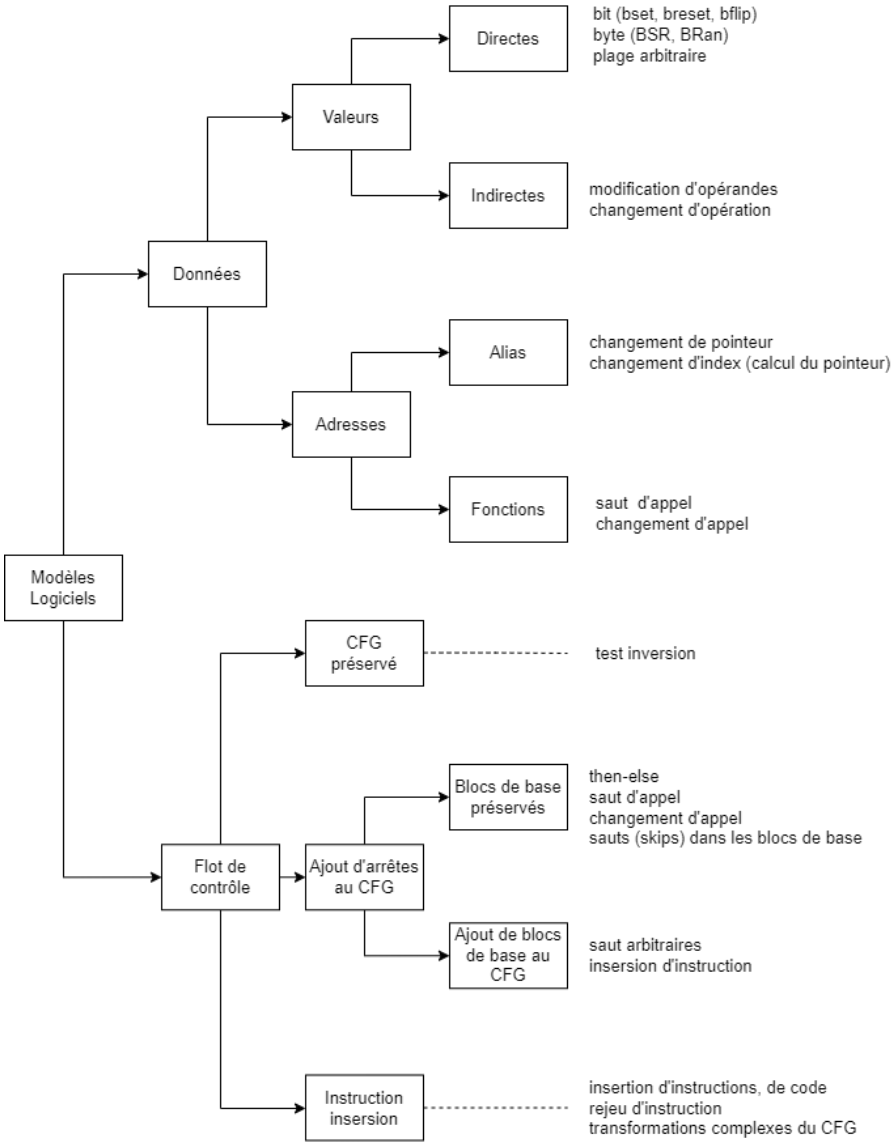

FIGURE 2.5 – Classification des modèles de faute au niveau source

et aux sauts. Ainsi la figure 2.5 distingue les modèles axés sur la modification du flot de contrôle en fonction de leur impact sur le graphe de flot.

Le modèle de l'inversion de test [Berthomé 2012, Potet 2014, Dureuil 2016b] consiste à inverser le chemin sélectionné après une opération de branchement conditionnel. Dans [Bouffard 2011], Bouffard et al. parviennent à modifier le flot de contrôle d'un programme Java Card [3] en attaquant l'adresse de retour d'une fonction. Les modèles basés sur le saut d'instructions existent aussi au niveau source [Moro 2013, Potet 2014, Barry 2016, Breier 2019]. Ils ont généralement une granularité plus faible au niveau logiciel qu'au niveau matériel, comme par exemple dans [Lalande 2014], où Lalande et al. s'intéressent à un modèle de saut d'instructions au niveau du langage C. Il peut aussi s'agir de sauts d'appels

3. Plateforme Java minimaliste embarquée pour les cartes à puce.

de fonction [Dureuil 2016b]. Ces modèles attaquant l'intégrité du flot de contrôle sont un sujet important au niveau source, car ils induisent des comportements difficiles à anticiper et peuvent produire des nouveaux chemins dans l'application, que ce soit dans les outils d'analyse de robustesse tels que SmartCM [Machemie 2011], l'évaluation de contre-mesures [Séré 2011] ou l'introduction de contre-mesures, par exemple à la compilation [Werner 2015].

La modification de données s'apparente principalement à la modification des valeurs lues ou écrites dans les variables, ou les valeurs temporaires utilisées pour calculer les expressions (i.e. les registres). Breier et al [Breier 2019] étudient l'effet d'un nombre arbitraire d'inversions de bits dans un registre. La modification d'une donnée lors de la lecture en mémoire est également un modèle standard dans l'analyse de robustesse au niveau logiciel [Moro 2013, Berthomé 2012]. Le modèle d'inversion de bit dans une zone mémoire est aussi fréquemment utilisé dans le domaine de la tolérance aux fautes [Benso 1998, Georgakoudis 2017]. L'attaque *Rowhammer* [Kim 2014] qui a été abordée dans le chapitre 1 correspond à une modification de données sur la mémoire.

La modification de données peut avoir un effet sur le flot de contrôle, par exemple dans le cas d'une faute d'une adresse de retour, d'une adresse de fonction, d'une adresse lors d'un saut arbitraire, et plus généralement lorsqu'une valeur est utilisée par la suite dans une condition ou un saut. A l'inverse, le flot de contrôle peut avoir un effet comparable à une mutation de donnée puisque le flot détourné peut modifier des données différemment du comportement normal du programme. Lacombe et al [Lacombe 2021] proposent un modèle visant les expressions au niveau CIL (représentation intermédiaire de l'outil Frama-C), permettant de considérer à la fois les modifications de données arbitraire et les fautes visant le flot de contrôle (avec la mutation de la valeur finale de l'expression dans les conditions).

2.2.3 Classification des modèles au niveau logiciel

Les sections précédentes ont présenté les problématiques liées à la mise en relation des modèles de faute à différents niveaux de représentation. Celles-ci concernent à la fois la représentativité des modèles étudiés à haut niveau par rapport à la réalité physique, la difficulté de l'observation et de la caractérisation des fautes et la compréhension de leurs effets [Dureuil 2015, Werner 2020]. Cette sous-section vise à présenter différentes approches pour classifier les modèles au niveau logiciel ainsi que la difficulté pour faire correspondre des modèles de niveaux différents.

La figure 2.5 présentée dans la section 2.2.2.3 fait une première distinction entre les modèles sur le flot de contrôle et ceux sur les données. La classification des modèles source peut aussi être réalisée en partant de la cible des fautes à bas niveau (instructions, données, registres, pipeline etc.). C'est en partie l'approche qui est utilisée dans la figure 2.3 présentant les modèles au niveau architectural. La modification de données sur des zones différentes de l'architecture implique alors des modèles haut niveaux très variables.

Une autre solution serait de classifier les modèles en fonction de l'inclusion de l'ensemble des fautes pouvant être réalisées. Cela peut être fait en considérant l'espace de faute de chaque modèle ou bien en comparant les comportements obtenus par chaque modèle. Il est par exemple possible de considérer qu'un modèle de mutation de donnée est un sous-ensemble d'un modèle de modification arbitraire de donnée ou encore qu'une inversion de test peut correspondre, en partie, à un cas spécifique du saut d'instruction lorsqu'on se place au niveau de représentation ISA. Cependant, cette approche a ses limites puisque les espaces de fautes de deux modèles peuvent ne pas avoir de relation d'inclusion et selon le niveau de représentation choisi, les relations entre les modèles de faute peuvent être

très différentes. La combinaison de modèles de faute et la prise en considération de fautes multiples compliquent encore cette notion d'inclusion [4].

Cela illustre la difficulté de faire correspondre des modèles à des niveaux de représentation différents et la question de la légitimité de raisonner à propos de modèles trop bas niveau lorsqu'on considère un programme au niveau logiciel. Les programmes doivent faire face à des attaquants très variés, et la définition ainsi que la modélisation des fautes sont des difficultés de l'évaluation de la robustesse d'un programme. Les modèles au niveau logiciel visent à être suffisamment génériques pour couvrir un maximum les modèles pouvant effectivement être observés mais sans viser la correspondance exacte avec ces modèles bas niveau et s'orientent plutôt vers la prise en compte de la logique algorithmique du programme.

2.3 Protection contre les attaques physiques

Afin de lutter contre les attaques physiques, de nombreuses solutions de protections ont été proposées [Bar-El 2006, Yuce 2018]. Ces protections peuvent prendre des formes variées en fonction du niveau de représentation et du modèle d'attaquant considéré. Cette section vise à donner un court aperçu des protections contre les attaques par injections de faute et les attaques physiques (fautes accidentelles et canaux auxiliaires). Le chapitre 5 approfondit les problématiques liées à l'analyse et l'évaluation de protections dans le cadre d'attaques en fautes et présente un état de l'art des protections logicielles existantes dans la littérature.

Définition 2.1. *Une protection correspond à toute modification du programme, de l'architecture ou du matériel et de leurs fonctionnements visant à empêcher une attaque d'aboutir, la détecter, la corriger ou la rendre plus difficile à effectuer.*

Définition 2.2. *Une contre-mesure est une protection visant à détecter une attaque afin de pouvoir ensuite la signaler, la corriger ou prendre des mesures comme le redémarrage du système ou sa coupure.*

Au niveau physique, les protections peuvent prendre la forme de détecteurs surveillant la tension d'alimentation ou la fréquence du circuit [Zussa 2014]. Les boucliers actifs (*active shields*) [Bar-El 2006] correspondent à un maillage métallique dans lequel les données circulent en continue. Si le maillage est endommagé ou modifié, le circuit ne fonctionne plus. Les détecteurs physiques permettent de réagir à une fréquence ou une tension inhabituelle [Bar-El 2006]. Les protections visant à limiter la connaissance de l'attaquant en rendant l'observation plus difficile sont très fréquentes dans le domaine des canaux auxiliaires. Parmi les protections physiques de ce type on peut citer le chiffrement de la mémoire [Barenghi 2012] ainsi que l'ajout d'aléa dans la consommation de courant par exemple [Moro 2014b] ou encore dans l'ordonnancement des instructions [Witteman 2008].

Au niveau architectural, la tâche de l'attaquant peut être rendue plus ardue en utilisant un encodage pour le jeu d'instructions compliquant attaques : par exemple un encodage différent de `0` pour l'instruction NOP est une solution pour se protéger contre les attaques de mise-à-0. Des protections par redondance des calculs ou des données peuvent par exemple être obtenues en ayant deux circuits de calcul dans le processeur ou l'Arithmetic Logic Unit (ALU), qui comparent ainsi leur résultats, cette redondance pouvant être étendue à n circuits redondants [Bar-El 2006].

Au niveau logiciel, les protections peuvent être appliquées sur toute la couche logicielle. On distinguera les protections appliquées en tant que transformation du programme et celles

4. Par exemple, un modèle de mutation de donnée sur une instruction d'écriture de mémoire (store) peut être simulée par un nombre non borné de mutations de donnée en lecture (load).

appliquées au niveau du système (système d'exploitation par exemple). Les protections qui effectuent une transformation du programme source peuvent être appliquées à plusieurs niveaux de représentation du programme. L'utilisation d'instructions idempotentes afin de se protéger contre les sauts d'instructions [Moro 2014b] a été proposée au niveau assembleur tandis que des protections ont été proposées au niveau du langage C [Lalande 2014] ou lors de la compilation [Barry 2016, Proy 2017]. Certaines approches visent la protection de l'intégrité du flot de contrôle par des mécanismes de signature de blocs ou de fonctions [Oh 2002, Reis 2005, de Ferrière 2019] ou encore par l'utilisation d'une *pile cachée* [Dureuil 2015] par exemple. L'utilisation de booléens endurcis visent à compliquer la tâche de l'attaquant d'une façon similaire à l'exemple de l'encodage du NOP pour le niveau architectural.

Une partie des protections physiques, notamment celles basées sur la redondance des calculs ou de la mémoire, ou le chiffrement de celle-ci, peuvent être implémentées à plus haut niveau. Cependant, les protections logicielles peuvent être limitées par rapport à leurs équivalents physiques. En général, les systèmes sont protégés en combinant différents niveaux de protection [Yuce 2016].

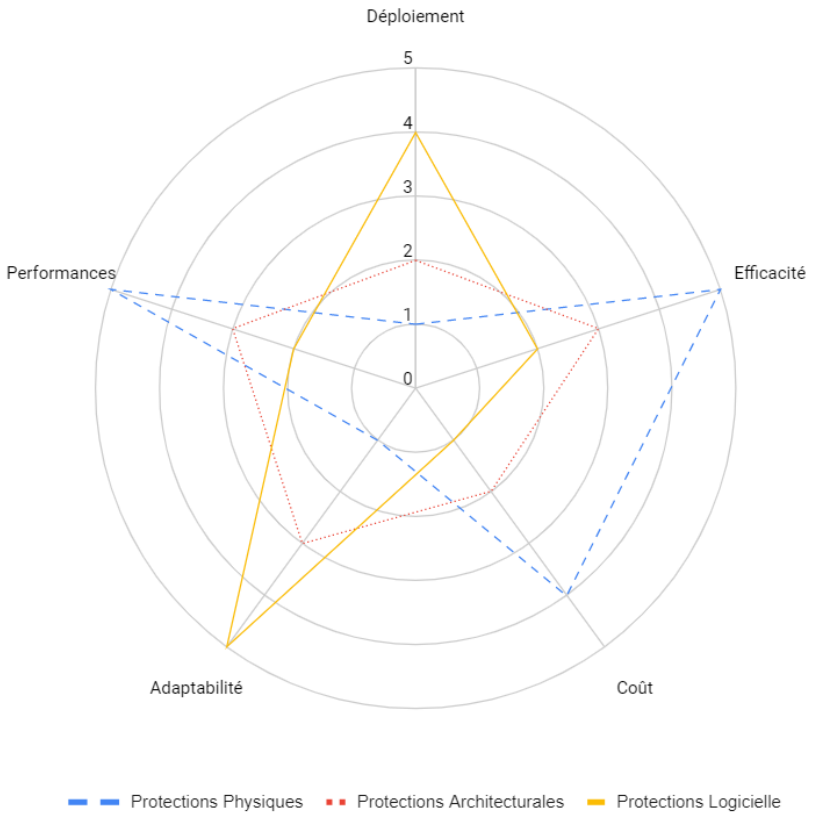

FIGURE 2.6 – Caractéristiques des protections à différents niveaux

La figure 2.6 compare certaines caractéristiques des protections en fonction du niveau auquel elles sont introduites :

— *Portabilité* : capacité à s'adapter à une architecture ou une application donnée.
— *Déploiement* : facilité de mise-à-jour de la protection.
— *Efficacité* : capacité à protéger.
— *Coûts* : coûts de développement et de mise en place.
— *Performance* : taille du code, consommation mémoire et temps d'exécution ajoutés.

Les protections physiques permettent de protéger l'ensemble des programmes d'un composant mais sont plus coûteuses à mettre en place et ne peuvent pas être modifiées une fois que le produit est déployé (une carte à puce par exemple). À l'inverse, les protections logicielles peuvent être ajoutées à l'aide de mises-à-jour logicielles. Les protections de plus haut niveau peuvent ainsi être adaptées aux protections disponibles à plus bas niveau sur lesquelles elles peuvent s'appuyer. Les protections bas niveau peuvent protéger des modules du système qui ne sont pas accessibles au niveau logiciel, mais ne peuvent pas être adaptées à un programme ou une pile logicielle particulière et visent une protection générique du composant.

La protection contre les attaques en fautes est fortement compliquée contre les attaques en *fautes multiples*.

2.4 Les fautes multiples

La littérature récente contient des exemples d'attaques en fautes multiples [Kim 2007, Barenghi 2012, Natella 2016, SSTIC 2020].

Les fautes multiples rendent le processus de développement d'un programme sécurisé d'autant plus difficile que le nombre de configurations de fautes possibles sur un programme croît rapidement. Cette explosion combinatoire complique l'expertise humaine et l'analyse des outils pour l'évaluation de robustesse. De plus, la combinaison de plusieurs modèles de faute doit être considérée et rend plus complexe la protection et l'analyse de robustesse.

Dans le cas de la protection d'un système, la présence de fautes multiples complique également la tâche. En effet, si plusieurs fautes peuvent être injectées, alors une protection peut elle-même être visée par une première faute pour rendre possible la seconde. Dans la fonction *compare* du programme `verify_pin` présentée dans le listing 2.2, une première injection peut éviter la boucle `for` (par exemple avec une inversion de la condition `i < size`) et une seconde peut inverser la condition `if(i != size) killcard()` afin d'éviter la contre-mesure.

```
1  bool compare(uint8_t* a1, uint8_t* a2, size_t size)
2  {
3      bool result = true;
4      for(size_t i = 0; i < size; ++i)
5          if(a1[i] != a2[i])
6              result = false;
7
8      if(i != size) // Protection
9          killcard();
10
11     return result;
12 }
```

Listing 2.2 – Compteur de boucle sur la fonction `compare`

Cette possibilité d'attaquer les protections elles-mêmes, d'autant qu'une grande partie de la littérature propose des contre-mesures visant principalement des fautes uniques, rend l'évaluation et la conception de protections plus complexes dans le contexte des fautes multiples. La prise en compte des fautes multiples pour l'analyse de robustesse de programme

et pour l'analyse de protections est une problématique à laquelle cette thèse vise à répondre sur deux aspects :

— Maîtriser l'exploration des exécutions en multi-fautes malgré l'explosion combinatoire des chemins (chapitres 3 et 4).

— Aider à l'analyse et au placement des contre-mesures dans un contexte multi-fautes (chapitres 5 et 6).

Avant de s'intéresser à l'outil Lazart, la suite de se chapitre se termine par une analyse des outils existants.

2.5 État de l'art des outils d'analyse de robustesse pour l'injection de faute

Cette section propose un aperçu des outils existants pour l'évaluation de robustesse dans le cadre de la tolérance aux fautes et des attaques par injection de fautes. Elle vise aussi à proposer un ensemble de critères pour la comparaison de ces outils. L'état de l'art des outils et des techniques d'analyse pour l'injection de fautes a déjà fait l'objet de plusieurs articles de recherche [Kooli 2014, Given-Wilson 2017, Gangolli 2022] et de surveys [Christofi 2013, Dureuil 2016a, Heydemann 2017]. Les outils visant la tolérance aux fautes sont également considérés.

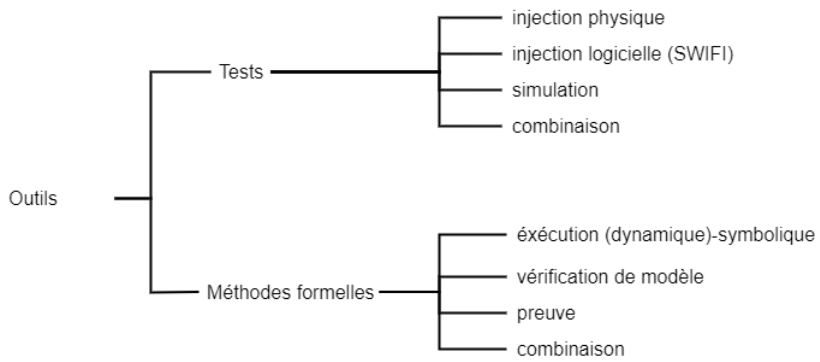

FIGURE 2.7 – Organisation et périmètre de l'état de l'art

La suite de cette section s'organise comme suit. La sous-section 2.5.1 présente différents critères qui seront considérés pour la comparaison des outils. L'état de l'art est organisé en fonction de la technique d'analyse employée comme montré dans la figure 2.7. Les sous-sections 2.5.2 et 2.5.3 présentent respectivement les méthodes basées sur les tests et celles reposant sur les méthodes formelles. Enfin, la sous-section 2.5.4 conclut cette section.

2.5.1 Caractéristiques des outils

Différentes caractéristiques peuvent être considérées lorsqu'on s'intéresse à un outil d'analyse concernant les attaques en fautes. La figure 2.8 présente une classification d'un ensemble de caractéristiques issus de la littérature et qui seront considéré dans cette section :

- **C1** Caractéristiques opérationnelles :
  - *Coûts* : regroupe les coûts liés au développement de l'outil en termes de moyens humains et de matériel ainsi que le niveau d'expertise requis par l'utilisateur pour l'utilisation de l'outil.

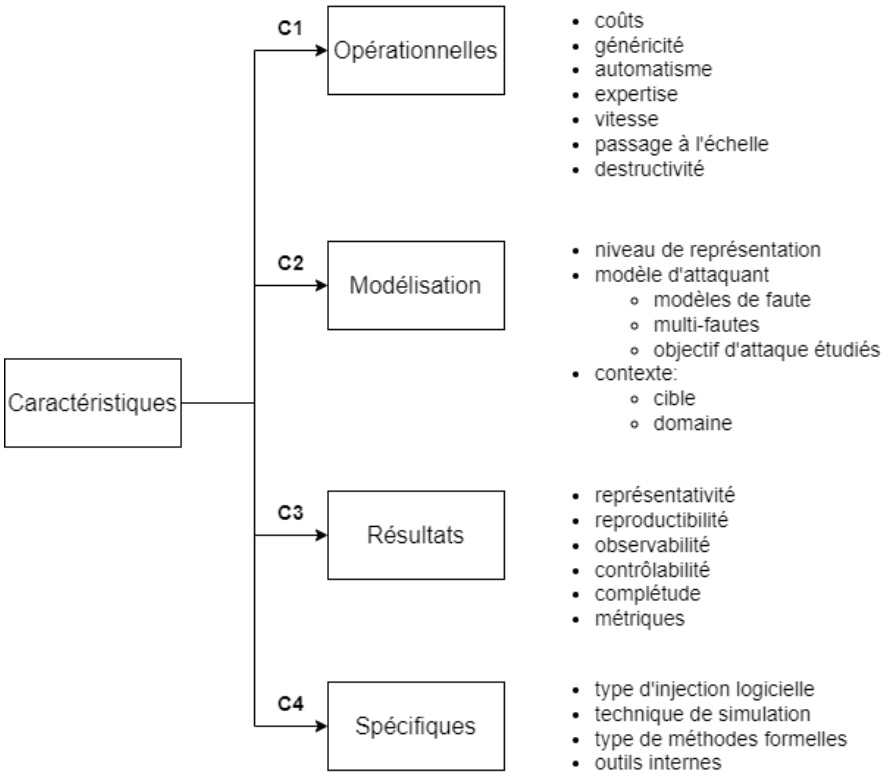

FIGURE 2.8 – Classification des caractéristiques pour les outils d'analyse de robustesse contre les fautes

- *Généricité* : détermine à quel point l'outil est spécifique à un matériel ou une architecture donnée (on parlera à l'opposé de *spécificité*).
- *Automatisation* : niveau d'intervention de l'utilisateur requis.
- *Expertise* : niveau d'expertise de l'utilisateur dans le domaine étudié ou la méthode d'analyse utilisée par l'outil.
- *Vitesse* : comprend la vitesse d'exécution de l'outil pour une analyse ou encore le temps de mise en place nécessaire entre deux expérimentations.
- *Passage à l'échelle* : capacité de l'outil à analyser des programmes / systèmes complexes.
- *Destructivité* : possibilité d'endommager le système étudié par l'analyse.

• **C2** Caractéristiques liées à la modélisation :
- *Niveau de représentation* : niveau de représentation de l'analyse (à ne pas confondre avec le niveau d'implémentation de l'outil).
- *Modèle d'attaque* : modèles de faute supportés, support des fautes multiples et objectifs d'attaque étudiés.
- *Contexte d'analyse* :
  - *Cible de l'analyse* : le type de système visé par l'analyse (programme, application, circuit, machines distribuées...).
  - *Domaine* : attaques en fautes, canaux auxiliaires ou fautes accidentelles.

• **C3** Caractéristiques liées aux résultats :
- *Représentativité* : représentativité des résultats obtenus par rapport à une véri-

table attaque.
  - *Reproductibilité* : détermine si les résultats obtenus peuvent facilement être reproduits.
  - *Observabilité* : correspond à la capacité de la méthode d'observer l'effet des fautes injectées.
  - *Contrôlabilité* : détermine le contrôle qu'offre la méthode sur les fautes qui seront injectées.
  - *Complétude* : couverture des exécutions fautées par l'analyse.
  - *Métriques* : métriques produites par l'outil.
- **C4** Caractéristiques spécifiques : sous-classe de la technique d'analyse, technique d'implémentation.

Les caractéristiques suivantes, correspondant à des combinaisons de caractéristiques, seront aussi étudiées :

— *Performance* (**C1**) : passage à l'échelle et vitesse.
— *Mise-en-place* (**C1**) : regroupe les coûts lié au développement de l'outil en termes de moyens humains et de matériel et la destructivité.
— *Accessibilité* (**C1**&**C3**) : regroupe les caractéristiques de l'expertise requise pour l'utilisateur, l'automatisme et la contrôlabilité.

2.5.2 Outils basés sur le test

Cette sous-section s'intéresse aux outils basés sur le test, c'est-à-dire qui effectuent une sous-approximation des exécutions. Cette classe regroupe un grand nombre de techniques et d'outils :

— L'*injection physique* (section 2.5.2.1) qui correspond à l'expérimentation en attaquant physiquement le système.
— L'*injection logicielle* (section 2.5.2.2) qui englobe tous les outils émulant l'effet des fautes physiques au niveau logiciel.
— La *simulation*[5] (section 2.5.2.3) qui correspond aux outils exécutant le programme dans un simulateur dans lequel les fautes sont injectées.

2.5.2.1 Injection physique

Une méthode d'injection de fautes physique (laser, glitches...) permet d'étudier le comportement du système soumis à des fautes. Le terme *injection de faute* a d'ailleurs émergé dans la littérature avant l'apparition des attaques par injection de fautes pour désigner des plateformes telles que MESSALINE [Arlat 1990] ou encore RIFLE [Madeira 1994]. Toute technique d'attaque par faute peut théoriquement être utilisée comme protocole de test physique pour un circuit.

La figure 2.9 résume les avantages et inconvénients de ce type de techniques d'analyse. L'indication *CI* fait référence aux classes de caractéristiques présentées précédemment. L'injection physique à l'avantage d'obtenir des résultats très représentatifs sur la robustesse du programme puisque la faute est injectée directement sur le système à analyser et n'inclut donc pas d'étape intermédiaire ou d'abstraction pouvant fausser les résultats. L'observation et la caractérisation de la propagation des fautes observées sont difficiles et la mise en place d'un système d'observation peut introduire une perte de précision [Faurax 2009, Kooli 2014].

5. Une partie de la littérature utilise le terme *simulation* par opposition aux méthodes formelles [Kooli 2014, Heydemann 2017]. Dans la suite de ce manuscrit, le mot *simulation* ne sera pas employé en ce sens, parlant à la place d'*outils basés sur les tests*.

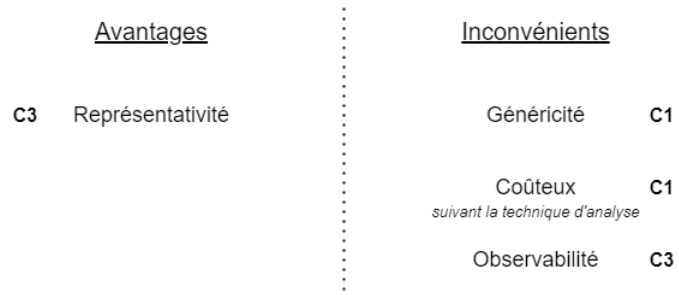

FIGURE 2.9 – Caractéristiques de l'injection physique

La mise en place de tels bancs de test peut s'avérer coûteuse en fonction de la technique d'injection de fautes utilisée. Ces méthodologies d'analyse par injection physique sont souvent très spécifiques au système étudié par rapport à des approches plus haut niveau. De plus, certaines techniques d'injections destructrices peuvent endommager l'appareil étudié. Cependant, ces méthodes d'injections physiques sont utilisées pour la certification des composants sécurisés afin de démontrer la faisabilité d'une attaque.

2.5.2.2 Injection au niveau logiciel (SWIFI)

Une autre technique très largement utilisée est la reproduction au niveau logiciel des fautes, ou de l'effet causé par des fautes au niveau de représentation considéré. Cette technique désignée SoftWare Implemented Fault Injection (SWIFI) a émergé dès la fin des années 1980 [Segall 1988, Kanawati 1992, Kao 1993] dans le milieu des fautes accidentelles pour ensuite se développer dans le domaine des attaques par injection de fautes jusqu'à aujourd'hui [Georgakoudis 2017].

Fault Injection-based Automated Testing (FIAT) [Segall 1988] est l'un des premiers outils de type SWIFI et a été proposé en 1988. Il vise l'analyse de systèmes distribués. Les fautes autorisées sont décrites par l'utilisateur à l'aide d'un langage spécifique et sont ensuite générées lors de la campagne d'expérimentation. Cet outil a notamment été utilisé pour analyser des fautes de type mise-à-zéro et mise-à-un d'octets et l'inversion de deux bits complémentaires [6] [Barton 1990]. Fault Injection and moNitoring Environment (FINE) [Kao 1993], est une plateforme d'analyse qui considère à la fois les fautes physiques et les fautes liées à des erreurs d'implémentation ou de conception logicielle, et qui vise à étudier la propagation des fautes dans un système d'exploitation. L'outil injecte les fautes au niveau logiciel à l'aide d'un noyau UNIX modifié traçant l'exécution et proposant des appels systèmes supplémentaires. Les fautes matérielles consistent en une mutation de données dans la mémoire, les registres ou les bus. Les fautes logicielles sont implémentées en tant que modifications du programme binaire et incluent : données mal initialisées, assignation incorrectes, vérifications des cas d'erreurs incomplètes ou encore mauvaise sortie du programme. L'outil vise aussi à étudier comment les erreurs se propagent entre les différentes machines et les différents niveaux.

L'outil DistributEd Fault Injection and moNitoring Environment (DEFINE) [Kao 1996] est une évolution de FINE qui supporte l'injection dans un système distribué. Celui-ci permet l'injection dans chaque machine, dans n'importe quelle application en mode utilisateur ou superviseur. En plus des appels systèmes rajoutés et de la modification de l'image binaire

6. C'est-à-dire deux bits de valeurs différentes, afin d'éviter d'être détecté par un code correcteur d'erreur simple.

du programme déjà utilisées dans FINE, DEFINE utilise un handler modifié d'interruptions de l'horloge matérielle afin de contrôler l'injection des fautes dans les bus et la mémoire. Des expérimentations ont été menées sur un ensemble de 7 machines SunOS pouvant être fautées, l'une d'entre elles étant utilisée comme serveur. Les auteurs constatent que les erreurs sur le serveur se révèlent plus critiques en général que les fautes sur le client.

EXception based Fault Injector (EXFI) [Benso 1998] est un environnement d'injection de fautes qui utilise le *trap execution mode* des microprocesseurs pour l'injection de fautes. Ainsi, EXFI ne nécessite pas de matériel spécialisé (comme FERRARI [Kanawati 1992] par exemple) ou la présence d'un système d'exploitation (comme les outils utilisant des traps logicielles type FINE). Une passe est utilisée pour réduire cet ensemble de fautes en fonction de règles spécifiques permettant d'écarter certaines fautes sans perdre en précision. Ces règles visent à écarter les fautes qui n'auraient pas d'effet sur le système, seraient forcément détectées par un mécanisme de détection d'erreur et celles dont le comportement est déjà couvert par une autre faute de la liste. EXFI supporte un modèle de fautes temporaires de type bit-flip pouvant cibler les instructions, la mémoire et les registres. Une faute est caractérisée par le nombre d'instructions depuis le démarrage de l'application.

LLVM Fault Injector (LLFI) [Thomas 2013, Lu 2015b], proposé en 2013, est un outil pour la tolérance aux fautes travaillant sur la représentation intermédiaire LLVM. Celui-ci instrumente le programme à analyser et génère un exécutable pour l'injection de fautes et un exécutable de profilage déterminant quelles fautes vont être injectées. LLFI modélise des inversions de un ou deux bits et vise une douzaine d'instruction comprenant le calcul d'adresse (`load` et `store`) ou encore les opérations arithmétiques (`mul`, `fmul` etc.). L'outil ne considère que les fautes *activées*, c'est-à-dire lorsque la donnée fautée est lue par le programme avant d'être écrasée par une autre instruction. Les auteurs constatent sur leurs expérimentations que certaines instructions (comme les instructions de comparaison) sont moins sensibles aux fautes que d'autres et que deux inversions augmentent le risque de crash sans augmenter le nombre d'erreurs de calcul non détectées.

REFINE [Georgakoudis 2017] est une approche basée sur l'injection de fautes à la compilation qui vise à modéliser plus finement des comportements qui sont difficiles à capter au niveau d'une représentation intermédiaire (comme par exemple les prologues / épilogues de fonctions qui sont abstraits par l'IR LLVM). REFINE se situe dans le backend du compilateur et a ainsi accès aux instructions spécifiques (au prix de la portabilité). L'outil se place ainsi après toutes les passes d'optimisation afin d'éviter que le code injecté puisse être transformé par le compilateur. REFINE se concentre sur un modèle de mutation d'opcode et d'opérande, ignorant les opcodes invalides. Les auteurs comparent leur approche à celle de LLFI qui s'applique également au niveau de l'IR LLVM et PINFI (qu'ils ont modifiés) qui travaille au niveau binaire sur différents programmes de calcul haute performance (HPC). Ils indiquent que les performances et la précision de REFINE sont meilleures que celles de LLFI et égalent, voire dépassent dans certains cas, celles de PINFI.

La table 2.2 présente une comparaison de quelques outils de type SWIFI. La colonne *Niveau* indique le niveau d'abstraction auquel se place l'outil. La colonne *Technique* correspond à la méthode utilisée pour émuler les fautes au niveau logiciel. La colonne *Modèle* précise les modèles de faute supportés et la colonne *Faute* la limite de faute pour les attaques. La colonne *Cible* correspond au type de système étudié par l'outil (système distribué, application etc.). La colonne *Oracle* indique le type de propriété de sécurité ou de sûreté étudiée par l'outil. Ici *SDC* fait référence aux corruptions silencieuses de données. Enfin, la colonne *FI* détermine si l'outil vise le milieu des attaques par injection de fautes ou celui des fautes accidentelles (ici tous).

Outil	Niveau	Technique	Modèle	Fautes	Cible	Oracle	FI
FIAT [Segall 1988]	Physique	materiel, OS spécialisé	mémoire registres communication	1	système distribué	SDC	Non
Ferrari [Kanawati 1992]	Physique	software-trap (syscall) materiel spécialisé	mémoire, registre remplacement inst. fautes permanentes	1	système distribué	SDC	Non
DEFINE [Kao 1996]	UNIX	kernel modifié interrupt modifiées	CPU (ALU, cache, decoder, reg, bus), communication fautes de design	1	système distribué	SDC	Non
Xception [Carreira 1998]	CPU	fonctionnalité CPU	flips, sa1, sa0, bit mask CPU registre bus	1	application	SDC	Non
EXFI [Benso 1998]	µkernel	software-trap	bitflip (mem / reg)	1	application	SDC	Non
LLFI [Thomas 2013]	LLVM	compilation	flip ALU store / load	1	application	SDC	Non
REFINE [Georgakoudis 2017]	LLVM	compilation	bit-flip sur opérandes	1	application (HPC)	SE	Non

TABLE 2.2 – Comparaison de quelques outils de type SWIFI

La figure 2.10 résume les avantages et inconvénients des outils SWIFI comparés aux méthodes d'injection physiques. L'injection logicielle est généralement moins coûteuse, plus facile à mettre en place et plus générique. L'observabilité est également supérieure avec les approches d'injection logicielle, qui peuvent tirer partie de certaines technologies comme les exceptions matérielle (EXFI) [Benso 1998].

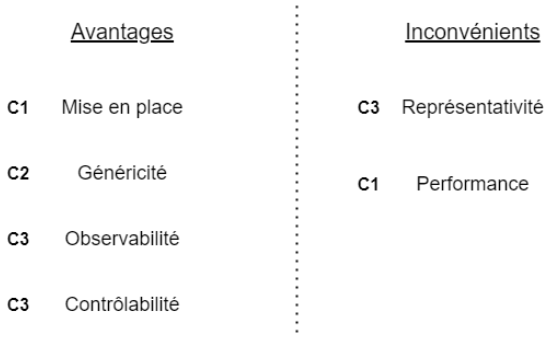

FIGURE 2.10 – Caractéristiques des méthodes SWIFI comparées à l'injection physique

La sur-couche logicielle pour émuler la faute offre une représentativité moindre par rapport à l'injection physique et peut aussi ajouter une surcharge qui diminue les performances. Cette perte de précision dépend de la technique d'injection logicielle utilisée et des optimisations ont été développées qu'il s'agisse de représentativité ou de performance [Lu 2015b, Georgakoudis 2017].

2.5.2.3 Outils basés sur la simulation

La simulation désigne les outils qui effectuent une exécution sur un système simulé, sans exécuter directement sur le matériel étudié. Les caractéristiques de l'outil dépendent alors en grande partie de la modélisation du système dans le simulateur.

Depend [Goswami 1997] est un simulateur qui vise à prendre en compte les interactions entre les différents composants d'un système complexe en cas de fautes accidentelles. L'outil modélise le système en C++ afin de simuler des fautes au niveau des portes logiques. Depend est capable de simuler des fautes sur la mémoire (bit-flip) et dans les communications (paquet perdu, corruption...). Les fautes sont décrites par l'utilisateur en C++ également.

SINJECT [Zarandi 2003] est un outil qui analyse la robustesse face aux fautes accidentelles au niveau de la représentation VHDL et Verilog. Il permet d'injecter un large nombre de fautes physiques au niveau des données et du circuit. Ceux-ci sont introduits dans le système Verilog à partir d'une description utilisateur. L'injection en VHDL est faite avec la technique mutant-saboteurs. L'outil profite du mode combiné VHDL/Verilog permettant de profiter de la représentativité de Verilog et de la description VHDL. SINJECT s'intéresse à des objectifs d'attaque de tolérance aux fautes sur les traces simulées.

Prototype of Another Fault Injector (PAFI) [Faurax 2006, Faurax 2009] est un outil étudiant les fautes au niveau du circuit utilisant la modélisation Verilog et VHDL. Il vise à vérifier l'absence de fuite de données secrètes dans le cadre d'attaques en multi-fautes au niveau circuit. L'outil s'appuie sur le simulateur Cadence NCSim (un outil propriétaire de STMicroelectronics) et utilise le fichier de commandes du simulateur pour contrôler les fautes à injecter. Une pondération basée sur des critères définis en fonction du modèle de faute permet de sélectionner les fautes à simuler (bit-flip sur les bascules ou faute sur les délais dans le circuit), ce qui permet un meilleur passage à l'échelle en multi-fautes. Par ailleurs, l'outil prend en compte les mécanismes de détection et indique les faux-positifs, faux-négatifs, attaques non détectées et attaques détectées. L'outil se veut extensible et générique en proposant un moyen de définir le modèle de circuit et les modèles de faute.

Celtic [Dureuil 2016a, Werner 2022] est un simulateur au niveau binaire développé en C++ qui permet de simuler des fautes au niveau de l'ISA et de l'architecture. Celtic supporte les fautes comme le saut d'instruction, la corruption du cache d'instruction [Rivière 2015] et les attaques sur les données sur les registres, la mémoire et le décodage des instructions. L'outil supporte également les fautes multiples, ceci étant facilité par la parallélisation des simulations. L'objectif d'attaque est exprimé dans le programme et évalué pour chaque trace simulée afin de récupérer les attaques réussies. Celtic propose un langage de spécification d'architecture (GISL) permettant d'adapter facilement l'outil à différentes architectures.

Outil	Niveau	Implem	Simu type	Modèle	Fautes	Cible	Oracle	FI
FOCUS [Choi 1992]	Physique	VLSI	externe	porte logiques	X	systèmes distribués	SDC	Non
Depend [Goswami 1997]	Circuit µ-arch	C++	intégrée	circuit communication mem.	1	systèmes distribués	SDC	Non
SINJECT [Zarandi 2003]	Verilog / VHDL	Verilog / VHDL	intégrée	circuit	1	application	SDC	Non
PAFI [Faurax 2006]	Circuit	Verilog / VHDL	externe	bit-flip délais	X	Verilog / VHDL	SC exploitabilité blocage	Oui
CELTIC [Werner 2022]	x86	C++	intégrée	data (reg, mem) cache, saut, rejeu	X	x86	oracle logiciel	Oui

TABLE 2.3 – Comparaison de quelques outils de simulation

La table 2.3 présente une comparaison de plusieurs outils de simulation. Les colonnes *Niveau*, *Modèle*, *Cible*, *Oracle* et *FI* ont la même signification que pour la table des outils SWIFI. La colonne *Fautes* indique la limite de faute, le symbole "X" indiquant le support des fautes multiples. La colonne *Implem* correspond au niveau auquel l'outil est implémenté (ce qui peut être différent du niveau de représentation étudié par l'outil). La colonne *Simu type* indique si l'injection de fautes est intégrée au simulateur ou si les fautes sont émulées pour un simulateur existant.

La figure 2.11 présente les avantages et inconvénients de la simulation par rapport aux méthodes d'analyse présentées précédemment. En fonction de la modélisation du système par le simulateur, la généricité de l'outil et la représentativité des résultats peuvent être très variables. La généricité de l'approche est en relation directe avec le niveau de représentation

du simulateur mais dépend aussi des choix du simulateur, GISL permettant par exemple une portabilité d'architecture pour CELTIC. Le développement d'un simulateur pour un système donné est le plus généralement coûteux par rapport à l'injection logicielle, mais le simulateur peut être disponible avant le matériel simulé ce qui a un intérêt dans le cycle de développement. Les performances sont souvent moins bonnes qu'avec les approches citées précédemment.

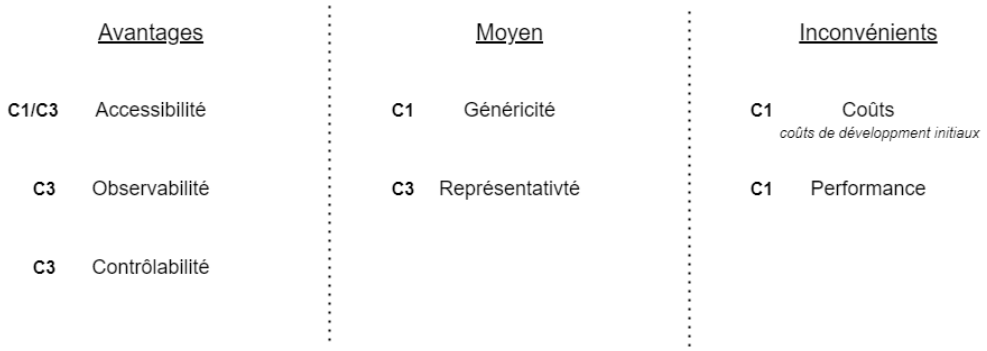

FIGURE 2.11 – Comparaison des outils de simulation par rapport aux méthodes précédentes

On peut également opposer les simulateurs qui prennent en compte les fautes (comme Depend ou Celtic par exemple) et les implémentations qui s'appuient sur un simulateur existant et suivent alors le modèle SWIFI en reproduisant l'effet d'une faute dans le modèle en question (tels que PAFI). L'observabilité et le contrôle de la description des fautes est souvent bonne pour ce type de techniques puisque l'accès au système est plus facile sur un simulateur que sur le matériel et le simulateur peut accéder à des parties plus bas niveau du système que pour l'injection logicielle.

2.5.3 Méthodes formelles

Les outils cités précédemment, à quelques exceptions près (comme Depend), comparent les exécutions fautées obtenues à une exécution de référence (*golden run*). Pour chaque expérimentation ou simulation, les entrées du programme ou du système sont fixées ainsi que les fautes devant être injectées. Ces outils tentent de couvrir le maximum d'attaques en effectuant une série de tests avec différents paramètres pour les entrées et les fautes.

Si différentes solutions ont été développées parmi ces outils basés sur les tests, comme réduire l'espace de faute [Benso 1998], sélectionner un seul représentant pour une attaque [Schmidt 2007] ou bien encore le développement de modèle plus haut niveau faisant abstraction de certains comportements, la couverture de ces outils est encore loin d'être parfaite. C'est pourquoi des solutions basées sur les méthodes formelles ont émergées afin d'obtenir une plus grande couverture des exécutions (des entrées et des fautes) ou bien de prouver des propriétés générales sur un programme ou un circuit, pour toutes fautes et pour toutes les entrées d'un modèle donné.

La figure 2.12 est un rappel du plan de la section 2.5 mettant en évidence les approches basées sur les méthodes formelles. Ces outils basés peuvent parfois être rapprochés des techniques de simulation, dans le sens où le système n'est pas exécuté directement sur le matériel, mais analysé statiquement. C'est le cas pour l'exécution symbolique où le moteur d'exécution simule une exécution (ou plus exactement un ensemble d'exécutions) dans un modèle donné. De la même manière, certaines approches se rapprochent des méthodes de types SWIFI, lorsque l'effet des fautes est mis en place dans le programme ou sa représen-

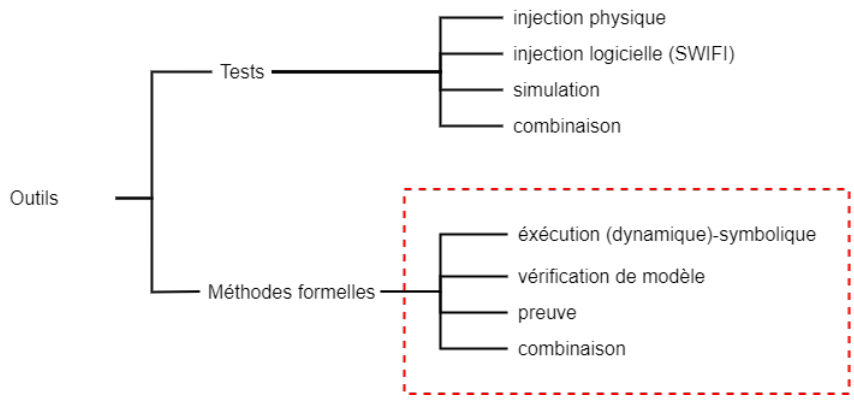

FIGURE 2.12 – Outils basés sur les méthodes formelles

tation, afin d'être fournies à un outil d'analyse formelle n'ayant pas de représentation pour les fautes [Potet 2014, Le 2018b]. Les analyses formelles peuvent être appliquées à différents niveaux de représentation, en allant du niveau source aux représentations bas niveau comme le RTL ou la spécification du circuit par exemple.

En 2007, Larsson et al. [Larsson 2007] proposent une méthode appelée *symbolic fault injection*. Ils présentent un outil d'analyse dans le cadre de la tolérance au fautes d'un programme source en Java et qui se base sur l'analyse formelle et l'exécution symbolique. Une instruction spécifique `inject(location);` est utilisée afin de spécifier que la zone mémoire `location` (variable, attribut etc.) peut être fautée. Leur implémentation utilise un modèle de bit flip sur les données et repose sur l'outil Key [Ahrendt 2005], une plateforme d'analyse formelle pour Java qui propose des fonctionnalités pour la vérification statique étendue, l'exécution symbolique, la vérification déductive et la spécification formelle. Des règles spécifiques ont été ajoutées dans Key pour simuler l'effet d'une faute dans l'état symbolique du programme. L'outil n'est cependant pas totalement automatique puisque l'utilisateur est sollicité pour spécifier les invariants de boucles par exemple.

SymplFIED [Pattabiraman 2008], est un outil d'évaluation reposant sur l'exécution symbolique et le model checking [Pattabiraman 2012]. L'outil prend en entrée un modèle de faute et un programme en assembleur qu'il transforme en un langage assembleur générique de type Reduced Instruction Set Computer (RISC). Cette représentation est implémentée avec le système Maude [Clavel 1996, Clavel 2002] qui est un outil permettant la vérification de propriété sur un modèle et qui se base sur la *rewritting logic*. SymplFIED supporte les fautes transitoires de un ou plusieurs bits dans le décodeur d'instruction, l'unité de calcul, les bus d'adresses et de données ou encore le fetch d'instruction, en faute unique. L'outil définit une procédure de modélisation de chaque type de faute sur le modèle machine dans Maude ainsi que des règles de propagation des erreurs entre les différents modules modélisés. Une erreur est représentée par un symbole `err` qui est propagé au fur et à mesure de l'exécution symbolique du programme. En plus de cela, l'outil vise à énumérer les cas d'erreurs qui ne sont pas détectées par des *détecteurs*. La modélisation dans Maude fait quelques hypothèses comme l'absence d'erreur dans les détecteurs ou le fait que tout accès à une mémoire non initialisée donnera lieu à une exception et certaines opérations comme la multiplication ne sont pas exécutées symboliquement [Pattabiraman 2012].

Lazart [Potet 2014] est un outil d'analyse de robustesse de programme sur la plateforme LLVM qui repose sur l'exécution symbolique effectuée par l'outil KLEE [Cadar 2008a]. La-

zart propose le modèle de l'inversion de test et chaque faute possible dans le programme est représentée par un booléen symbolique ce qui permet de générer les différents chemins d'attaque. L'objectif d'attaque est défini à l'aide d'une propriété d'atteignabilité (par exemple le bloc d'authentification dans le programme `verify_pin`) et les auteurs utilisent un algorithme de coloration de graphe pour réduire l'espace des fautes injectées (en ne considérant que les fautes sur les branches qui peuvent atteindre le bloc de base visé). La dernière version (version 4) de Lazart est l'objet des chapitres 3 et 4.

En 2018, Le et al [Le 2018b] proposent aussi un outil reposant sur l'exécution symbolique au niveau LLVM utilisant KLEE. En ce sens, il ressemble à Lazart à la différence qu'il se concentre sur la tolérance aux fautes, c'est-à-dire avec un modèle bit-flip en simple faute qui vise le registre de calcul virtuel de LLVM ainsi que l'instruction `load`. Ils utilisent une version légèrement modifiée de KLEE dans la fonctionnalité permettant de démarrer l'exécution symbolique avec une *graine* de manière à ne considérer que les chemins dans lesquelles l'injection d'une faute est possible. L'outil propose d'activer et désactiver l'injection localement ou globalement à l'aide de fonctions fournies dans une bibliothèque C qui est liée au programme à analyser.

Outil	Niveau	FA	Mode	Modèle	Fautes	Oracle	Technique	Mode	PAE
Larsson et al. [Larsson 2007]	Source (Java)	Non	Interne	flip	1	propriété logique	MC + SE (Key)	Interne	Elevé
Symplified [Pattabiraman 2008]	ASM (MISP)	Non	Interne	data (decoder, bus, reg, mem)	1	SDC	rewriting logic + SE (Maude)	Interne	Elevé
Lazart [Potet 2014]	LLVM	Oui	Externe	TI	X	logiciel	DSE (KLEE)	Externe	variable
Le et al. [Le 2018b]	LLVM	Non	Externe	bit-flip (reg., op.)	1	logiciel	DSE (KLEE)	Externe	variable

TABLE 2.4 – Comparaison de quelques outils reposant sur l'analyse formelle

La table 2.4 présente une table de comparaison d'outils basés sur l'analyse formelle. Les colonnes *Niveau*, *FA*, *Modèle*, *Faute* et *Oracle* sont équivalentes à celles des tables précédentes. La colonne *Mode* précise si l'outil utilise de façon externe un outil d'analyse formelle en instrumentant le code à la manière d'un outil SWIFI. La colonne technique liste les méthodes formelles utilisées par l'outil. La colonne *PAE* correspond au passage à l'échelle de l'outil.

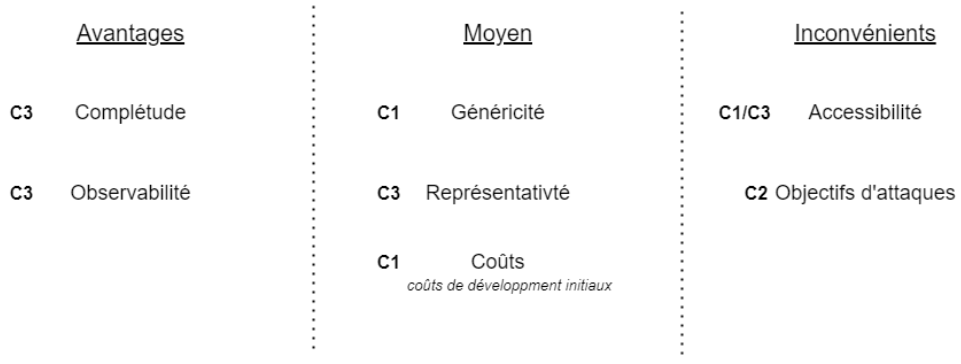

FIGURE 2.13 – Caractéristiques des méthodes formelles

La figure 2.11 présente les caractéristiques des outils basés sur les méthodes formelles. Les méthodes formelles ont un avantage en ce qui concerne la complétude des résultats, dépendant de la technique utilisée. Ces approches étant appliquées à des niveaux variés d'abstraction, et donc de modèles de faute, la représentativité des résultats est très variable.

La performance n'est pas réellement comparable par rapport aux outils basés sur les tests, puisque tous les chemins et toutes les combinaisons de fautes sont potentiellement explorés, mais certains outils souffrent d'un problème de passage à l'échelle, corrélé à la méthode formelle utilisée. L'accessibilité et l'automatisation ne sont pas toujours bons pour ces outils, demandant parfois une expertise de l'utilisateur dans la méthode formelle utilisée ou bien une intervention importante de la part de l'utilisateur [Larsson 2007].

2.5.4 Conclusion

Des approches très variées ont été proposées pour l'évaluation de la robustesse dans le cadre de l'injection de fautes, comme l'ont montré les sections précédentes.

Méthode	Mise-en-place	Représentativité	Reproductibilité	Observabilité
Injection Physique	Très élevé	Très élevé	Faible - Elevé	Faible
SWIFI	Faible	Faible - élevé	Moyen	Moyen
Simulation	Faible elevé	Moyen - élevé	Elevé	Elevé - Très élevé
Méthodes formelles	Faible - Elevé	Moyen - élevé	Moyen - elevé	Elevé - Très élevé

Méthode	Contexte	Complétude	Accessibilité	Généricité	Contexte
Inj Physique	Varié	Faible	Faible	Faible	Varié
Swifi	Varié	Faible	Moyen	Elevée	Varié
Simu	Varié	Faible	Moyen	Moyen - élevé	Varié
Formal	Limité	Elevé - Très élevé	Faible	Moyen - élevé	Limité

TABLE 2.5 – Comparaison des méthodes d'analyse de robustesse contre les fautes

La table 2.5 compare les caractéristiques de la section 2.5.1 pour les différentes techniques d'analyse présentées dans les sections précédentes. Chaque valeur correspond à une pondération entre *faible*, *moyen*, *élevé* et *très élevé*, visant à préserver les rapports entre les classes de techniques d'analyse ainsi que les variations au sein d'une même classe.

Les méthodes formelles ont un très net avantage en ce qui concerne la couverture des exécutions fautées, mais certaines techniques comme l'exécution symbolique souffrent de l'explosion combinatoire induite par les fautes pour le passage à l'échelle, en particulier dans un contexte multi-fautes. De plus, certaines propriétés et objectif d'attaque peuvent être difficiles à exprimer ou à analyser avec certaines approches formelles, notamment les propriétés basées sur la comparaisons de traces (canaux-auxiliaires) qui peuvent être plus simples à évaluer avec un golden run. La simulation offre des avantages en termes de contrôlabilité. Les méthodes de type SWIFI sont généralement les plus faciles à mettre en place.

Certaines caractéristiques dépendent fortement des choix de conception et d'implémentation de l'outil. Si la contrôlabilité est difficile dans le cadre de l'injection physique, cette caractéristique dépend surtout de ce que l'outil met à disposition à l'utilisateur pour les autres classes d'outils, les outils simulant un système (que ce soit dans un simulateur ou par une modélisation formelle) étant aussi avantagés. L'accessibilité est un bon exemple de caractéristique dépendant en grande partie de l'outil plus que de la méthode utilisée, qu'il s'agisse d'automatisation, de contrôlabilité ou d'expertise.

D'autres caractéristiques dépendent aussi fortement du niveau de représentation considéré par l'outil, qui peut être très variable au sein des outils d'une même méthode d'analyse. Dans ce cas, le niveau de représentation de l'outil influe directement sur la représentativité, les modèles de faute supportés et la généricité de l'outil.

L'outil Lazart, qui est l'objet du chapitre suivant, est un outil d'analyse formelle utilisant l'exécution dynamique-symbolique. Lazart utilise l'outil KLEE qui effectue une exécution

dynamique-symbolique sur le code LLVM et peut ainsi être vu comme un simulateur utilisant un modèle mémoire et une représentation du système particuliers. Les fautes sont introduites par une mutation du programme fourni à KLEE, correspondant à ce qui existe dans le cadre des outils SWIFI au niveau LLVM.

Lazart vise à répondre à certaines problématiques tirées de cette section en ce qui concerne ce type d'outils reposant sur les méthodes formelle ainsi que l'émulation des fautes au niveau logiciel.

— *Performance* et *passage à l'échelle* : maîtriser le multi-fautes et l'explosion combinatoire engendrée.
— *Représentativité* : limiter la perte de représentativité liée à la sur-couche de l'émulation des fautes.
— *Accessibilité* et *généricité* : limiter le niveau d'expertise requis pour l'utilisation de l'outil et laisser une contrôlabilité élevée à l'utilisateur.

Les contributions de Lazart visent ainsi à proposer des solutions pour ces problématiques :

— Support de l'inter-procédural et de la combinaison de modèles de faute ainsi qu'une description à grain fin des fautes par l'utilisateur (*représentativité* et *contrôlabilité*).
— Une Application Programming Interface (API) Python facilitant la description des analyses et la présentation des résultats (*accessibilité*).
— Une analyse plus fine des résultats produits.

Chapitre 3

Lazart

Lazart [Potet 2014] est un outil d'analyse de code haut niveau qui repose sur l'exécution symbolique et vise à évaluer la robustesse d'un programme dans le cadre de l'injection de fautes multiples. Cet outil a été développé à Verimag depuis 2014 et a été étendu au cours de cette thèse. Ce chapitre présente l'outil Lazart, dans sa version la plus récente, en décrivant son fonctionnement et son architecture du point de vue de l'utilisateur. Il s'agira aussi de discuter de ses avantages et limitations par rapport aux outils présentés dans le chapitre précédant. Le chapitre 4 rentrera plus en détails dans l'implémentation de Lazart et discutera des choix qui ont été faits pour l'outil.

La suite de ce chapitre est organisée comme suit. La section 3.1 présente le principe général de l'outil, la plateforme LLVM et l'exécution concolique sur lesquels Lazart s'appuie. La section 3.2 décrit l'architecture de l'outil et les différentes étapes d'une analyse. La section 3.3 présente la façon dont l'utilisateur décrit les paramètres d'une analyse avec Lazart. La section 3.4 s'intéresse à la représentation des traces dans Lazart ainsi qu'aux analyses de robustesse proposées par l'outil. La section 3.5 s'intéresse aux résultats fournis par l'outil, leur exploitation et à l'aspect méthodologique en présentant des évaluations réalisées avec Lazart. Finalement, la section 3.6 conclut ce chapitre.

Table des Matières

3.1 Principe général et pré-requis

Lazart est un outil d'analyse de code évaluant la robustesse d'un programme dans le cadre d'injection de fautes multiples. Il travaille au niveau de la représentation intermédiaire LLVM-Intermediate Representation (LLVM-IR) et repose sur l'exécution concolique [Baldoni 2018] (pour concret-symbolique), qui combine l'exécution symbolique avec l'exécution concrète. Lazart est destiné à être utilisé en amont du processus de développement pour aider le développeur à concevoir un programme sécurisé, ou en aval, en apportant une aide aux auditeurs pour en déterminer les vulnérabilités [ANSSI 2020]. Il propose plusieurs analyses visant à évaluer la robustesse d'un programme dans le cadre de l'injection de fautes. L'outil est également destiné à l'évaluation de contre-mesures, cet aspect de l'outil sera présenté plus en détail dans les chapitre 5 et 6.

Limite de faute	0	1	2	3	4
Attaques	0	1	5	10	11

TABLE 3.1 – Résultat d'analyse d'attaque sur verify_pin avec Lazart

L'analyse d'attaque permet de rechercher des chemins d'attaques dans un programme, en fonction d'un modèle d'attaquant (objectif d'attaque et modèle de faute). Cette analyse retourne la liste des chemins d'exécution fautés trouvés pour lesquels l'objectif d'attaque est atteint. La table 3.1 correspond aux résultats d'une analyse d'attaque pour le programme `verify_pin` avec le modèle de l'inversion de test, une limite de fautes de quatre et comme objectif d'attaque ϕ_{auth}, c'est-à-dire s'authentifier malgré un PIN faux. Chaque colonne indique le nombre de chemins d'attaques trouvés (deuxième ligne) pour un nombre de fautes fixé (première ligne).

Lazart émule les fautes au niveau de la représentation intermédiaire LLVM et transmet le programme muté à l'exécution concolique de manière à explorer tous les chemins avec toutes les combinaisons de fautes possibles. La section 3.1.1 présente la plateforme LLVM et la représentation intermédiaire LLVM-IR, niveau auquel sont représentées les fautes. La section 3.1.2 présente le fonctionnement de l'exécution symbolique et la section 3.1.3 décrit l'exécution concolique et les problématiques liées à ce mode de génération de traces. La section 3.1.4 présente plus en détail l'outil KLEE et ses fonctionnalités et finalement, la section 3.1.5 s'intéresse à la façon dont les fautes sont simulées dans une implémentation pour le moteur d'exécution symbolique.

3.1.1 Low Level Virtual Machine (LLVM)

Lazart repose sur la plateforme LLVM [Lattner 2004], qui est un ensemble d'outils pour la conception de compilateurs, d'optimiseurs et d'outils d'analyse statique. LLVM est destiné à faciliter l'analyse et l'optimisation de programmes tout au long de la durée de vie d'un programme (*lifelong program analysis and transformation*).

L'architecture LLVM (figure 3.1) s'organise autour de trois blocs majeurs : le front-end, la représentation intermédiaire et le back-end.

Le *front-end* compile le code source du programme en la *représentation intermédiaire* qui est transformée en code binaire spécifique à l'architecture cible par le *back-end*. Clang

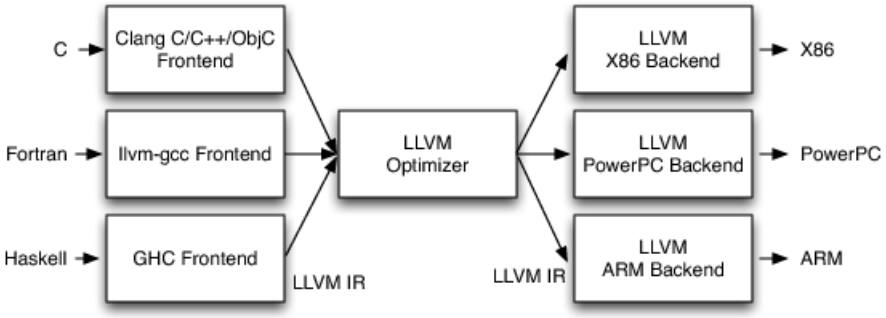

FIGURE 3.1 – Architecture d'un compilateur en trois temps [Amy Brown 2011]

[Lattner 2004] est certainement le front-end de compilateur le plus célèbre de LLVM et permet la compilation des langages C, C++, Objective-C et Objective-C++. LLVM supporte d'autres langages, tels que Rust, Swift, Haskell, D et Ada pour n'en citer que quelques uns, au travers d'autres front-end.

Les analyses et optimisations sont effectuées sur la représentation intermédiaire LLVM-IR. La figure 3.1 présente le cycle de vie d'un programme dans le cadre d'une compilation Ahead Of Time (AOT), c'est-à-dire lorsque le processus d'optimisation et d'analyse a lieu en amont de l'exécution du programme. L'architecture de LLVM permet de simplifier l'optimisation inter-procédurale (*link-time optimization*) en gardant la même représentation intermédiaire tout au long du cycle de vie. LLVM prend aussi en charge d'autres cycles de vie de programmes tels que :

- *Install Time Optimisation* : le programme est optimisé au moment de son installation sur l'appareil. Il s'agit d'un type particulier de compilation AOT [1] qui permet d'avoir des informations exactes sur l'architecture et la machine cible.
- Just In Time (JIT) optimisation : le programme est compilé et optimisé pendant son exécution. Le JIT est très largement utilisé pour les langages basés sur une machine virtuelle (Java, C# par exemple).
- *Idle Time Optimisation* : le programme peut-être optimisé entre les exécutions.

La représentation intermédiaire de LLVM est ainsi manipulée par des passes d'analyse et d'optimisation tout au long de la durée de vie du programme. LLVM-IR est un langage correspondant à une abstraction des langages assembleur. Sa forme binaire (utilisée en mémoire pendant les opérations ou stockée sous la forme de fichier) peut être transformée en équivalent textuel (et inversement). Le listing 3.1 présente un exemple de code `Hello world` en LLVM-IR sous forme textuelle.

```
1  @.str = private unnamed_addr constant [14 x i8] c"Hello world !\00", align 1
2
3  define dso_local i32 @main(i32 %0, i8** %1) {
4    %3 = alloca i32, align 4
5    %4 = alloca i32, align 4
6    %5 = alloca i8**, align 8
7    store i32 0, i32* %3, align 4
8    store i32 %0, i32* %4, align 4
9    store i8** %1, i8*** %5, align 8
```

1. C'est par exemple la méthode employée sur Android avec la machine virtuelle Dalvik (qui n'est pas basée sur LLVM).

```
10    %6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str
          , i64 0, i64 0))
11    ret i32 0
12  }
13
14  declare dso_local i32 @printf(i8*, ...)
```

Listing 3.1 – Programme *Hello World!* en IR LLVM (LLVM-9)

LLVM-IR rend explicite le flot de contrôle et de données. Toute instruction appartient à un bloc de base et tout bloc de base appartient à une fonction. Le langage suit la forme Single Static Assignement (SSA) [Rosen 1988] [2] qui impose que chaque variable soit assignée avant toute utilisation et que chaque variable soit assignée exactement une fois. LLVM-IR utilise un nombre infini de registres typés, identifiés par les $\%XX$ dans le programme d'exemple. Les conventions d'appels de fonctions sont abstraites à l'aide des instructions `ret` et `call`.

Le langage utilise un jeu d'instructions comparable à RISC et est fortement typé. Toutes les conversions sont explicites, à l'aide notamment des instructions de conversion (`cast`, `trunc`, `zext` ...) et de l'instruction `getelementptr` qui permet d'abstraire les opérations d'arithmétique de pointeur tout en préservant le typage (c'est-à-dire que `getelementptr` effectue les calculs appropriés au type sous-jacent des opérandes). La ligne 10 du programme 3.1 correspond à l'appel de la fonction C standard `printf` et l'accès à la chaîne de caractère globale `@.str` utilise l'instruction `getelementptr` [3].

Les instructions LLVM-IR peuvent être utilisées de façon générique pour chaque type de donnée de base et existent en plusieurs versions. Par exemple, l'instruction arithmétique `add` est la même pour les différents types entiers ou flottant et supporte différentes opérandes (par exemple deux registres `add i16 %var1, %var2` ou bien avec une valeur immédiate `add i64 %var, i64 10`).

L'allocation de mémoire fait la distinction entre la pile et le tas. Dans l'exemple *Hello world* ci-dessus, une instruction `alloca` est utilisée pour allouer un espace mémoire sur la pile pour chaque argument et variable locale de la fonction `main` (`%3`, `%4` et `%5`). L'allocation sur le tas est effectuée avec l'appel des primitives (par exemple `malloc` et `free`). La lecture et l'écriture en mémoire sont faites respectivement avec les instructions `load` et `store`. Dans l'exemple 3.1, chaque opération précise l'alignement des données avec le mot clef `align`.

La plateforme LLVM est largement utilisée dans le monde de la recherche et de l'industrie et la page LLVM [LLVM 2021] recense plus de 250 publications liées à la plateforme. Le projet LLVM sur Github [LLVM 2011] compte plus de 4000 forks et 2000 contributeurs.

3.1.2 Principe de l'exécution symbolique

L'exécution symbolique est une technique d'analyse de programme qui a été théorisée dans les années 70 [Boyer 1975, King 1976, Clarke 1976] et qui vise à générer des entrées de tests pour couvrir chaque partie d'un programme. Le principe consiste à parcourir le programme en utilisant des *variables symboliques* plutôt que des valeurs concrètes pour les entrées, et de résoudre la formule logique associée à chaque chemin d'exécution. L'exécution *dynamique-symbolique* regroupe un ensemble de techniques se basant sur une combinaison d'une exécution symbolique et d'exécution(s) concrète(s) dans le but d'adresser certaines problématiques liées à l'exécution symbolique (section 3.1.3).

2. L'instruction `phi` correspondant à la forme standard (non-gated, c'est-à-dire que l'assignation conditionnelle n'est évaluée qu'à partir d'une variable) de la fonction ϕ de SSA.

3. le mot clef `inbound` est utilisé (ligne 10) pour apporter des garanties sur l'accès à la mémoire pour les phases d'analyse.

Pour la fonction `exemple` en Figure 3.2, un chemin d'exécution possible consiste à passer dans la branche *then* de la première condition et atteindre l'assertion dans la branche `else` ligne 9. Un ensemble d'entrées possible pour arriver à cet état est $\{x = 1, y = 3\}$. On peut alors chercher s'il existe un ensemble d'entrées invalidant l'assertion, comme par exemple $\{x = 0, y = 3\}$. L'objectif d'un outil d'exécution symbolique est d'obtenir une couverture maximale des chemins du programme et de fournir des valeurs concrètes pour les entrées permettant d'exécuter chaque chemin.

```
1  void exemple(int x, int y) {
2      if (x + y > 2) {
3          z = 2*x + y;
4          if (z == 20) {
5              x = y;
6              print(x);
7          }
8          else {
9              y *= 2;
10             assert(x != 0);
11         }
12     }
13     else {
14         foo();
15     }
16 }
17
```

Listing 3.2 – La fonction exemple

Le moteur d'exécution symbolique maintient un *état symbolique* et un *prédicat de chemin*. L'*état symbolique* σ associe aux variables une expression symbolique qui est mise à jour à chaque fois qu'une affectation est rencontrée. Pour une affectation $var = expr$, σ est mis à jour en associant var à $\sigma(expr)$. La figure 3.3 présente le code de la fonction *exemple* avec l'évolution de l'état symbolique à chaque assignation dans le programme.

```
1  void exemple(int x, int y) {      σ = { x → x0, y → y0 }
2      if (x + y > 2) {
3          z = 2*x + y;              σ = { z → 2*x0 + y0, x → x0, y → y0 }
4          if (z == 20) {
5              x = z + 3;            σ = { z → 2*x0 + y0, x → 2*x0 + y0 + 3, y → y0 }
6              print(x);
7          }
8          else {
9              y *= 2;               σ = { z → 2*x0 + y0, x → x0, y → 2*y0 }
10             assert(x != 0);
11         }
12     }
13     else {
14         foo();
15     }
16 }
17
```

Listing 3.3 – Etats symboliques pour la fonction exemple

Le *prédicat de chemin* (Path Constraint (PC)) est une formule logique du premier ordre qui est mise à jour à chaque fois qu'un branchement conditionnel est rencontré. La fonction *exemple* contient donc cinq prédicats de chemin, présentés dans le listing 3.4, dont les valeurs sont :

$$PC_0 \equiv true \tag{3.1}$$
$$PC_1 \equiv x_0 + y_0 > 2 \tag{3.2}$$
$$PC_2 \equiv x_0 + y_0 > 2 \wedge 2 * x_0 + y_0 = 20 \tag{3.3}$$
$$PC_3 \equiv x_0 + y_0 > 2 \wedge 2 * x_0 + y_0 \neq 20 \tag{3.4}$$
$$PC_4 \equiv x_0 + y_0 <= 2 \tag{3.5}$$

```
1  void exemple(int x, int y) {    // PC0
2      if (x + y > 2) {            // PC1
3          z = 2*x + y;
4          if (z == 20) {          // PC2
5              x = y;
6              print(x);
7          }
8          else {                  // PC3
9              y *= 2;
10             assert(x != 0);
11         }
12     }
13     else {                      // PC4
14         foo();
15     }
16 }
17
```

Listing 3.4 – La fonction exemple

Lorsqu'un branchement $if(expr)$ est rencontré, le prédicat de chemin est mis à jour tel que $PC_{then} = PC \wedge \sigma(expr)$ pour la branche vraie, et $PC_{else} = PC \wedge \neg\sigma(expr)$ pour la branche fausse. S'il existe une solution qui satisfait PC_{then}, l'exécution symbolique clone l'état symbolique σ et continue avec le prédicat de chemin PC_{then} dans la branche *then*, et respectivement pour la branche *else*.

La résolution des prédicats de chemins est automatisée avec des solveurs de contraintes. Les solveurs particulièrement utilisés dans ce contexte sont les solveurs Satisfiability Modulo Theory (SMT) [De Moura 2011, Barrett 2018] qui permettent de prouver la satisfaisabilité de certaines classes de formules logiques du premier ordre sans quantificateur. Le solveur de contraintes prend en entrée une formule logique et peut indiquer si elle est satisfaisable (en exhibant une valuation des variables correspondantes), si elle est non satisfaisable (il n'existe aucune solution), ou bien ne pas conclure dans le temps imparti (*timeout*).

L'évolution des solveurs SMT au cours de ces dernières décennies a permis de rendre réaliste l'usage des techniques d'exécution symbolique. Parmi les principales théories supportées on trouve l'arithmétique linéaire ou non linéaire (sur entiers ou réels), les tableaux, les vecteurs de bits, etc.

On peut s'intéresser à la correction et la complétude des prédicats de chemins [Godefroid 2011].

Définition 3.1. *Un prédicat de chemin PC_ω est correct si tout modèle satisfaisant PC_ω fournit des entrées pour une exécution du programme suivant le chemin ω.*

Définition 3.2. *Un prédicat de chemin PC_ω est complet si toute entrée suivant le chemin ω est un modèle validant PC_ω.*

3.1.3 Exécution dynamique-symbolique (DSE)

L'exécution *dynamique-symbolique* (Dynamic Symbolic Execution (DSE)) consiste à combiner l'exécution symbolique avec l'exécution concrète. Cela permet de résoudre en partie certains des problèmes liés à l'exécution symbolique, potentiellement au prix d'une perte de correction et de complétude lorsqu'une partie de l'analyse est concrétisée.

Les premières approches sont apparues dans les années 2000 avec notamment le *test concolique* introduit dans l'outil Directed Automated Random Testing (DART) [Godefroid 2005]. DART propose de maintenir un état concret parallèlement à l'état symbolique. Cette approche nécessite des entrées concrètes au démarrage de l'analyse, qui peuvent être générées aléatoirement ou fournies par l'utilisateur. L'état concret associe chaque variable du programme à une valeur et maintient le prédicat de chemin et l'état symbolique de manière à calculer à l'aide du solveur des valeurs concrètes pour chaque chemin. Cette concrétisation permet notamment d'exécuter des fonctions externes à l'aide de valeurs concrètes sans disposer de leur code.

EXE [Cadar 2008b] et son successeur KLEE [Cadar 2008a] utilisent une approche similaire que les auteurs nomment *execution-generated testing*, proposée initialement en 2005 [Cadar 2005], qui maintient également un état concret. Si une opération n'implique que des valeurs concrètes, celle-ci est exécutée directement, sinon, l'opération est effectuée symboliquement.

L'exécution dynamique-symbolique est limitée par un certain nombre de problématiques : la résolution des prédicats de chemin, le nombre de chemins générés, la modélisation du système et l'utilisation de code non disponible (comme une bibliothèque externe). La suite de cette section s'intéresse à ces problématiques et présente certaines approches qui ont été développées pour les pallier.

3.1.3.1 Complexité des formules passées au solveurs

Les solveurs de contraintes sont sensibles au type d'application considéré. Ils peuvent être limités si les théories supportées ne permettent pas (dans un temps raisonnable) de résoudre le prédicat de chemin ou bien si le prédicat de chemin contient des formules qui sont par nature difficiles à résoudre (comme inverser une fonction de hachage cryptographique par exemple).

Certaines techniques permettent cependant d'atténuer ces difficultés du côté du moteur d'exécution symbolique. Les formules logiques transmises aux solveurs peuvent être simplifiées ou optimisées par différentes techniques : réutilisation des résultats de résolution de contraintes (*incremental solving* [Sen 2005]), suppression des contraintes non liées au test qu'on veut valider ou inverser (*irrelevant constraint elimination* [Cadar 2008a]), simplification des expressions logiques passées au solveur, comme des simplifications linéaires ou des remplacements par des expressions équivalentes (par exemple transformer une puissance de 2 en un décalage de bit). La *constraint set simplification* [Cadar 2013] vise à simplifier les expressions du prédicat de chemin au fur et à mesure de l'exécution lorsque des contraintes plus précises sont rencontrées (par exemple $x_0 < 4 \wedge x_0 = 1 \rightarrow x_0 = 1$).

Outre la simplification des contraintes, le moteur d'exécution symbolique peut réduire le nombre d'appels au solveur par plusieurs techniques. Tout d'abord, le moteur peut continuer l'exploration sur certains chemins tandis que des requêtes sont en attente pour d'autres chemins. C'est par exemple ce que fait EXE qui utilise un serveur responsable de traiter les requêtes demandée par les différentes instantes du moteur d'exploration. Certains outils, tels que KLEE et Binsec [Djoudi 2015, List 2015], proposent d'utiliser plusieurs solveurs de contraintes. Ceux-ci peuvent être appelés en parallèle ou bien choisis à l'aide d'heuristiques. Des formats standards de requête aux solveurs tels que SMT-LIB [Barrett 2016] ont été

développés pour faciliter l'interchangeabilité des solveurs. La réutilisation des contraintes peut être faite en utilisant un cache des contraintes déjà envoyées au solveur : par exemple le *constraint caching* effectuant du hash consing sur les contraintes [Cadar 2013], ou bien le *counter example caching scheme* de KLEE, qui part du principe qu'en général l'ajout de contraintes n'invalide pas les solutions déjà trouvées. L'approche décrite dans [Yang 2012], appelée *memoized symbolic execution*, encode les préfixes des chemins d'exécution afin de les réutiliser. Green Framework [Visser 2012, Jia 2015] propose d'aller plus loin en proposant la réutilisation de contraintes entre des programmes et des analyses différentes. Dans les cas où les contraintes sont trop complexes pour le solveur, la concrétisation de certaines variables est une solution. Certaines concrétisation peuvent néanmoins faire perdre la complétude.

3.1.3.2 Explosion des chemins

L'exécution symbolique est aussi sensible à *l'explosion des chemins* (*path explosion*). Le nombre de chemins d'exécution d'un programme peut rapidement devenir grand, voir infini (lorsque le programme contient des boucles dépendantes des entrées - que ce soit des boucles directes telles que des *for*/*while*, ou par récursion). S'il n'est pas possible de borner statiquement le nombre d'itérations, alors l'analyse doit se restreindre à une couverture partielle des chemins d'exécution.

Pour pallier la problématique d'explosion des chemins, plusieurs heuristiques peuvent être mises en place en plus de borner le nombre d'itérations ou la profondeur d'une exécution [Cadar 2008a, Godefroid 2008, Schwartz 2010, Jamrozik 2013]. Ces heuristiques visent généralement en priorité la couverture de code (que ce soit au niveau des instructions ou des branches). Celles-ci comptent notamment la sélection aléatoire de chemins [Cadar 2013], le choix du chemin se rapprochant le plus d'une instruction non couverte, la sélection du chemin le plus court (*shortest distance symbolic execution (SDSE)* [Chipounov 2012]) ou la sélection des chemins couverts le moins de fois. Certaines stratégies peuvent combiner plusieurs heuristiques (comme *cov new* de KLEE) ou utiliser d'autres techniques comme le fuzzing [Majumdar 2007, Stephens 2016, Yun 2018].

La combinaison avec d'autres méthodes d'analyse peut également aider dans la réduction du nombre de chemins à explorer. Certaines fonctions peuvent être analysées indépendamment, de manière à générer des pré-conditions et post-conditions sur les entrées et sorties afin d'être ensuite réutilisées lors de l'appel de ces fonctions [Godefroid 2011]. Triton [Salwan 2020] propose aussi une option pour ne rendre symbolique que les espaces mémoires associés à une valeur teintée, la teinte étant propagée pendant l'exécution à partir des teintes initiales spécifiées par l'utilisateur.

3.1.3.3 Modélisation de l'architecture

La modélisation de l'architecture cible est également un problème important dans le cadre de l'exécution concolique et de l'analyse statique plus généralement. Un modèle plus précis permet des résultats plus représentatifs mais peut être plus coûteux (notamment à cause d'une plus grande complexité des formules logiques passées aux solveurs).

Concernant la mémoire, différentes approchent existent. Certains outils considèrent toutes les variables d'un programme comme symboliques par défaut, comme c'est le cas pour Angr [Shoshitaishvili 2016] par exemple. D'autres comme KLEE nécessitent que l'utilisateur spécifie quelles sont les variables à rendre symboliques. La gestion de KLEE utilise un modèle mémoire composé de « blocs mémoires » qui sont des tableaux de vecteurs de bits mais toutes les adresses (pointeurs) sont concrètes [Cadar 2020]. DART utilise une approche similaire et concrétise un pointeur en prenant en compte le cas nul et le cas d'un objet complet. Binsec représente la mémoire comme un tableau symbolique [Djoudi 2015].

La prise en compte d'un environnement d'exécution complet (appels systèmes, bibliothèques externes, réseau...) est aussi une difficulté importante pour l'analyse statique d'un programme. Une solution consiste à interagir avec un environnement réel en concrétisant les paramètres au moment de l'appel. Cette approche est utilisée par EXE, DART et CUTE par exemple, mais celle-ci souffre de l'incomplétude de la concrétisation. L'interaction avec un environnement concret nécessite de prendre des précautions pour que les interactions d'une exploration de chemin n'aient pas un effet de bord sur un prochain chemin, pour cela des solutions basées sur la virtualisation ont été mises en place (par exemple S2E [Chipounov 2012] qui utilise QEMU). Une autre solution est de rendre symbolique tout retour d'un appel externe afin de prendre en compte tous les comportements, ce qui mène toutefois à une sur-approximation. KLEE propose une implémentation de la bibliothèque standard C (appelée μlibc) et qui peut être exécutée concrètement ou symboliquement pendant l'exécution concolique. Klover [Li 2011] supporte aussi des bibliothèques POSIX supplémentaires. Certaines approches nécessitent l'intervention de l'utilisateur pour décrire un modèle [Xiao 2011].

3.1.4 KLEE

KLEE [Cadar 2008a] est l'outil d'exécution symbolique utilisé dans Lazart. C'est un outil *open-source* [Klee 2008b] basé sur llvm-ir. En plus des caractéristiques déjà évoquées dans les sections précédentes, KLEE propose le paramétrage des stratégies d'exploration des chemins, le choix des solveurs à utiliser et l'accès aux formules SMT générées. L'outil supporte les solveurs STP [Ganesh 2002], boolector [Brummayer 2009], CVC4 [Barrett 2011], Yices 2 [Dutertre 2014a], Z3 [De Moura 2011] et meta SMT [Haedicke 2011], qui est un projet permettant de supporter plusieurs solveurs de façon générique.

KLEE a été utilisé pour analyser la bibliothèque COREUTILS [LLC 1999] (plus de 400k lignes de code) et a permis de trouver 56 bugs majeurs dont 3 qui étaient inconnus jusqu'alors [Cadar 2008a]. L'outil a fini second lors de la compétition TestComp 2019 [Cadar 2020] avec 1226 points contre 1238 pour l'outil Verifuzz [Basak 2019], en ayant presque aucun point dans la catégorie des nombres en virgule flottante puisque ceux-ci ne sont pas supportés par KLEE.

L'outil est largement utilisé pour différents types d'analyse de code (test fonctionnel, recherche de vulnérabilités, tolérance aux fautes, etc.) et la page [KLEE 2008a] recense plus de 100 publications liées à l'outil. Un certain nombre de projets ont été proposés à partir de KLEE. Cloud9 [Bucur 2011] est un projet open-source qui permet d'effectuer de l'exécution symbolique parallélisée avec KLEE, ce qui peut fortement améliorer les performances. Klover [Li 2011] est une extension de KLEE proposant un support complet du C++ et qui utilise un solveur spécialisé pour les chaînes de caractères. KleeNet [Sasnauskas 2010] est un outil basé sur KLEE visant la recherche de chemins d'erreur dans des systèmes distribués et les piles de protocoles réseaux.

Le listing 3.5 présente un exemple utilisation de KLEE contenant une fonction `foo` effectuant un branchement et le code d'analyse dans la fonction `main`. La fonction `klee_make_symbolic` (lignes 11 et 12) est utilisée pour déclarer une variable comme symbolique, la fonction prend en paramètre un pointeur vers la zone mémoire à rendre symbolique, sa taille et un nom (utilisé pour l'affichage des résultats). La fonction `klee_assume` permet de couper l'exploration des chemins qui ne valident pas le prédicat en argument. Dans l'exemple (ligne 16), l'exploration se restreint aux exécutions où le retour de *foo* est strictement positif. Cela permet d'encoder la vérification d'un objectif d'attaque par exemple.

```
1  int foo(int a, int b)
2  {
3      if(b == 0)
4          return a;
5      return a / b;
6  }
7
8  int main() {
9      int a;
10     int b;
11     klee_make_symbolic(&a, sizeof(a), "a"); // (memory pointer, size, name)
12     klee_make_symbolic(&b, sizeof(b), "b");
13
14     int result = foo(a, b);
15
16     klee_assume(result > 0); // attack objective.
17 }
18
```

Listing 3.5 – Exemple d'instrumentation du programme pour KLEE

KLEE fournit un ensemble de cas de tests (sous la forme de fichiers .ktest) qui correspondent chacun à un représentant d'un prédicat de chemin. Chaque cas de test est aussi associé à un type de terminaison qui peut être notamment :

- Une erreur d'exécution : adresse invalide lors d'une écriture ou lecture mémoire (*ptr*), arithmétiques (division par zéro (*div*), dépassement d'entier (*overflow*)), pointeur invalide fournit à `free` (*free*), violation d'une mémoire read-only (*rom*).
- Des comportement spécifiques : arrondi implicite (*trunc*), conversion implicite (*conv*).
- Une sortie secondaire : appel à la fonction standard `abort` ou à `klee_abort` (*abort*), assertion invalide (*assert*).
- Une erreur utilisateur (*user*) : un assume faux (`klee_assume`) ou une utilisation incorrecte de KLEE.
- Concrétisation forcée (*model*) : appel symbolique à une fonction externe ou allocation de mémoire de taille symbolique par exemple.
- Une terminaison normale (aucun `klee_assume` n'est violé) (*satisfy*).
- Un timeout : le chemin n'a pas pu être exploré dans le temps imparti (*timeout*).

KLEE propose par ailleurs une fonctionnalité de replay qui permet de rejouer une exécution à partir d'un ktest en utilisant les valeurs du représentant pour les variables symboliques. Cela permet d'observer les sorties de l'exécution de ce chemin (sortie console ou interaction avec des fichiers par exemple).

3.1.5 Exécution symbolique et injection de fautes

Comme dit au début de ce chapitre, Lazart se base sur l'exécution concolique et le parcours des chemins pour chercher des attaques en fautes multiples. Cette section explique comment les fautes sont émulées pour le moteur d'exécution concolique.

Le déclenchement potentiel d'un point d'injection est représenté à l'aide d'une variable symbolique interprétée comme un booléen, comme présenté dans la figure 3.2. Lorsque l'exécution symbolique rencontre un point d'injection, l'exécution est divisée en un chemin non fauté et un chemin fauté. La fonction `symbolic_bool` correspond à la génération d'une variable symbolique booléenne et la condition ligne 2 induit une division des chemins. Soit PC_0 le prédicat de chemin précédant le point d'injection, alors le chemin sans faute a pour prédicat de chemin $PC_{normal} \equiv PC_0 \wedge \neg inject$ et celui fauté $PC_{inject} \equiv PC_0 \wedge inject$. Dans cet exemple, les variables *_fault_ count* et *_fault_ limit* correspondent respectivement

au nombre de fautes injectées dans l'exécution en cours et à la limite de fautes. C'est en cela que l'approche utilisée par Lazart s'apparente aux outils de type SWIFI, présentés dans le chapitre 2.

```
1 normal_behavior()
2
3
4
5
6
7
```

Listing 3.6 – Comportement nominal

```
1 inject = symbolic_bool()
2 if inject and _fault_count <=
3         _fault_limit:
4     _fault_count++
5     faulted_behavior()
6 else:
7     normal_behavior()
```

Listing 3.7 – Comportement avec fautes

FIGURE 3.2 – Représentation des fautes en pseudo-code pour l'exécution symbolique

Ainsi l'exécution symbolique effectue un parcours de l'arbre d'exécution en y incluant toutes les exécutions fautées possibles. Cette disjonction des cas intensifie le phénomène d'*explosion des chemins*, tout particulièrement dans le cadre de fautes multiples. Les fautes peuvent aussi introduire des chemins qui ne terminent pas.

Illustrons cette explosion de chemins avec l'exemple de la fonction `fib` présentée dans le listing 3.8 et qui correspond à une implémentation récursive du calcul de la suite de Fibonacci. Dans une exécution non fautée, cette fonction est constituée d'un ensemble d'exécutions correspondant à l'appel récursif de la fonction `fib` dépendant de l'entier naturel n.

```
1 uint fib(uint n) {
2     if (n <= 1)
3         return n;
4     return fib(n-1) + fib(n-2); // IP1 et IP2
5 }
```

Listing 3.8 – Fonction `fib`

Si on considère un modèle de faute permettant de remplacer le retour d'un appel à `fib` par `0`, le programme contient deux points d'injection à la ligne 4, c'est-à-dire `fib(n-1)` (IP_1) et `fib(n-2)` (IP_2). Cela implique qu'à chaque appel à la fonction `fib(n)` dans une exécution normale, le chemin d'exécution se divise en cinq : le chemin retournant n, le chemin retournant `fib(n-1) + fib(n-2)` et les trois chemins fautés avec le déclenchement des points d'injection $\{IP_1\}$, $\{IP_2\}$ et $\{IP_1, IP_2\}$.

Pour une exécution avec une valeur d'entrée n fixée, l'exécution non fautée produit un unique chemin dont le nombre de blocs de base traversés dépend de n. Pour une exécution fautée, il existe 5^n chemins, pour un nombre de fautes non borné. L'explosion du nombre de chemins est donc renforcée par la combinatoire des fautes en particulier en fautes multiples (ce problème est lié à l'injection de fautes et n'est pas spécifique à l'exécution concolique).

Les définitions 3.1 et 3.2 présentées précédemment définissent la *correction* et la *complétude* d'un prédicat de chemin PC_ω. Soit Ω^M l'ensemble des chemins d'exécution fautée pour un programme P et un modèle de faute M. Un chemin d'exécution fautée $\omega \in \Omega^M$ peut correspondre à une exécution nominale (sans-faute) ou une attaque (exécution avec au moins une faute). On note $\mathcal{E}^M$ l'ensemble des prédicats de chemins fautés produits par un moteur d'exécution symbolique fauté ε, séparant l'exploration des chemins entre le cas fauté et le cas non fauté, à chaque fois qu'un point d'injection est rencontré.

On peut alors s'intéresser à la *correction* et la *complétude* d'un prédicat de chemin fauté PC_ω^M, qui fournit des entrées et des fautes menant à une exécution suivant le chemin d'exécution fauté ω (définitions 3.3 et 3.4). Les définitions 3.5 et 3.6 s'intéressent respectivement à la *correction* et la *complétude* de l'énumération des prédicats de chemins fautés $\mathcal{E}^M$ par un moteur d'exécution symbolique fauté ε.

Définition 3.3. *Un prédicat de chemin fauté PC_ω^M est correct si tout modèle satisfaisant PC_ω^M fournit des entrées et des fautes pour une exécution (fautée) suivant le chemin ω.*

Définition 3.4. *Un prédicat de chemin fauté PC_ω^M est complet si toute paire (entrées, fautes) suivant le chemin ω est un modèle satisfaisant PC_ω^M.*

Définition 3.5. *L'énumération des prédicats de chemins fautés $\mathcal{E}^M$ est correcte ssi, $\forall PC_\omega^M \in \mathcal{E}^M$, PC_ω^M est correct.*

Définition 3.6. *L'énumération des prédicats de chemins fautés $\mathcal{E}^M$ est complète ssi :*
- *$\forall PC_\omega^M \in \mathcal{E}^M$, PC_ω^M est complet, **et**,*
- *pour tout chemin d'exécution fauté $\omega \in \Omega^M$, $\exists PC_\omega^M \in \mathcal{E}^M$ tel que PC_ω^M fournit des entrées et des fautes pour une exécution (fautée) suivant le chemin fauté ω.*

La complétude de l'énumération des prédicats de chemins fautés (définition 3.6) correspond donc à avoir un ensemble de prédicats complets et que tous les chemins d'exécutions fautées soient représentés par au moins un prédicat de chemin dans $\mathcal{E}^M$. Avec l'émulation des fautes comme décrit dans la figure 3.2, les fautes sont des entrées (booléen symbolique). Chaque cas de test (ktest) fourni par KLEE correspond ainsi à un prédicat de chemin fauté avec un modèle représentant qui fournit ainsi les entrées et les fautes d'une exécution.

3.2 Fonctionnement et architecture de Lazart

La figure 3.3 présente le processus d'une analyse avec Lazart, qui s'organise en 5 étapes principales :
- *Étape 1 :* l'utilisateur fournit le code source du programme et le modèle d'attaquant, décrit dans un *script d'analyse* en Python et par l'instrumentation du programme.
- *Étape 2 :* le programme est compilé pour produire le bytecode LLVM-IR.
- *Étape 3 :* le bytecode est muté pour y introduire le comportement des fautes.
- *Étape 4 :* le bytecode LLVM-IR muté est donné au moteur d'exécution concolique KLEE qui génère un ensemble de cas de tests (traces).
- *Étape 5 :* les différents traitements sont effectués à partir des traces générées par KLEE.

Le fonctionnement indépendant des différentes étapes permet de reprendre une analyse en réutilisant les données intermédiaires des étapes précédentes, ceci afin de ne pas ré-exécuter certaines opérations coûteuses (comme l'exécution concolique). Cette architecture modulaire permet aussi de simplifier l'intégration avec d'autres chaînes logicielles qui peuvent ainsi modifier une étape ou se placer entre les étapes du processus d'analyse.

Lazart est organisé en plusieurs modules :
- *Lazart Core* : une API Python permettant la description et la manipulation des analyses, des traitements, des traces et des résultats.
- *Wolverine* : l'outil permettant de muter un bytecode LLVM afin d'y introduire les potentielles fautes.
- *Les outils externes* : dont notamment la chaîne de compilation de LLVM et l'outil d'exécution concolique KLEE.

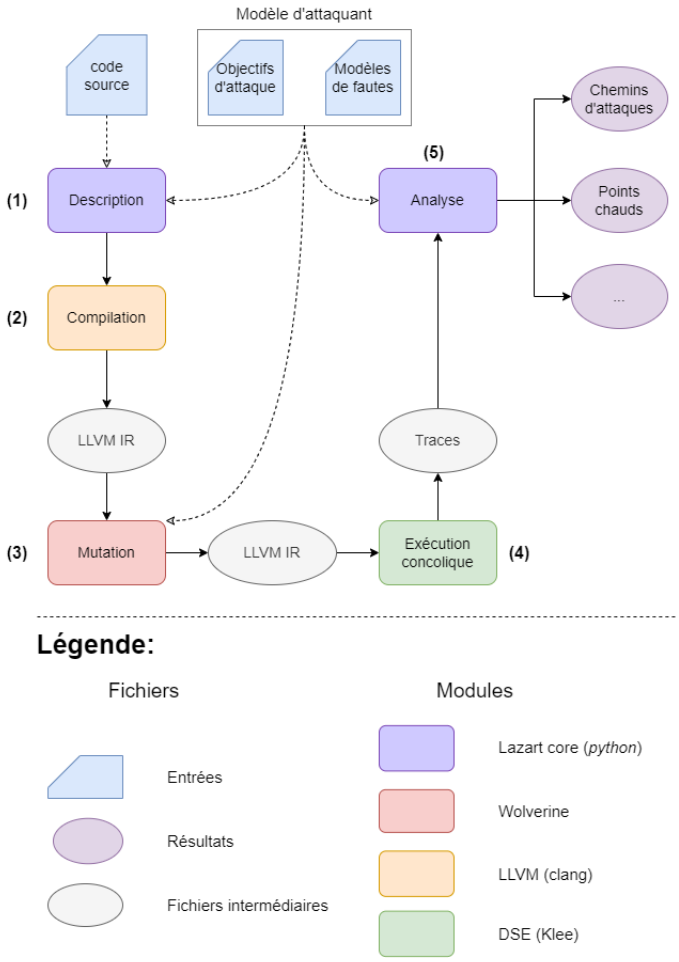

FIGURE 3.3 – Processus d'analyse avec Lazart

Lazart Core permet à l'utilisateur de définir ses propres traitements à l'aide d'une API en Python. Six traitements, appelés ici « analyses », sont proposés nativement dans Lazart :

— *Recherche d'attaques* : recherche des chemins d'attaques.
— *Analyse d'équivalence* : élimination des attaques « équivalentes » (voir section 3.4.3).
— *Analyse de redondance* : élimination des attaques « redondantes » (voir section 3.4.4).
— *Analyse de points chauds* : identification des points d'injection les plus sensibles (voir section 3.4.5).
— *Analyses de placement de contre-mesures* : recherche des placements optimaux pour des contre-mesures « locales » (décrites dans le chapitre 5).
— *Analyse d'optimisation de contre-mesures* : recherche des protections superflues ou redondantes dans une programme protégé (décrite dans le chapitre 6).

Lazart supporte trois modèles de faute :

— *L'inversion de test* (Test Inversion (TI)) : qui correspond à la sélection de la mauvaise branche lors d'un branchement conditionnel (instruction `br`).
— *La mutation de données* (Data Load (DL)) : la donnée lue lors de l'exécution d'une instruction `load` est modifiée, la valeur injectée étant laissée au choix de l'utilisateur (valeur fixe ou valeur symbolique, éventuellement contrainte).

— *Le saut arbitraire intra-procédural* (JuMP (JMP)) : la faute provoque un saut vers un point du programme (défini par l'utilisateur).

Ces trois modèles peuvent être combinés en multi-fautes, et permettent déjà d'exprimer un grand nombre de scénarios d'attaque. La suite de ce chapitre se concentre sur la description et l'utilisation des modèles de faute de Lazart pour l'utilisateur, le chapitre 4 reviendra plus en détail sur l'implémentation de ces modèles.

3.3 Description d'une analyse

Cette section présente la description par l'utilisateur d'une analyse de recherche de chemin d'attaque avec Lazart. La section 3.3.1 présente la fonction *memcmps* qui servira d'exemple. La section 3.3.2 décrit le script d'analyse et les paramètres qui lui sont associés. La section 3.3.3 s'intéresse à la description de l'objectif d'attaque et la section 3.3.4 à la description des modèles de faute.

3.3.1 La fonction `memcmps`

Les différentes étapes d'analyse seront illustrées sur la fonction `memcmps` présentée dans le listing 3.9. `memcmps` effectue une comparaison de deux tableaux d'octets en appelant plusieurs fois la fonction `memcmp` de la bibliothèque standard C de manière à se protéger contre des fautes. Si une faute sur la donnée ou une inversion de test fausse le résultat d'un appel à `memcmp`, la variable `result` ne contiendra ni `FALSE` ni `TRUE` et l'attaque sera donc détectée. On considérera l'objectif d'attaque qui consiste à obtenir le retour `TRUE` malgré des entrées différentes, et la combinaison des modèles de faute d'inversion de test et de mutation de données arbitraire (non contrainte).

```
1  // memcmps.h
2  typedef BOOL uint16_t;
3  #define TRUE    0x1234u
4  #define FALSE   0x5678u
5  #define MASK    0xABCDu
6
7  // memcmps.c
8  #include "memcmps.h"
9
10 BOOL memcmps(uint8_t* a, uint8_t* b, size_t len)
11 {
12   BOOL result = FALSE;
13
14   if (!memcmp(a, b, len)) {
15     result ^= MASK;           // result = FALSE ^ MASK
16     if (!memcmp(a, b, len)) {
17       result ^= FALSE ^ TRUE; // result = MASK ^ TRUE
18       if (!memcmp(a, b, len)) {
19         result ^= MASK;       // result = TRUE
20       }
21     }
22   }
23
24   return result;
25 }
```

Listing 3.9 – Fonction `memcmps`

3.3.2 Script d'analyse et paramètres

L'utilisateur décrit les paramètres et les traitements d'une analyse dans un *script d'analyse* et en instrumentant le programme étudié. Le listing 3.10 correspond au script d'analyse pour l'exemple `memcmps`. Ce script contient tout d'abord l'importation de Lazart (ligne 2). Les paramètres d'une analyse sont définis dans un objet de type *Analysis*. Le constructeur (ligne 4) prend deux paramètres obligatoires :

— les fichiers sources du programme : ligne 6, `memcmps.c` contenant la fonction à analyser ainsi que `main.c`, la fonction principale de l'analyse (voir section 3.3.3).

— le modèle d'attaquant, pour le moment représenté par la variable `attack_model` (voir section 3.3.4).

```
1  #!/usr/bin/python3
2  from lazart.lazart import *
3
4  a = Analysis(["memcmps.c", "main.c"], # Input files
5      attack_model, # Attack model
6      flags=AnalysisFlag.AttacksAnalysis, # Analysis type
7      compiler_args="-Wall",
8      max_order=4,
9      path="my_analysis")
10
11 execute(a)
```

Listing 3.10 – Script d'analyse d'attaque pour l'exemple `memcmps`

Un certain nombre de paramètres optionnels peuvent être spécifiés, la liste complète étant disponible dans l'annexe A.1. Dans l'exemple précédant, `flags` (ligne 6) indique les traitements qui seront effectués (ici une recherche d'attaques, `AttacksAnalysis`, la valeur par défaut). `compiler_args` (ligne 7) indique les arguments supplémentaires pour l'étape de compilation et `max_order` (ligne 8) correspond à la limite de fautes de l'analyse (par défaut 2). `path` (ligne 9) est le chemin du dossier d'analyse dans lequel les différents fichiers temporaires et les résultats seront stockés.

La fonction `execute` lance les différentes étapes de l'analyse (conformément à la figure 3.3) ainsi que la génération et l'affichage des résultats, en exécutant la séquence d'opérations [4] indiquée dans le listing 3.11.

```
1 compile_results(a) # Compilation.
2 run_results(a) # Mutation and DSE.
3 traces_results(a) # Gathering traces information from KLEE's ktests.
4 attacks_results(a) # Computing attacks analysis.
5 print_results(a) # Prints results of analysis.
6 generate_reports(a) # Generate analysis reports files.
```

Listing 3.11 – Équivalent de la fonction `execute` pour une analyse d'attaque

La suite de cette section explique comment l'utilisateur décrit l'objectif d'attaque et le modèle d'attaquant.

3.3.3 Description de l'objectif d'attaque

L'objectif d'attaque est spécifié dans le programme à analyser. Le listing 3.12 présente le fichier principal de l'analyse `main.c`, instrumenté avec la spécification de l'objectif d'attaque. C'est cette fonction `main` qui sera le point d'entrée de l'exécution symbolique après l'étape de mutation.

4. Dans le cadre d'une recherche de chemins d'attaques (`flag=AttacksAnalysis`) comme pour l'exemple 3.10.

Les deux tableaux d'entrée `a1` et `a2` sont rendus symboliques à l'aide de la fonction `_LZ__SYM` (ligne 11 et 13) qui prend en paramètre le pointeur vers la zone mémoire à rendre symbolique et la taille de cette zone mémoire. L'objectif d'attaque est vérifié avec la macro `_LZ__ORACLE` qui limite la recherche aux d'exécutions qui valident le prédicat passé en paramètre. Elle permet ainsi de décrire la contrainte d'inégalité des entrées symboliques (ligne 19) et de vérifier que le retour de la fonction est `TRUE` (ligne 23). Les macros `_LZ_SYM` et `_LZ_ORACLE` sont implémentées à partir des directives de KLEE, respectivement `klee_make_symbolic` et `klee_assume` (section 3.1.4).

```
1  // main.c
2  #include "lazart.h"
3  #include "memcmps.h"
4
5  #define SIZE 4
6
7  int main()
8  {
9      // Inputs
10     uint8_t a1[SIZE];
11     _LZ__SYM(a1, SIZE); // Symbolic array
12     uint8_t a2[SIZE];
13     _LZ__SYM(a2, SIZE); // Symbolic array
14
15     bool equals = true;
16     for(size_t i = 0; i < SIZE; ++i)
17         if(a1[i] != a2[i])
18             equals = false;
19     _LZ__ORACLE(!equal); // Consider only different inputs
20
21     BOOL res = memcmps(a1, a2, SIZE); // Call studied function
22
23     _LZ__ORACLE(res == TRUE); // Attack objective
24 }
```

Listing 3.12 – Fonction principale de l'analyse

3.3.4 Description des modèles de faute

Les modèles de faute peuvent être spécifiés de deux manières :

— dans le script d'analyse en Python (via l'argument `attack_model`), permettant une granularité de l'ordre des fonctions ou des blocs de base.

— en instrumentant le code, ce qui permet un contrôle plus précis.

Modèle	Description		Granularité	
	Python	[redacted]	Python	Instr.
Inversion de test (TI)	Oui	Oui	Fonctions et blocs de base	Branchements
Mutation de donnée (DL - mutation de constantes	Oui Non	Oui Oui	Fonctions, blocs de base et variables	Instructions
Saut arbitraire JMP)	Non	Oui	N/ A	Instructions

TABLE 3.2 – Définition et granularité des modèles de faute

La table 3.2 indique la granularité de chaque modèle de faute en fonction de la méthode de définition utilisée. La colonne *Description* détermine si le modèle de faute peut être défini avec chaque méthode et la colonne *Granularité* précise le niveau de granularité pour chaque méthode. La description en Python associe des modèles de faute à des portions de

programme (fonction ou bloc de base), mais n'est disponible que pour l'inversion de test et la mutation de données.

La suite de cette section explique comment ces modèles de faute sont décrits par l'utilisateur.

3.3.4.1 Description du modèle d'attaquant en Python

La description du modèle d'attaquant en Python est faite en associant des modèles de faute à des fonctions ou des blocs de base du programme. Le listing 3.13 présente un exemple de définition d'un modèle d'attaquant `am` composé de trois modèles de faute. L'annexe A.2 décrit l'ensemble des paramètres disponibles pour chaque modèle de faute.

```
model_ti = { "type": "test-inversion" }

model_breset = {
    "type": "data-load",
    "all": 0 # All load are faulted to 0
    "exclude": ["i", "tmp"] # Except variables 'i' and 'tmp'.
}

model_gl = {
    "type": "data-load",
    "vars": {
        "my_global": "__sym__" # Arbitrary (unconstrained) value.
    }
}

am = {
    "functions": { # Associate models to functions.
        "__all__": [model_gl], # All functions
        "foo": [model_ti, model_breset]
    }
}
```

Listing 3.13 – Exemple de modèle d'attaquant décrit en Python

Dans l'exemple ci-dessus un modèle d'inversion de test (`model_ti`) est appliqué sur la fonction `foo` avec le modèle de mutation de mise-à-zero (`model_breset`). Le modèle `model_gl` injecte une valeur arbitraire pour toute lecture de la variable `my_global` (avec le mot clef `__sym__`). Le champ `functions` permet l'application des modèles de faute sur des fonctions du programme (le champ `basic-blocks` est l'équivalent pour les blocs de base).

Le listing 3.14 présente le modèle d'attaquant pour l'exemple `memcmps` afin d'obtenir une combinaison de l'inversion de test et de la mutation de données. Une mutation en donnée arbitraire est appliquée sur les variables `result` et `len` et l'inversion de test est appliquée sur l'ensemble de la fonction. La ligne 14 est une version plus compacte de la définition de ce même modèle utilisant des fonctions utilitaires : `function_list` associe un ensemble de modèles de faute à une liste de fonctions, et `ti_model` et `data_model` permettent de créer un modèle avec le champ `type` correspondant.

```
attack_model = {
    "functions" = {
        "memcmps" = [ {
                "type": "test-inversion"
            },
            {
                "type": "data-load",
                "vars": { "result": "__sym__", "len":  "__sym__" }
            }
        ]
```

```
    }
}

attack_model = functions_list(["memcmps"], [ti_model(), data_model({ "vars": { "result": "
    __sym__", "len":  "__sym__" } })])
```

Listing 3.14 – Modèle d'attaquant pour l'exemple `memcmps`

3.3.4.2 Spécification du modèle de mutation de données

Le modèle de mutation de données de Lazart vise la généricité en laissant l'utilisateur spécifier la valeur qui sera injectée. Cette valeur peut être une constante, une valeur contrainte par un prédicat (encodé par une fonction définie par l'utilisateur dans le programme source) ou entièrement symbolique (non contrainte) pour le type donné. Le listing 3.15 contient un exemple de description de la mutation de données pour chacun de ces cas.

```
{
    "type": "data-load",
    "vars": {
        "x": 0,               # Fixed value
        "y": "my_predicate",  # Constrained symbolic
        "z": "__sym__",       # Not-constrained symbolic
    }
}
```

Listing 3.15 – Différentes spécifications de valeurs fautées en Python

Le listing 3.16 présente un exemple de deux prédicats `flip_pred` et `bf_pred` encodant respectivement le modèle d'inversion de tous les bits (*flip*) et le modèle d'inversion d'un seul bit (*bit-flip*), implémenté en langage C. Un prédicat prend en entrée la valeur originale (potentiellement symbolique) et la valeur injectée (toujours symbolique).

```
bool bf_pred(int value, int faulted)
{
    return hamming_distance(value, faulted) == 1; // on binary encoding.
}

bool flip_pred(int value, int faulted)
{
    return faulted == ~value;
}
```

Listing 3.16 – Prédicat pour le modèle *bit-flip*

La table 3.3 présente les types de mutation de données de Lazart permettant de représenter certains modèles de faute de la littérature. La première colonne indique le modèle de faute considéré et les colonnes suivantes indiquent si la méthode correspondante peut être utilisée (*oui*) ou *non*, *possible* indiquant que la méthode peut représenter le modèle mais n'est pas la plus adaptée.

L'approche par *prédicat* est la plus générique, pouvant s'appliquer dans tous les cas, la *valeur fixe* et la valeur non-contrainte (*symbolique*) étant réservées à des cas spécifiques. Le chapitre suivant (section 4.3.2) s'intéresse aux considérations liées aux performances de l'analyse dans le choix de la fonction de mutation associé à chaque type de valeur fautée.

3.3.4.3 Mutation par instrumentation

L'instrumentation du programme est complémentaire à la description du modèle d'attaquant en Python. L'instrumentation peut être utilisée afin :

Modèles de la littérature	Valeur fixe	Symbolique	Prédicat
bit-set bit-reset	possible	non	oui
bit-flip	non	non	oui
byte-set byte-reset	oui	non	oui
byte-flip	non	non	oui
set reset	oui	non	possible
inversion	non	non	oui
valeur aléatoire	non	oui	possible
valeur arbitraire	non	oui	oui

TABLE 3.3 – Modèle de faute sur les données

— d'obtenir un contrôle plus fin sur l'application des modèles de faute.
— de définir un point d'injection spécifique dans le programme.
— de spécifier un modèle de faute non disponible via l'API Python (saut arbitraire).

Le listing 3.17 présente une fonction `foo` contenant des macros de Lazart permettant le contrôle de l'activation de modèles. Les fonctions `_LZ__disable_model`, et `_LZ__enable_models` permettent respectivement de désactiver un modèle nommé (à l'aide du champ `name` d'un modèle de faute) et de réactiver tous les modèles. Ces fonctions doivent être placées en prenant en compte l'ordre dans lequel le compilateur va placer les blocs de base dans la représentation LLVM-IR [5]. L'état d'activation des modèles n'est pas préservé entre les fonctions. La fonction `_LZ__disable_bb` permet ici de désactiver tous les modèles dans un bloc de base.

```
1  int foo(int i, float f)
2  {
3      _LZ__disable_model("fm:ti"); // Disable a named fault model.
4      int ret = 0;
5      if(a < 0)
6      {
7          _LZ__disable_bb(); // Disable all models in the current basic block.
8          ret = a * f;
9      }
10     else
11     {
12         ret = bar(a * f);
13     }
14     _LZ__enable_models(); // Enable all disabled models.
15
16     return ret;
17 }
```

Listing 3.17 – Activation et désactivation de modèles

Le listing 3.18 présente une fonction `bar` contenant des points d'injection définis manuellement. La fonction `_LZ__DATA_i32` (ligne 6) permet de spécifier un point d'injection de mutation de données sur la constante `10` correspondant à la limite de boucle, le second paramètre est l'identifiant du point d'injection. Aux lignes 4 et 5, deux points d'injection de sauts d'instruction (`lend` et `end`) sont définis avec la macro `_LZ__JUMP`. La faute provoque un saut respectivement aux labels `loop_end` (ligne 9) et `end` (ligne 12).

5. Cela dépend du compilateur et de sa version mais l'utilisateur peut inspecter le code LLVM-IR généré en cas d'ambiguïté.

```
int bar(int a, int b)
{
    int total = 0;
    _LZ__JUMP(loop_end, "lend"); // (label, id)
    _LZ__JUMP(end, "end"); // (label, id)
    for(int i = 0; a < _LZ__DATA_i32(10, "loop_bound"); ++i) // (original, id)
    {
        total += a + b;
        loop_end:
    }

    end:
    return foo(total);
}
```

Listing 3.18 – Définition manuelle de points d'injection

3.4 Traces et traitements

Les différents traitements de Lazart prennent en entrée un ensemble de traces d'exécution qui sont obtenues à partir des cas de tests de KLEE. Cette section présente la modélisation des traces au sein de Lazart (section 3.4.1) et décrit les différentes analyses de robustesse de Lazart : *l'analyse d'attaque* (section 3.4.2), *l'analyse d'équivalence* (section 3.4.3), *l'analyse de redondance* (section 3.4.4) et *l'analyse de points chauds* (section 3.4.5).

3.4.1 Représentation des traces dans Lazart

Comme cela a été vu dans la section 3.1.5, l'exécution concolique sur le mutant embarquant les fautes revient à une exploration de tous les chemins fautés [6]. Chaque cas de test généré par KLEE peut être vu comme une trace d'exécution représentant un ensemble d'exécutions qui suivent le même chemin et comprennent les mêmes déclenchements de fautes. Les valeurs concrètes (pour les entrées et les fautes) d'un cas de test permettent d'obtenir un représentant de cet ensemble d'exécutions.

Ces traces d'exécution sont représentées dans Lazart par un type de *terminaison*, et une liste ordonnée de *transitions* pouvant être du type :

— *bloc de base* : entrée dans un bloc de base.
— *faute* : un point d'injection a été déclenché.

Définition 3.7. *Soit P un programme et m un modèle de faute, on note $T(P, m)$ l'ensemble des traces d'exécution de P, ou plus simplement $T(P)$.*

Les transitions correspondant à l'entrée dans un bloc de base permettent de tracer le flot d'exécution. Les transitions fautées sont paramétrées par le point d'injection déclenché (et donc le modèle de faute correspondant) et les paramètres de la faute. Le type de terminaison d'une trace t, notée $Term(t)$, correspond aux terminaisons des cas de tests de KLEE [7] (section 3.1.4) étendues avec la terminaison en contre-mesure (`detected`) et des cas d'erreurs supplémentaires spécifiques à Lazart (`lzerrs`).

Définition 3.8. *Soit $t \in T(P)$, on note :*
— *$Tr(t)$ la liste ordonnée des transitions dans t.*
— *$N(t)$ la liste ordonnée des transitions non fautées dans t.*

6. Dans le cas idéal, c'est-à-dire dans le cas où l'exécution concolique est complète. Les chemins fautés dépassant la limite de fautes fixée par l'utilisateur ne sont pas explorés.
7. Pour rappel : `ptr`, `div`, `free`, `abord`, `assert`, `model`, `satisfy`, `timeout` etc.

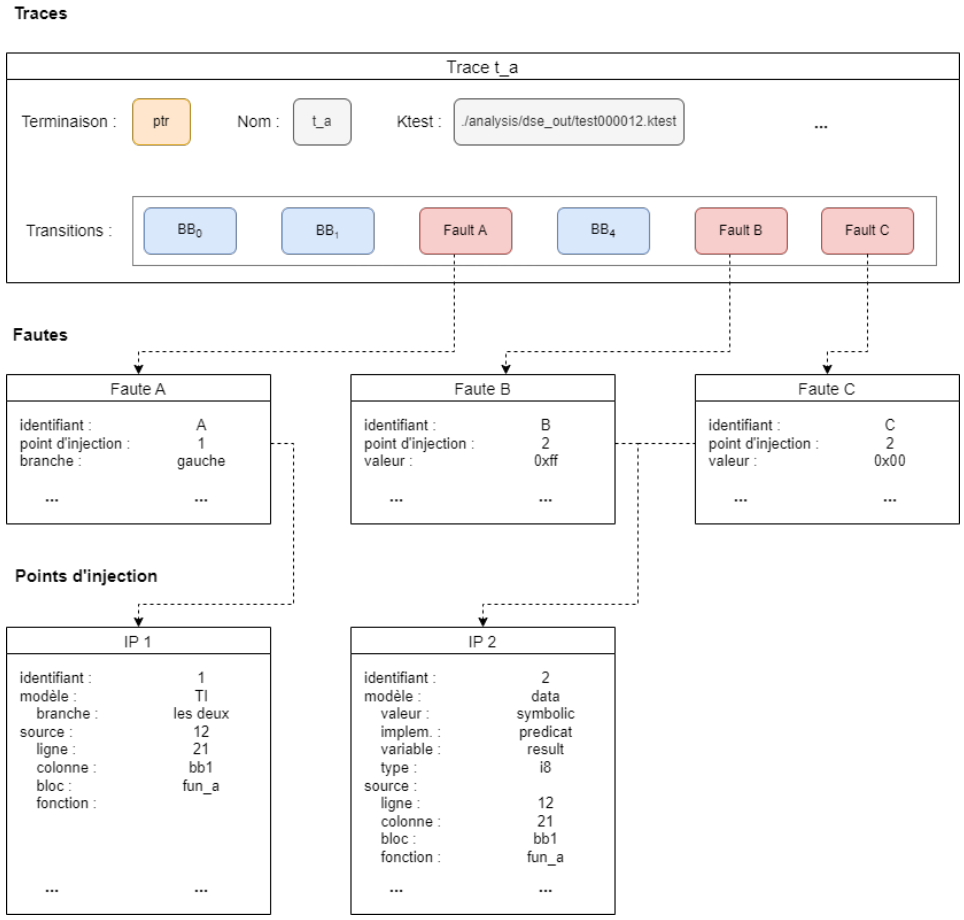

FIGURE 3.4 – Représentation d'une trace d'attaque dans Lazart

— $F(t)$ *la liste ordonnée des transitions fautées dans* t.

La figure 3.4 donne la représentation d'une trace d'exécution `t_a` contenant la faute A en inversion de test et les fautes B et C en mutation de données et deux points d'injection.

3.4.2 Analyse d'attaque

L'analyse d'attaque correspond à la recherche de chemins d'attaques validant un objectif d'attaque ϕ.

Définition 3.9. *Une attaque est une trace d'exécution comportant au moins une faute :* $F(t) \neq \emptyset, t \in T(P)$.

Définition 3.10. *Une attaque réussie, pour un objectif d'attaque* ϕ*, est une attaque validant l'objectif d'attaque :* $F(t) \neq \emptyset \wedge t \vDash \phi,\ t \in T(P)$.

La table 3.4 donne le nombre de chemins d'attaque trouvés pour notre exemple avec une limite de fautes fixée à 4, et avec le modèle de faute combiné d'inversion de test et de mutation de données sur `result` et `i`. La taille des tableaux d'entrées est fixée à 4 et leurs valeurs sont symboliques (et contraintes d'être différentes sur au moins un octet

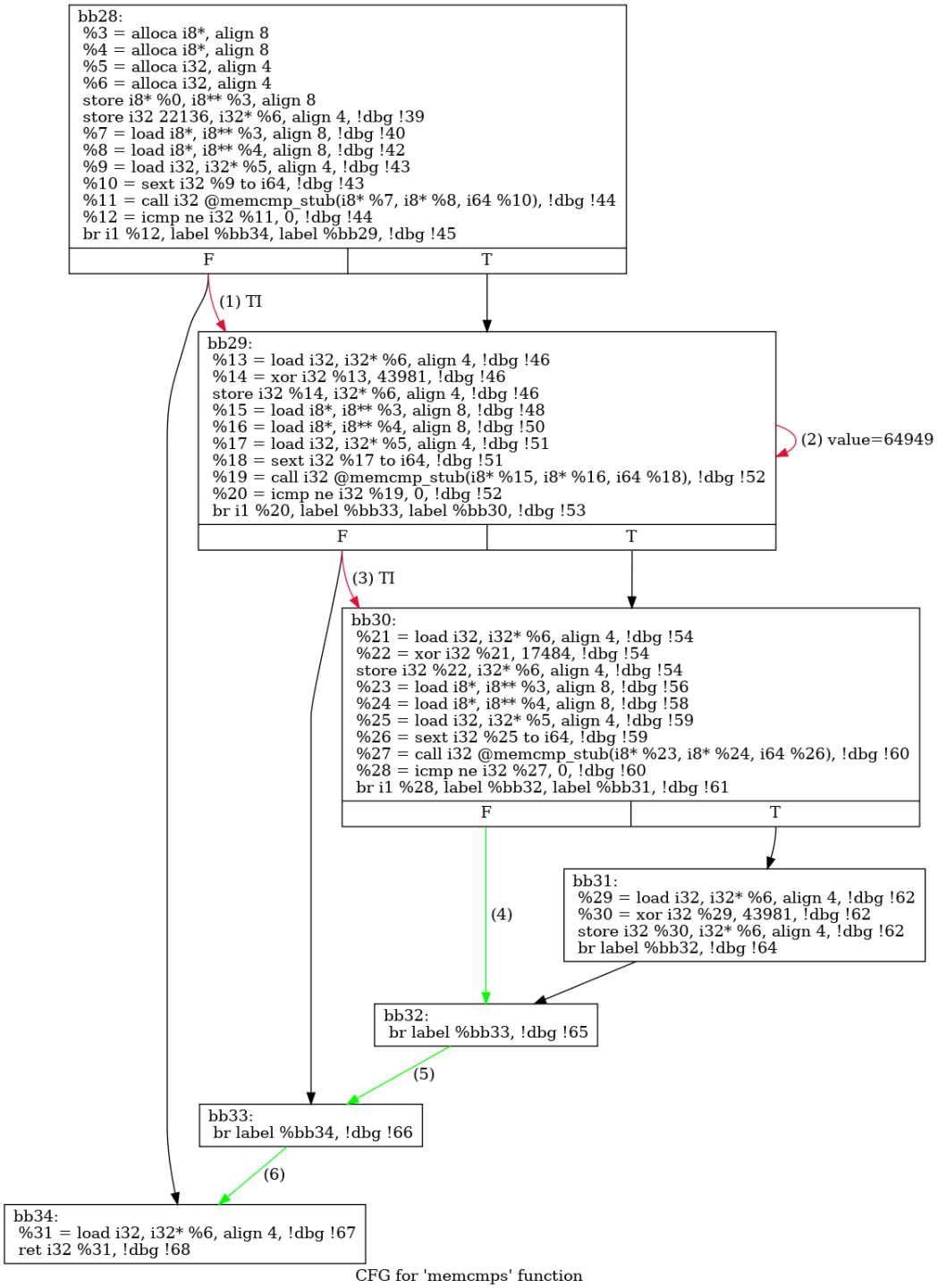

FIGURE 3.5 – Graphe de faute de l'attaque en trois fautes sur `memcmps`

TABLE 3.4 – Chemins d'attaque trouvés avec `memcmps`

Fautes	1F	2F	3F	4F	5F	Total
Attaques réussies	0	4	8	20	28	60

Le listing 3.19 présente une trace en trois fautes composée de deux inversions de test et une mutation de donnée. On y trouve en premier lieu le nombre de fautes dans la trace (`3`), son nom (`t15`) puis la liste ordonnée des transitions [8]. Finalement le mot-clef `Satisfy` correspond au type de terminaison (ici l'objectif d'attaque a été validé).

La figure 3.5 correspond au graphe d'attaque généré par Lazart pour cette attaque. Le chemin suivi dans le graphe de flot de contrôle du programme étudié est indiqué en vert et les fautes sont indiquées en rouge. Cette attaque consiste à inverser le test du premier appel à `memcmp` (ligne 14 [9]), puis injecter la valeur $FALSE \oplus MASK$ ($0x5678 \oplus 0xABCD = 64949$) dans la variable *result* (ligne 15) et enfin inverser le test ligne 16. La ligne 17 est ainsi exécutée, on a $result = result \oplus FALSE \oplus TRUE \implies result = FALSE \oplus FALSE \oplus TRUE = TRUE$. Le dernier test n'est pas fauté et la ligne 19 n'est pas exécutée, la valeur $TRUE$ est donc retournée par la fonction.

```
1 (<3> t15: [BB(bb28), FAULT(0|TI|: bb28 => bb29 [l:27, bb:bb28]), BB(bb29), FAULT(1|DL|: ~
      => 64949 [l:28, bb:bb29]), FAULT(2|TI|: bb29 => bb30 [l:29, bb:bb29]), BB(bb30), BB(
      bb32), BB(bb33), BB(bb34)]: Satisfy)
```

Listing 3.19 – Attaque en trois fautes sur `memcmps`

Comme toutes les analyses de Lazart, l'analyse d'attaque prend l'argument `satisfies_fct` correspondant à un prédicat déterminant si une trace valide l'objectif d'attaque ou non afin de laisser un contrôle fin à l'utilisateur. Cela permet d'exprimer des objectifs d'attaque qui ne peuvent pas être décrits uniquement par l'instrumentation (par exemple des erreurs à l'exécution). Le listing 3.20 présente la définition d'un prédicat considérant comme une attaque les traces finissant en erreur arithmétique ou d'adresse, en plus des exécutions validant l'objectif d'attaque [10]. La liste complète des paramètres des différentes analyses est donnée dans l'annexe A.2.

```
1 Term = TerminationType # alias
2 attacks_analysis(satisfies_fct=lambda trace: trace.termination_is(Term.Satisfy, Term.Ptr,
      Term.Div, Term.Free))
```

Listing 3.20 – Filtrage des traces d'attaque en fonction de la terminaison

3.4.3 Équivalence des attaques et entrées symboliques

Lazart propose les notions d'*équivalence* et de *redondance* (abordée en section 3.4.4) qui forment une relation d'ordre partiel entre les attaques permettant de présenter en priorité certaines attaques à l'utilisateur. Ces notions visent à aider à la protection et l'évaluation du code, en permettant de cibler les attaques jugées plus importantes.

Deux définitions d'équivalence sont proposées par Lazart nativement. Elles visent chacune à regrouper entre-elles des traces d'attaques selon des objectifs différents :

8. Ici chaque faute indique dans l'ordre : l'identifiant du point d'injection concerné, le modèle de faute (ici `TI` ou `DL`), les informations spécifiques au modèle et finalement les informations sur la position du point d'injection dans le code source.
9. Voir code de `memcmps` dans le listing 3.9.
10. Par défaut, seules les traces avec pour type de terminaison *satisfy* sont prises en compte.

1) *Équivalence* (section 3.4.3.1) : vise à pallier à une problématique relative à une division de certains chemins d'attaques, en raison du code implémentant la vérification de l'objectif d'attaque dans le cadre d'entrées symboliques.
2) *Faute-équivalence* (section 3.4.3.2) : vise à regrouper des attaques qui ne diffèrent que par des variations légères et que l'on souhaite ne pas distinguer pour l'utilisateur.

3.4.3.1 Distinguabilité et équivalence

Définition 3.11 (Équivalence)**.** *Une attaque a est dite* ***équivalente*** *(noté $\equiv$) à une attaque a' si leur séquence de transitions sont égales (égalité syntaxique des séquences) :*
$a \equiv a' =_{def} Tr(a) = Tr(a')$

La notion d'équivalence des attaques (définition 3.11) vise à regrouper les attaques qui suivent le même chemin, contiennent les mêmes fautes, mais ont des contraintes sur les entrées différentes. On définira la *distinguabilité* d'un prédicat de chemin fauté au sein d'un ensemble ω^M (définition 3.12) et la *distinguabilité* de l'énumération des prédicats de chemins fautés (définition 3.13).

Définition 3.12. *Un prédicat de chemin fauté PC^M_ω est dit distinct dans $\mathcal{E}^M$, si pour tout modèle m satisfaisant PC^M_ω, il n'existe pas $PC' \in \mathcal{E}^M$ tel que $m \vDash PC'$.*

Définition 3.13. *L'énumération des prédicats de chemins fautés (d'une exécution symbolique fautée ε) est distincte si tous les prédicats de chemins sont distincts.*

Il est possible que l'exécution concolique fautée ne soit pas *distincte*, par exemple à cause du code encodant la vérification de l'objectif d'attaque qui divise certains chemins en plusieurs chemins non distincts (équivalents). Le programme présenté dans le listing 3.21 compte le nombre d'octets nuls dans un tableau de taille donnée. Une exécution symbolique sur la fonction `null_bytes` produit quatre chemins pour `size` fixé à 2 :
— Le chemin avec : $tab[0] = 0 \wedge tab[1] = 0$.
— Le chemin avec : $tab[0] \neq 0 \wedge tab[1] = 0$.
— Le chemin avec : $tab[0] = 0 \wedge tab[1] \neq 0$.
— Le chemin avec : $tab[0] \neq 0 \wedge tab[1] \neq 0$.
Ces quatre chemins sont distincts (les séquences de transitions ne sont pas strictement équivalentes), chacun exécutant ou non l'incrémentation (ligne 5) à chacun des deux tours de boucle.

```
1 int null_bytes(uint8_t* tab, size_t size) {
2     int total = 0;
3     for(int i = 0; i < size; i++)
4         if(tab[i] == 0)
5             total++;
6     return total;
7 }
```

Listing 3.21 – Fonction `null_bytes`

Un objectif d'attaque consiste à retourner la mauvaise valeur en injectant des fautes [11]. Il est nécessaire d'encoder cette propriété dans le programme et pour cela, une méthode classique consiste à appeler la fonction étudiée sans injecter de faute et comparer les résultats, comme présenté dans le listing 3.22. Le tableau `tab` est rendu symbolique (ligne 3) et la valeur retournée (`ret`) est comparée à une version non fautée de la fonction à étudier (ici `null_bytes_safe` ligne 8).

11. Silent Data Corruption (SDC).

```
1 int main() {
2     uint8_t tab[SIZE];
3     _LZ__SYM(&tab, SIZE);
4
5     int ret = null_bytes(tab, SIZE);
6
7     // Verify attack objectives
8     _LZ__ORACLE(ret != null_bytes_safe(tab, SIZE)); // Attacks division
9 }
```

Listing 3.22 – Point d'entrée de l'analyse pour `null_bytes`

Pour chaque chemin atteignant le point de vérification de l'objectif d'attaque (ligne 8), la fonction `null_bytes_safe` est exécutée par le moteur d'exécution concolique. Dans le cas d'une exécution nominale, l'appel à `null_bytes_safe` produit exactement les mêmes contraintes pour chaque chemin que l'appel à `null_bytes`, seuls quatre chemins sont donc générés, aucun ne validant l'objectif d'attaque. Dans le cadre d'une exécution fautée, l'appel à `null_bytes_safe` peut diviser certains chemins d'attaque, en ajoutant des contraintes différentes de celles de `null_bytes` en raison des fautes. Si un chemin est divisé en plusieurs chemin qui valident l'objectif d'attaque, plusieurs ktests sont produits par KLEE alors qu'ils suivent le même chemin (dans la fonction étudiée `null_bytes`, mais dont le chemin diffère dans `null_bytes_safe`) et contiennent les mêmes points d'injection de fautes.

La table 3.5 compare les résultats obtenus pour les trois variantes de l'encodage de l'objectif d'attaque :

- *fix* : le tableau d'entrée est fixé à [1, 1].
- *sym-post* : le tableau d'entrée est symbolique et l'objectif d'attaque est vérifié à la fin.
- *sym-pre* : le tableau d'entrée est symbolique et l'objectif d'attaque est vérifié à la fin, mais la valeur attendue est pré-calculée avant l'appel de la fonction à analyser.

TABLE 3.5 – Résultats d'analyse d'équivalence sur les attaques réussies pour `null_bytes`

	Chemins Expl.	Traces	Attaques	1 faute	2 fautes	Total
fix	14	11	attaques	7	4	11
			classes eq.	7	4	11
sym-pre	14	11	attaques	7	4	11
			classes eq.	7	4	11
sym-post	19	16	attaques	9	7	16
			classes eq.	8	4	12

La colonne "Chemins Expl." indique le nombre de chemins explorés par KLEE, et "Traces" le nombre de traces générées par Lazart. Les colonnes suivantes indiquent le nombre d'attaques réussies trouvées (ligne "attaques") et le nombre de classes d'équivalence obtenues (ligne "classes eq."). On peut constater que l'analyse est *distincte* en cas de pre-vérification dans cet exemple contrairement à la post vérification. Les attaques obtenues après équivalence (lignes "classes eq.") sont bien égales entre les versions *sym-pre* et *sym-post*, à l'exception d'une classe d'équivalence supplémentaire qui n'est pas trouvée dans le cas *sym-pre*. Le pré-calcul est plus efficace puisque les contraintes sur les entrées sont exprimées avant l'appel de la fonction à étudier. Par conséquent, certains chemins peuvent être coupés directement pendant l'exécution symbolique fautée de `null_bytes`. La version *sym-pre* est aussi plus efficace, moins de chemins étant explorés. La version *fix* est distincte mais est incomplète par rapport à l'utilisation d'entrées symboliques. L'utilisateur devrait

dans le cas idéal préférer calculer au plus tôt les variables nécessaires pour vérifier l'objectif d'attaque (valeur attendues, contraintes des entrées...), même si cet exemple illustre le cas où un chemin supplémentaire est obtenu dans le cas de pre-vérification.

En résumé, la notion d'équivalence (définition 3.11) permet de regrouper les attaques qui ne varient que par leurs contraintes sur les entrées [12] de manière à ne présenter que les groupes d'équivalence à l'utilisateur.

3.4.3.2 Faute-équivalence

La définition d'*équivalence* impose la stricte égalité des transitions des attaques. La définition *faute-équivalence* (définition 3.14) se concentre sur l'égalité de la liste des transitions fautées uniquement. Cette définition ignore ainsi les variations de chemins (transitions non fautées) entre le déclenchement des fautes.

Définition 3.14 (Faute-équivalence)**.** *Une attaque a est dite* ***faute-équivalente*** *(noté $\sim_f$) à une attaque a' si leur séquence de transitions fautées sont égales (égalité syntaxique des séquences des fautes [13]) :*
$a \sim_f a' =_{def} F(a) = F(a')$

Cette définition vise à regrouper entre-elles des attaques qui ne diffèrent que par des variations de chemins liées aux entrées mais contiennent les mêmes fautes. Parmi ces différences qui sont captées par la *faute-équivalence* :

— des attaques qui diffèrent par un branchement n'ayant pas d'impact sur l'objectif d'attaque.
— des attaques qui diffèrent par des fautes injectées à des itérations différentes dans une boucle.

La table 3.6 compare les résultats obtenus pour `null_bytes` avec les deux définitions de l'équivalence et donne les résultats obtenus pour l'inversion de test avec un objectif d'attaque visant à retourner un résultat incorrect (SDC). La ligne *attaques* correspond au total d'attaques réussies avant l'analyse d'*équivalence*. La suivante correspond aux résultats d'équivalence et la troisième aux résultats de *faute-équivalence*.

TABLE 3.6 – Comparaison des définitions de l'équivalence sur `null_bytes`

Fautes	1F	2F	3F	Total
attaques	13	5	0	18
classes d'équivalence	11	4	0	15
classes de faute-équivalence	2	2	0	4

Seules deux classes de faute-équivalence en une faute sont trouvées :

— L'attaque consistant à inverser la condition d'entrée dans la boucle `for` (ligne 3). Cette attaque ne fonctionne que si le tableau d'entrée n'est pas "[0, 0]".
— l'attaque consistant à inverser une des deux itérations afin de fausser le résultat.

La table 3.7 présente de la même manière les résultats obtenus avec la fonction `memcmps`. On peut constater que, dans cet exemple, les définitions d'équivalence et de faute-équivalence fournissent les mêmes classes, seul le regroupement des chemins non distincts a un impact [14].

12. Et potentiellement qui varient aussi au niveau du chemin dans l'encodage de l'objectif d'attaque.
13. Seuls les points d'injection sont comparés, pas les paramètres de fautes tels que la valeur d'une donnée injectée.
14. `memcmps` *n'ayant pas de boucle (fautée), les attaques ne diffèrent pas par les itérations de boucle.*

TABLE 3.7 – Comparaison des définitions de l'équivalence sur `memcmps`

Fautes	1F	2F	3F	4F	5F	Total
attaques	4	8	20	28	28	88
classes d'équivalence	1	2	5	7	7	22
classes de faute-équivalence	1	2	5	7	7	22

3.4.4 Redondance des attaques

La notion de *redondance* des attaques vise également à sélectionner les attaques qui seront à regarder en priorité par l'utilisateur. On considère qu'une attaque a' est *redondante* par rapport à une attaque a, si a' est une variation de a avec des fautes supplémentaires, noté $a < a'$. L'idée sous-jacente est que protéger les *attaques minimales* (définition 3.15), protégera probablement les attaques redondantes qui leurs sont liées.

Définition 3.15 (Terminologie)**.** *Soit $<$ un ordre partiel correspondant à une relation de redondance, une attaque $a \in T_s(P)$ est dite minimale si elle n'est redondante par rapport à aucune autre attaque :*

$$Min(a) =_{def} \nexists\, a' \in T_s(P) \mid a > a' \wedge a \neq a'$$

On note $Red(a)$ l'ensemble des attaques redondantes par rapport à a.

3.4.4.1 Préfixe et sous-mot

Lazart propose deux définitions pour la redondance des attaques : *préfixe* (définition 3.16) et *sous-mot* (définition 3.17).

Définition 3.16 (Redondance par préfixe)**.** *Une attaque a' est redondante par préfixe, noté $<_p$, par rapport à une attaque a, si le mot des transitions fautées de a ($F(a)$) est un* ***préfixe propre*** *du mot $F(a')$.*

Définition 3.17 (Redondance par sous-mot)**.** *Une attaque a' est redondante par sous-mot, noté $<_s$, par rapport à une attaque a, si $F(a)$ est un* ***sous-mot strict*** *du mot $F(a')$.*

La définition *préfixe* se concentre sur les attaques où les fautes supplémentaires sont situées après un préfixe commun, ce qui apporte une garantie que les fautes redondantes de a' n'ont pas d'impact sur le préfixe. La définition *sous-mot* autorise que des fautes redondantes soient injectées entre les fautes communes.

La table 3.8 présente les résultats de l'analyse de redondance et d'équivalence sur l'exemple `memcmps`. Chaque colonne indique le nombre d'attaques obtenues pour un nombre de fautes donné, la dernière colonne indiquant le total. La première ligne correspond au nombre d'attaques réussies pour chaque nombre de fautes et la seconde aux attaques réussies minimales. Les deux dernières lignes correspondent respectivement aux classes d'équivalence et aux classes d'équivalence minimales, avec la définition de faute-équivalence. Les classes d'équivalence minimales sont des classes d'équivalence d'attaques minimales [15]. Se concentrer sur les classes d'équivalence minimales, permet donc de réduire le nombre d'attaques à considérer pour l'utilisateur.

15. Les attaques d'une classe d'équivalence sont toutes minimales ou toutes redondantes.

TABLE 3.8 – Attaques minimales trouvées pour `memcmps`

Fautes	1F	2F	3F	4F	5F	Total
Attaques réussies	0	4	8	20	28	88
Attaques réussies minimales	0	4	4	8	0	16
Classes fautes-équivalence	0	1	2	5	7	22
Classes fautes-équivalence minimales	0	1	1	2	0	4

3.4.4.2 Combinaison des notions d'équivalence et de redondance

La recherche des attaques minimales nécessite simplement de déterminer si chaque attaque est redondante par rapport à au moins une attaque, mais pas nécessairement de trouver $Red(a)$ pour chaque attaque. Lorsque seules les attaques minimales sont recherchées, on parlera d'analyse de redondance *paresseuse*. L'analyse de redondance, qui consiste à trouver l'ensemble $Red(a)$ pour chaque attaque, revient à calculer le graphe de redondance associant chaque attaque aux attaques qui lui sont redondantes. Cela permet de calculer les attaques maximales, centrales et isolées (définition 3.18).

Définition 3.18 (Terminologie)**.** *Une attaque t est dite maximale si aucune attaque n'est redondante par rapport à elle :*

$$Max(a) =_{def} \nexists a' \in T(P) \mid a < a' \wedge a \neq a'$$

Une attaque a est dite centrale si elle n'est ni minimale, ni maximale :

$$Mid(a) =_{def} \neg Min(a) \wedge \neg Max(a) \wedge a \neq a'$$

Une attaque a est dite isolée si elle est minimale et maximale. :

$$Isol(a) =_{def} Min(a) \wedge Max(a) \wedge a \neq a'$$

Les attaques minimales sont supposées être plus prioritaires à examiner ou protéger. A l'inverse, les attaques maximales ont assez peu d'intérêt pour la protection et l'évaluation de la robustesse d'un programme, mais peuvent être utilisées comme métriques pour certaines heuristiques visant à retirer des points d'injection les moins importants par exemple, tout comme les nombres d'attaques centrales ou isolées (voir section 3.5).

L'analyse d'équivalence et l'analyse de redondance sont effectuées en même temps pour des raisons d'efficacité [16]. Ainsi, la suite de ce manuscrit parlera d'analyse de *redondance-équivalence*. La figure 3.6 présente différentes classes d'attaques obtenues en combinant les résultats fournis par les notions d'équivalence et de redondance.

Elle positionne chaque classe en fonction de celles qui devraient être favorisées par l'utilisateur et donc examinées en priorité. En abscisse, les définitions d'équivalence et de faute-équivalence constituent les deux paliers associés à l'ensemble des attaques (sans notion de redondance), les classes faute-équivalence devant être favorisées par l'utilisateur. En ordonnées, les classes d'attaques sont également ordonnées en fonction de la priorité, avec les classes minimales à favoriser par l'utilisateur (une seule définition de redondance est présentée par soucis de simplicité).

16. Le chapitre 4 rentre plus en détail dans la complexité de ces analyses.

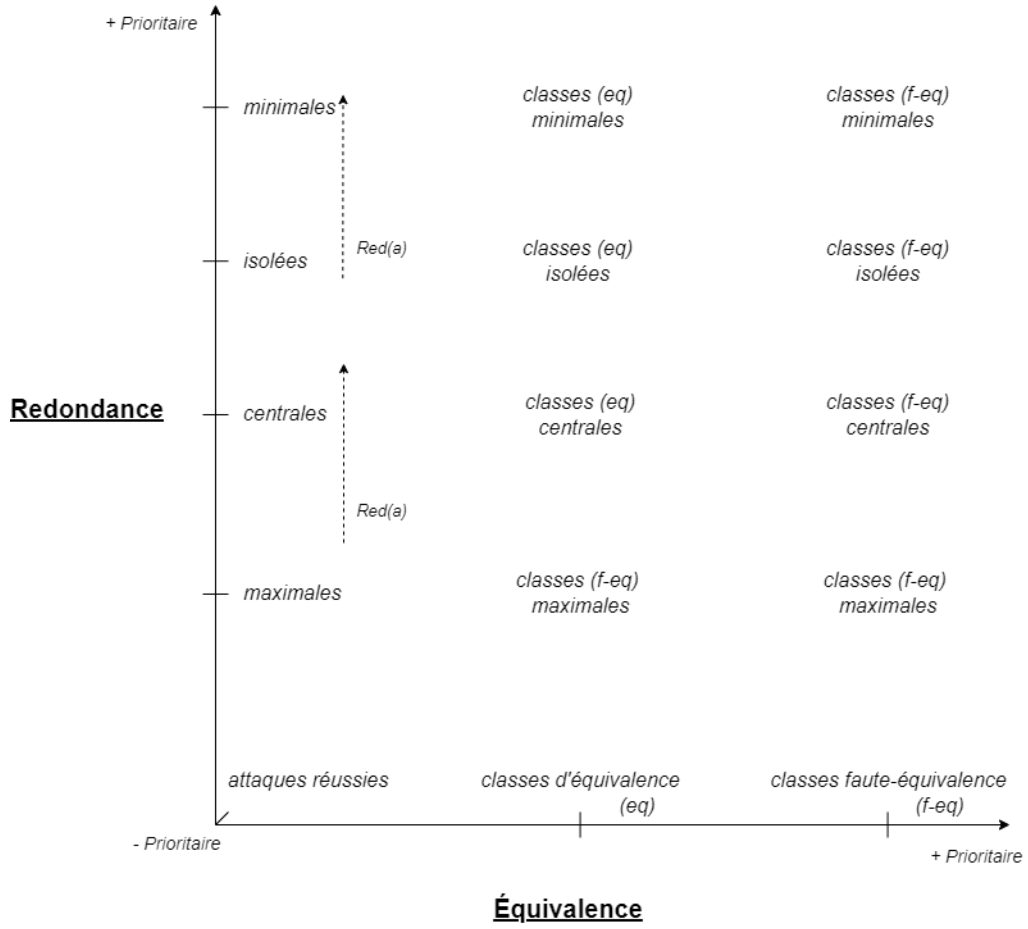

FIGURE 3.6 – Indicatif de priorité des différentes classes d'attaques

3.4.4.3 Autres définitions

Cette section discute des différentes définitions d'équivalence et de redondance proposées par Lazart et des autres définitions qui pourraient être considérées par l'utilisateur.

Les définitions d'équivalence et redondance présentées précédemment ne prennent pas en compte le type de terminaison des traces. En effet, on considère que si différents types de terminaison sont considérés comme des attaques par l'utilisateur, alors ces types de terminaison sont équivalents. Il est possible de considérer des définitions séparant les attaques en fonction de leur terminaison, par exemple en différenciant les traces terminant en validant l'objectif d'attaque (Satisfy) et les erreurs à l'exécution (RunTime Error (RTE))).

Les notions de faute-équivalence et les définitions de redondance considèrent uniquement les fautes, et font donc abstraction des variations de chemins entre les injections de fautes. Cette abstraction est très générale, et si un utilisateur souhaite capter *uniquement* des variations liées à l'itération de boucle par exemple, alors il est nécessaire d'utiliser une définition plus spécifique. Cela pourrait être implémenté en vérifiant que les transitions non fautées entre deux fautes sur un même point d'injection ne sortent pas d'une boucle par exemple.

Au delà de ces considérations, le concept de redondance pourrait être raffiné. La figure 3.6 présentée précédemment inclut $Red(a)$ comme paramètre de la priorité des attaques à

regarder. L'idée est que dans une même classe d'attaques, il peut être intéressant de regarder en priorité les attaques avec $Red(a)$ élevé puisque corriger ces attaques impliquera potentiellement la correction des attaques redondantes dans $Red(a)$. Et à l'inverse, les attaques maximales, dans un contexte où l'on souhaite écarter les attaques les moins « prioritaires », pourraient être classées par $Red(a)$ décroissant.

Des définitions se basant sur la relation $Max(a) < Mid(a) < Isol(a) < Min(a)$ pourraient aussi être envisagées. Celles-ci pourraient être étendues à des ordres multiples, par exemple : $Max_n(a) < Max_{n+1}(a) < ... < Min_n(a) < Min_{n+1}(a)$ (avec n une limite de faute).

Lazart laisse donc l'utilisateur définir sa propre relation d'ordre partiel, constituée d'une définition d'équivalence et de redondance. Par défaut, l'analyse de redondance-équivalence utilise la définition *faute-équivalence* et la redondance en *préfixe*.

3.4.5 Analyse de points chauds

L'analyse de points chauds (*hotspots*) vise à obtenir des métriques sur les points d'injection sur un ensemble de traces. La section 3.4.5.1 présente les métriques proposées par l'analyse de points chauds. La section 3.4.5.2 discute de l'exploitation des résultats en fonction de l'ensemble de traces sur lequel l'analyse de points chauds est appliquée.

3.4.5.1 Métriques

L'analyse de points chauds parcourt un ensemble de traces d'exécution et fournit pour chaque point d'injection i et pour chaque nombre de fautes n les données suivantes :

— fc : nombre de traces en n fautes dans lesquelles le point d'injection i est déclenché.
— tc : nombre total de déclenchements de i (en comptant les multiples déclenchements au sein d'une trace).
— mc : maximum de déclenchements de i au sein d'une même trace.

Dans le cadre d'aide au placement des contre-mesures dans un programme, fc et tc permettent d'identifier les points d'injection les plus sensibles à protéger en priorité (voir chapitre 5). Dans le cadre d'une analyse de robustesse, ces deux métriques permettent de déterminer les points d'injection les plus sensibles pour une expertise manuelle ou comme guide pour une analyse plus précise (par exemple une simulation à plus bas niveau). Dans un contexte d'heuristiques de sélection de points d'injection (voir section 3.5.2), mc permet d'identifier les points d'injection qui sont situés dans des parties répétées du flot de contrôle, et dont le retrait pourrait sensiblement réduire l'explosion combinatoire des fautes. De la même façon tc (et le ratio fc/tc) peuvent indiquer qu'un point d'injection a une influence forte sur l'explosion combinatoire des chemins.

La table 3.9 présente les résultats de l'analyse de points chauds pour le programme `memcmps` jusqu'à 4 fautes. La colonne "IP" indique l'identifiant du point d'injection. La colonne "Info" précise le type de modèle et la ligne correspondant dans le code C (voir listing 3.9). Les colonnes suivantes indiquent la valeur de fc (nombre de traces déclenchant le point d'injection) pour chaque nombre de faute. Les valeurs tc et mc ne sont pas présentées pour cet exemple puisqu'il n'y a pas de trace contenant plusieurs déclenchements d'un même point d'injection [17]. Dans cet exemple, l'IP 1 est le plus sensible en 1 faute, puisqu'il intervient dans deux attaques.

17. Cela implique $fc = tc$, $tc > 0 \implies mc = 1$ et $tc = 0 \implies mc = 0$

TABLE 3.9 – Résultats d'analyse de points chauds sur `memcmps`

IP	Info	1 faute	2 fautes	3 fautes	4 fautes
0	type : ti ligne : 14	0	1	2	3
1	type : data ligne : 15	0	2	5	7
2	type : ti ligne : 16	0	0	4	7
3	type : data ligne : 17	0	0	1	3
4	type : ti ligne : 18	0	0	1	4
5	type : data ligne : 19	0	0	0	1
6	type : data ligne : 24	1	1	2	3

3.4.5.2 Application sur d'autres ensembles de traces

Par défaut, l'analyse de points chauds considère toutes les attaques réussies. Il est possible d'appliquer cette analyse sur un ensemble arbitraire de traces et en fonction de l'ensemble de traces sélectionné, les résultats peuvent être interprétés différemment :

- *attaques réussies minimales* : permet de déterminer les IPs les plus sensibles (pour expertise manuelle ou afin de placer des protections).
- *attaques réussies* :
  - déterminer les IPs les plus sensibles sans utiliser la notion de redondance.
  - déterminer les IPs les plus coûteux pour l'exécution concolique.
- *attaques réussies maximales* : les IPs qui se déclenchent le plus dans des attaques jugées non prioritaires à observer.

En comparant les résultats obtenus par des analyses de points chauds sur des ensembles de traces différents, il est possible d'obtenir d'autres propriétés :

- les IPs qui se déclenchent dans les attaques maximales mais pas dans les attaques minimales, peuvent ainsi être retirés en priorité d'une analyse en limitant le nombre d'attaques qui ne seraient pas découvertes.
- les IPs se déclenchant dans les attaques maximales en sous-mot, mais pas dans les attaques minimales en préfixe, permettent de mettre en évidence des IPs qui n'ont pas d'impact, puisqu'ils ne permettent de gagner que si d'autres fautes ont déjà permis de gagner.

L'analyse de points chauds est donc utile dans différents contextes d'analyse, et le choix de l'ensemble de traces à considérer, voir la comparaison de plusieurs analyses de points chauds, permet d'obtenir des métriques pour des cas d'usage plus précis.

3.5 Résultats et méthodologie

Lazart vise à aider l'utilisateur en traitant les résultats qui lui sont fournis. Les concepts de redondance, d'équivalence et de points chauds présentés précédemment en sont un exemple. L'API Python vise à rendre la manipulation des résultats plus simple pour l'utilisateur et divers outils, tels que la génération des graphes d'attaques, permettent aussi d'aider à l'interprétation des résultats pour l'utilisateur.

Cependant, certaines problématiques liées à l'utilisation de Lazart se posent. Naturellement les limitations dues à KLEE et à l'exécution concolique (voir section 3.1.3) telles que l'analyse de bibliothèques externes ou l'explosion combinatoire sont héritées par Lazart. La combinatoire des fautes renforce ce problème, d'autant plus en fautes multiples, et les fautes peuvent aussi introduire des chemins qui ne terminent pas (voir section 3.1.5). Le passage à l'échelle peut s'avérer difficile en fonction du code, des modèles de faute et de l'analyse considérés.

La sélection du périmètre de l'analyse est une problématique courante dans l'analyse statique de code, et qui a un impact fort sur le passage à l'échelle : ce périmètre devant parfois être réduit pour permettre que l'analyse termine dans le temps imparti. Dans le cadre de l'outil Lazart, ce périmètre correspond au choix de l'objectif d'attaque, du contexte (point d'entrée, entrées du programme), des modèles de faute (et la surface de code sur laquelle ils sont appliqués) et de la surface du programme à analyser.

Enfin, les retours sur la complétude et la correction de l'analyse ne sont pas forcément faciles à interpréter. Il peut être difficile de savoir si les réductions du périmètre de l'analyse n'ont pas pour effet de rater un chemin d'attaque (incomplétude), ou si une concrétisation a eu un effet sur la correction. Des précautions concernant le paramétrage de KLEE et de Lazart permettent cependant d'éviter certaines pertes de complétude et de correction.

Le chapitre 4 revient sur l'implémentation de Lazart et discute de ce qui a été fait au niveau de l'outil pour limiter ces problématiques. Cette section se concentre sur les résultats et leur exploitation du point de vue de l'utilisateur. La section 3.5.1 revient sur les résultats qui sont fournis par l'outil, en présentant une expérimentation réalisée sur un ensemble de programmes, et comment ces résultats peuvent être exploités par l'utilisateur. La section 3.5.2 s'intéresse à l'aspect méthodologique de l'utilisation de l'outil afin de pallier les différentes problématiques énoncées précédemment. Finalement, la section 3.5.3 discute des autres variables du périmètre d'analyse et présente l'exemple du choix de la stratégie d'exploration dans le cas d'une analyse qui est interrompue avant son terme.

3.5.1 Sorties de l'outil

Les métriques sorties par Lazart peuvent être réparties en trois catégories principales :
- Les métriques *statiques* : nombre d'instructions LLVM, nombre de branchements, nombre de points d'injection, paramètres de l'analyse...
- Les métriques d'*analyse* : nombre d'attaques, nombre d'attaques minimales, métriques de points chauds...
- Les métriques d'*exécution* : temps d'exécution, nombre de traces générées, nombre d'instructions exécutées, couverture du code...

La section 3.5.1.1 présente les différents programmes d'exemples étudiés dans cette section, ainsi que le modèle d'attaquant pour chacun d'entre-eux. La section 3.5.1.2 s'intéresse aux résultats des analyses d'attaques, de redondance, d'équivalence et de points chauds. La section 3.5.1.3 porte sur les métriques d'exécution et de couverture. Finalement, la section 3.5.1.4 s'intéresse à la couverture, la correction et le déterminisme des analyses.

3.5.1.1 Ensemble de programmes

Les programmes étudiés ici sont issus de la collection Fault Injection and Simulation Secure Collection (FISSC) 3 [18] qui est utilisée par Lazart :
- *vp* : une collection de *verify_pin*.
- *rsa* : une collection d'implémentations de l'algorithme *CRT-RSA*.

18. Correspondant à une version enrichie de la collection FISSC présentée dans [Dureuil 2016b].

— fu : une collection d'implémentations d'un chargeur de micro-programme.

Les programmes *verify_pin* sont analysés avec le modèle de l'inversion de test et l'objectif d'attaque de s'authentifier avec un PIN invalide. La table 3.10 présente les huit versions étudiées et leurs protections :

— *HB* : booléens endurcis.
— *FTL* : boucles en temps constant.
— *INL* : inlining de la fonction compare.
— *DTC* : décrémentation anticipée du compteur d'essai `try_counter`.
— *TCBK* : duplication du compteur d'essai.
— *CD* : double appel à la fonction `compare`.
— *TD* : duplication des tests.
— *SC* : compteur d'étape.

TABLE 3.10 – Différentes versions de *verify_pin*

Version	HB	FTL	INL	DTC	TCBK	CD	TD	SC
vp0	-	-	-	-	-	-	-	-
vp1	✓	-	-	-	-	-	-	-
vp2	✓	✓	-	-	-	-	-	-
vp3	✓	✓	✓	-	-	-	-	-
vp4	✓	✓	-	✓	✓	-	-	-
vp5	✓	✓	-	✓	-	✓	-	-
vp6	✓	✓	✓	✓	-	-	✓	-
vp7	✓	✓	✓	✓	-	-	✓	✓

Les expérimentations sur les programmes RSA utilisent le modèle de mise-à-0 (*reset*). L'objectif d'attaque correspond à la recherche d'une attaque de type bellcore [Boneh 2001b] visant les restes du CRT (recomposition par théorème des restes chinois). Trois versions sont étudiées [Puys 2014] :

— $rsa0$: version non protégée.
— $rsa1$: implémentation de la version Shamir [Shamir 1999].
— $rsa2$: implémentation de la version Aumuller [Aumüller 2002].

Deux versions de fu seront étudiées : $fu1$ et $fu2$. $fu1$ correspond à la version du programme utilisée dans [Boespflug 2020], utilisant une duplication de test systématique, et est analysée avec la combinaison du modèle de mutation de données non contrainte (symbolique) et du modèle de l'inversion de test. Les expérimentations se concentrent sur l'objectif d'attaque correspondant à mettre à jour un micro-programme avec un code corrompu ou à ne pas mettre à jour le micro-programme malgré l'intégrité du code d'entrée. $fu2$ inclut des protections telles qu'un code détecteur d'erreur dans les pages du micro-programme et des vérifications d'intégrité (voir annexe B.3). Ce programme est analysé avec les modèles TI et DL séparément [19], les résultats correspondants seront respectivement notés $fu2\ TI$ et $fu2\ DL$. L'objectif d'attaque consiste à charger au moins une donnée corrompue (soit une valeur invalide dans le micro-programme, soit l'écriture en dehors de l'espace mémoire du micro-programme).

19. L'analyse ne termine pas en moins de 12h en combinant les deux modèles.

3.5.1.2 Résultats des analyses d'attaque

La table 3.11 indique les résultats obtenus avec Lazart pour les analyses. Pour chaque programme d'exemple (colonne "Nom"), la colonne "LoCs" correspond au nombre de lignes de code du programme étudié [20] [21], la colonne "IPs" au nombre de points d'injection et la colonne "Det" aux nombre de points de détection des contre-mesures. La colonne "Attaques" correspond aux attaques réussies totales pour chaque nombre de fautes tandis que la colonne "Minimales" correspond aux attaques minimales pour une définition en sous-mot. Pour chaque ligne, la colonne "Eq" indique l'équivalence des attaques : "atq" correspond aux attaques indépendemment de l'équivalence et "f-eq" correspond aux nombre de classes de faute-équivalence.

Les pourcentages de réduction des attaques présentées pour l'utilisateur à l'aide de la faute-équivalence et de la redondance sont indiqués en bas de la table. On constate qu'en moyenne, la combinaison de la faute-équivalence et de la redondance permet de diviser par un facteur de plus de 100 le total de traces d'attaques à étudier. Pour les exemples sur RSA, l'analyse de faute-équivalence n'obtient que des groupes d'équivalence d'un élément. Cela peut s'expliquer par l'absence de points d'injection dans les boucles. En revanche, l'analyse de redondance parvient à réduire fortement les attaques, seules les attaques en une faute étant minimales pour $rsa0$ et $rsa1$.

La colonne "TA%" indique la part du temps d'exécution de l'analyse qui est consacrée à l'analyse d'équivalence et de redondance. Cette valeur correspond au temps pour la recherche de toutes les relations de redondance et d'équivalences. Les analyses d'attaques et de points chauds sont peu coûteuses par rapport à l'analyse de redondance-équivalence qui passe à l'échelle plus difficilement, leurs durées ne sont donc pas présentées ici. En moyenne, l'analyse redondance-équivalence compte pour 39% du temps d'analyse, mais ce pourcentage varie très fortement d'un programme à l'autre, négligeable dans certains exemples mais très coûteux dans d'autres comme vp5. Savoir si l'analyse de redondance-équivalence va effectivement réduire de façon significative le nombre d'attaques présentées à l'utilisateur est difficile, puisqu'au delà de la complexité du programme, la structure du graphe de redondance et le nombre de classes d'équivalence peut fortement varier.

3.5.1.3 Métriques d'exécution

Un certain nombre de métriques liées à l'exécution de l'analyse sont fournies par Lazart : les métriques de *performance*, les métriques liées aux *chemins* et les métriques de *couverture*.

Les métriques de performance correspondent tout d'abord aux temps d'exécution des différentes étapes de l'analyse : compilation, mutation, exécution concolique, lecture des traces et les différentes traitements. KLEE fournit aussi une découpe de son temps d'exécution détaillant la durée de certaines étapes de son exécution (temps de fork, temps d'appel au solveur...) ainsi que des informations concernant l'usage mémoire par exemple.

Les métriques liées aux chemins donnent des informations sur le parcours des chemins dans l'exécution symbolique :

- Le nombre de ktests générés par KLEE.
- Le nombre de traces générées par Lazart.
- Le nombre de chemins terminés (*completed paths*), ce qui correspond aux nombre de ktests où l'objectif d'attaque est validé.
- Le nombre de chemins partiellement explorés (interrompus par un timeout ou une erreur d'exécution par exemple).

20. Cette valeur n'est pas une sortie de l'outil et a été calculée manuellement.
21. En ne prenant pas en compte le code nécessaire à l'analyse (script d'analyse et instrumentation).

TABLE 3.11 – Analyse d'attaque sur le benchmark d'exemple

Programme				Eq	Attaques					Minimales					
Nom	LoCs	IPs	Dets		1F	2F	3F	4F	Total	1F	2F	3F	4F	Total	TA%
vp0	25	4	0	atq :	12	21	18	7	58	12	0	0	0	12	4
				f-eq :	3	3	3	3	12	3	0	0	0	3	
vp1	28	5	0	atq :	12	21	18	7	58	12	0	0	0	12	6,5
				f-eq :	3	3	3	3	12	3	0	0	0	3	
vp2	39	7	2	atq :	34	122	204	183	543	34	4	0	0	38	48
				f-eq :	3	4	7	0	14	3	1	0	0	4	
vp3	35	7	1	atq :	34	122	204	183	543	34	4	0	0	38	47
				f-eq :	3	4	7	0	14	3	1	0	0	4	
vp4	45	11	6	atq :	34	118	180	147	479	34	0	4	0	38	41
				f-eq :	3	3	5	7	18	3	0	1	0	4	
vp5	37	7	5	atq :	0	80	644	2566	3290	0	80	124	16	220	74
				f-eq :	0	9	20	48	77	0	9	6	1	16	
vp6	39	8	6	atq :	4	40	122	199	365	4	34	0	0	38	10
				f-eq :	1	4	4	7	16	1	3	0	0	4	
vp7	78	8	8	atq :	4	36	116	173	329	4	30	0	0	34	21
				f-eq :	1	3	3	4	11	1	2	0	0	3	
rsa0	65	15	0	atq :	7	37	151	425	620	7	0	0	0	7	40
				f-eq :	7	37	151	425	620	7	0	0	0	7	
rsa1	92	36	0	atq :	5	25	118	423	571	5	0	0	0	5	36
				f-eq :	5	25	118	423	571	5	0	0	0	5	
rsa2	109	52	0	atq :	5	22	106	420	553	5	3	5	19	32	37
				f-eq :	5	22	106	420	553	5	3	5	19	32	
fu1	93	23	0	atq :	1	72	915	8191	9179	1	32	915	8191	9139	61
				f-eq :	1	8	9	25	43	1	8	9	13	31	
fu2 TI	126	7	0	atq :	17	119	425	1031	1592	17	2	10	53	82	80
				f-eq :	2	3	7	14	26	2	1	2	3	8	
fu2 DL	126	4	0	atq :	16	344	3553	23324	27237	16	344	1576	5624	7560	52
				f-eq :	1	5	19	67	92	1	5	11	27	44	
Moyennes				atq :	13	84	484	2663	3244	13	38	188	993	1233	**39%**
				f-eq :	3	10	33	103	149	3	2	2	5	12	
						f-eq / atq :			**4,58%**			f-eq / min :		**0,97%**	
						min / atq :			**37,9%**			min / f-eq :		**8,08%**	
												f-eq_min / atq :		**0,37%**	

TABLE 3.12 – Métriques d'exécution pour le benchmark d'exemple

Nom	EI	Chemins					Couverture (%)			Temps (s)			
		TR	KT	CP	EP	PP	Instrs	BBs	BRs	DSE	TP	TA	Tot
vp0	2125	9	11	9	17	11	100	100	100	0.05	0.02	0.002	00.345
vp1	2647	9	12	9	23	18	97.73	100	100	0.06	0.015	0.001	00.367
vp2	18085	51	62	51	171	149	97.59	100	100	0.14	0.123	0.008	04.230
vp3	17855	51	62	51	171	149	97.41	100	100	0.12	0.105	0.008	04.268
vp4	20259	39	54	39	198	206	96.73	100	100	0.15	0.076	0.006	04.175
vp5	149490	197	626	197	964	1781	98.25	100	100	0.61	0.482	0.161	01 :27
vp6	17769	32	55	32	169	172	97.86	100	100	0.12	0.068	0.005	03.221
vp7	25205	23	34	23	260	328	95.57	100	100	0.22	0.047	0.003	04.112
rsa0	769534	621	622	621	673	673	81.76	76.67	76.92	1.22	1.142	1.692	4.242
rsa1	421298	572	573	572	1597	1597	56.79	50	100	1.32	1.095	1.504	4.147
rsa2	507016	554	557	554	1104	1104	55.2	47.73	75	1.26	1.032	1.505	4.079
fu 1	36632126	9179	9183	9179	137670	128491	97.06	91.67	85.14	08 :25	31.366	13 :58	22 :46
fu 2 TI	1424370	1592	1597	1592	2783	1191	98.51	95	90	4.61	3.492	34.685	43.47
fu 2 DL	15222597	27237	27240	27237	59807	32570	96.97	92.5	85	1:21:12	01 :12	1:32:06	2:54:46
vp0 sym	12734	58	61	58	130	75	100	100	100	0.17	0.103	0.024	00.345
vp1 sym	16856	58	62	58	178	132	97.97	100	100	0.19	0.106	0.024	00.367
vp2 sym	190112	543	633	543	1930	1671	97.82	100	100	0.97	1.057	2.047	04.230
vp3 sym	187944	543	633	543	1930	1671	97.67	100	100	0.96	1.119	2.040	04.268
vp4 sym	229512	479	578	479	2419	2387	97	100	100	1.29	1.003	1.730	04.175
vp5 sym	2057305	3290	9483	3290	14213	24763	98.42	100	100	12.61	7.189	01 :05	01 :27
vp6 sym	188349	365	547	365	1941	1909	98.08	100	100	0.94	1.31	0.844	03.221
vp7 sym	302605	329	450	329	3279	3954	95.88	100	100	1.9	1.175	0.868	04.112

— Le nombre de chemins explorés (chemins terminés + chemins partiellement explorés).

KLEE propose aussi des retours sur la couverture du code : couverture des instructions, des blocs de base et des branchements. Lazart calcule également la couverture des points d'injection avec l'analyse de points chauds.

La table 3.12 présente les résultats obtenus sur l'ensemble des expérimentations. Les programmes `verify_pin` sont présentés en version avec entrées fixes et symboliques (vpX sym) pour comparaison (la table des résultats d'attaques 3.11 considère uniquement les versions symboliques). La colonne "Nom" correspond au programme et la version considérée. La colonne "EI" indique le total d'instructions exécutées par KLEE (au niveau LLVM). Les colonnes "TR" correspond aux nombre de traces générées par Lazart. Pour les paramètres utilisés dans ces exemples, cette valeur est égale au nombre de chemins terminés. La colonne "KT" indique le nombre de ktests générés par KLEE. Les colonnes "EP" et "PP" correspondent respectivement aux chemins explorés et aux chemins partiellement explorés. Les valeurs "EP" et "EI" permettent d'avoir une indication sur la complexité du code étudié.

Les colonnes "Couverture" indiquent dans l'ordre la couverture des instructions ("Instrs"), des blocs de base ("BBs") et des branchements ("BRs") sous la forme de pourcentage. La couverture du code sur les programmes `verify_pin` est maximale pour les blocs de base et les branches, même dans le cas d'entrées symbolique. La couverture des instructions est la plus précise et donc celle qui fournit les valeurs les plus faibles. Seuls les programmes RSA donnent une couverture faible, de l'ordre de 50% pour les blocs de base et les instructions pour $rsa2$, ce qui peut s'expliquer par le fait que les entrées soient fixées.

Des métriques de temps sont données pour l'exécution symbolique ("DSE"), la lecture des traces ("TP") et pour les différentes analyses ("TA") : analyse d'attaque, analyse de redondance et d'équivalence et analyse de points chauds. La durée des étapes de la compilation et de la mutation sont négligeables et ne sont pas mentionnées. L'exécution symbolique constitue la part principale des analyses, mais l'analyse de redondance-équivalence peut s'avérer coûteuse dans certains exemples (comme pour vp5 ou fu2).

3.5.1.4 Déterminisme, complétude et correction

Cette section s'intéresse au déterminisme des analyses, ainsi qu'aux résultats et métriques fournies par Lazart en ce qui concerne la complétude et la correction d'une analyse (voir définitions section 3.1.5).

Déterminisme L'exécution concolique et les solveurs SMT étant fortement basés sur des heuristiques, qui peuvent inclure de l'aléatoire [22], le déterminisme n'est pas garanti par Lazart. Refaire la même analyse peut donc potentiellement fournir des résultats différents. Les expérimentations réalisées tendent à faire penser que ce non-déterminisme reste relativement rare et davantage présent dans les analyses complexes [23].

Couverture et complétude Lazart effectue une émulation des fautes de telle sorte que l'exploration des chemins et des fautes est complète si l'exécution symbolique explore tous les chemins du mutant. Cependant la complétude peut être perdue à différents niveaux :

— au niveau du périmètre de l'analyse et de la configuration de l'outil.

— au niveau de l'exécution concolique elle-même (voir section 3.1.3 : timeout, heuristiques incomplètes).

22. Comme la *random path selection* utilisée par KLEE [Cadar 2008a].

23. Notez cependant qu'il s'agit là d'un rapport principalement manuel, l'outil ne vérifiant pas le déterminisme automatiquement.

Lazart fournit certaines métriques concernant la complétude, mais cette évaluation reste difficile. Les métriques de couverture fournies par KLEE correspondent à la couverture du programme muté. La couverture des instructions et des blocs de base indiquent donc la couverture du programme original modifié en y incluant le code de simulation des fautes et l'instrumentation de l'analyse. De la même manière, les informations concernant les chemins (EP, PP, CP) présentées précédemment considèrent les chemins du programme muté.

Si une couverture incomplète des points d'injection dans le mutant est un signal que la complétude a pu être perdue, une couverture complète des IPs ne veut pas dire que l'analyse est complète. En effet, si deux chemins peuvent déclencher un point d'injection, il suffira qu'un seul des deux soit explorés pour que le point d'injection soit considéré comme couvert. Lazart tente donc de signaler au maximum les possibles pertes de complétude afin que l'utilisateur puisse estimer la complétude de l'analyse. Lazart indique aussi d'autres métriques telles que les analyses qui ont été interrompues par l'utilisateur ou bien les traces qui n'ont pas pu être rejouées pour cause de timeout [24], qui constituent une autre indication concernant la complétude.

Cependant, si aucune des métriques ne montre une perte de complétude, seule une validation manuelle permet d'avoir des garanties sur le fait que toutes les exécutions fautées ont effectivement été explorées.

Correction La correction est une propriété plus délicate puisque très peu de retours sont fournis par KLEE et Lazart. Si l'exécution symbolique classique est *correcte*, l'exécution concolique peut être incorrecte en raison de certaines concrétisations ne préservant pas la correction.

KLEE peut donner des informations sur des concrétisations incorrectes, par exemple via la terminaison de traces de type "model". Cela étant, la validation d'un chemin d'attaque par l'utilisateur reste nécessaire.

3.5.2 Méthodologie et chaîne d'outils

Cette section s'intéresse à l'aspect méthodologique de l'utilisation de Lazart vis-à-vis des problématique présentées en début de section 3.5 : passage à l'échelle, spécification du périmètre d'analyse et les pertes de complétude.

La méthodologie proposée dans [Lacombe 2021, Lacombe 2023] sera prise pour exemple pour discuter des solutions et approches existantes pour limiter ces problématiques. Cette méthodologie vise à limiter la combinatoire des fautes en réduisant le nombre de points d'injection de fautes dans l'analyse. Cette approche peut être vue comme une instance particulière d'une méthodologie reposant sur trois grands principes :

— la réduction du périmètre d'analyse par sur-approximation.
— l'utilisation d'une chaîne d'outils combinant différentes techniques d'analyse.
— une approche itérative par raffinement du périmètre à l'aide d'heuristiques.

Des expérimentations ont été conduites sur les programmes *vp*6 et *vp*9 (voir section 3.5.1.1), la commande sudo (de Linux-PAM) et la bibliothèque iso7816 du projet Wookey [ANSSI 2018]. Ces expérimentations utilisent le modèle XOR (présenté dans le chapitre 2) consistant à fauter les expressions au niveau CIL en effectuant une opération XOR avec une donnée fautée.

La section 3.5.2.1 présente la méthodologie proposée. La section 3.5.2.2 s'intéresse à l'analyse de dépendance, les sections 3.5.2.3 et 3.5.2.4 aux heuristiques qui peuvent être appliquées pour réduire davantage l'ensemble des points d'injection à analyser.

24. Notez qu'un ktest peut avoir le type de terminaison différent de *timeout* (le chemin étant terminé du côté de l'exécution symbolique) sans pouvoir être rejoué dans une limite de temps imparti.

3.5.2.1 Présentation générale

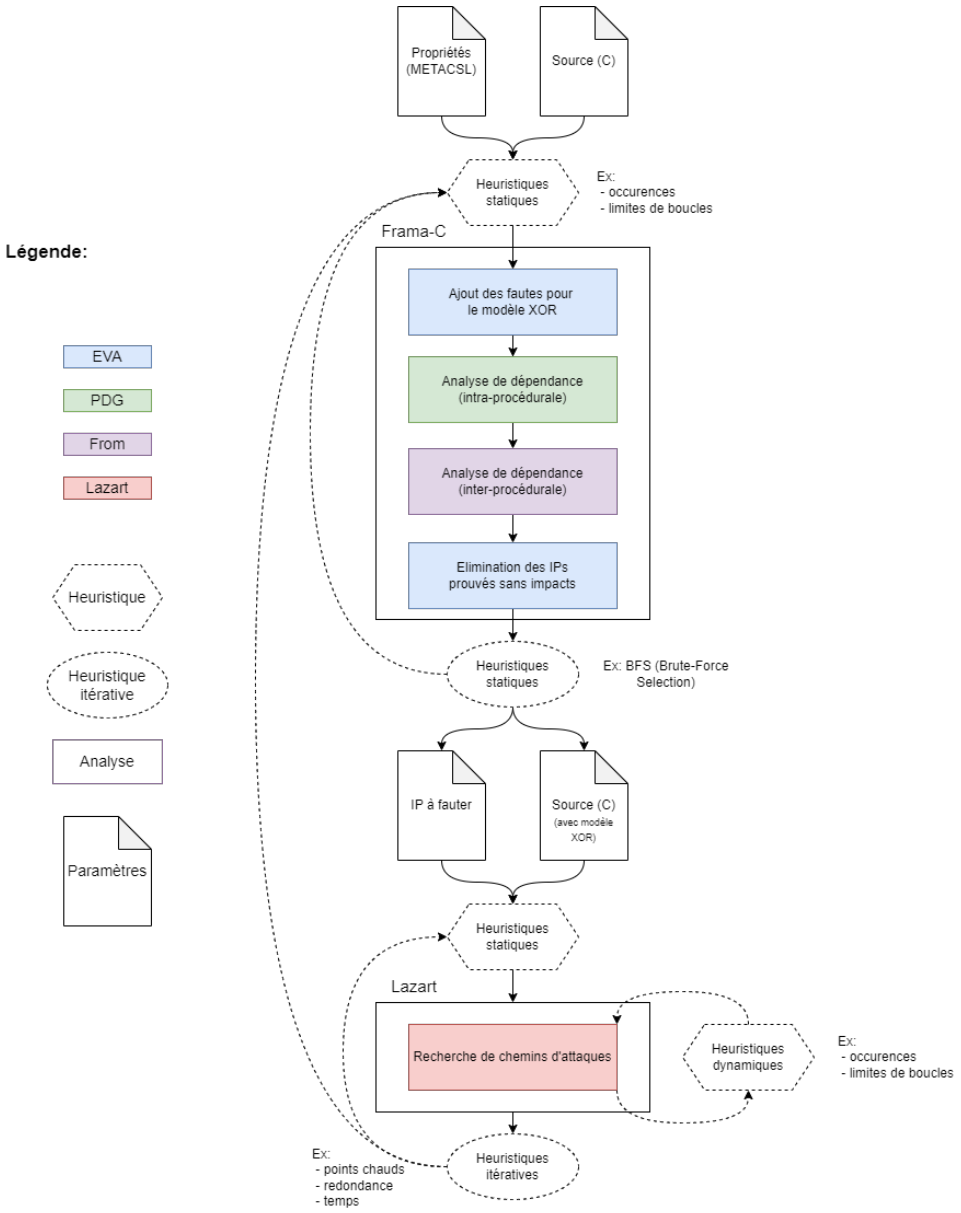

FIGURE 3.7 – Schéma général de la méthodologie [Lacombe 2023]

La figure 3.7 présente l'architecture générale de la méthodologie proposée. Cette méthodologie utilise l'analyse statique afin de déterminer quels sont les points d'injection qui ont un impact sur la propriété de sécurité étudiée. Cet ensemble de fautes est ensuite transmis à Lazart qui recherche des chemins d'attaques. Cette approche est présentée dans la section 3.5.2.2. Des heuristiques ont aussi été proposées afin de réduire encore l'espace des fautes à injecter. Ces heuristiques peuvent être séparées en deux groupes :

— Les heuristiques *simples* qui s'appuient sur les résultats d'une unique exécution d'une analyse (section 3.5.2.3).

— Les heuristiques *itératives*, qui utilisent une approche itérative pour raffiner le périmètre d'analyse à chaque appel d'un outil, présentées dans la section 3.5.2.4.

3.5.2.2 Approximation des points d'injection de fautes utiles

Cette étape prend en entrée le programme source C et les propriétés à vérifier sous la forme d'assertions (ANSI/ISO C Specification Langage (ACSL)). Celles-ci peuvent être définies par l'utilisateur, ou générées automatiquement : les expérimentations conduites utilisent le plugin RTE de Frama-C permettant de placer automatiquement des assertions contre les erreurs d'exécution.

L'étape d'analyse statique se déroule en quatre sous-étapes :

— 1) Simulation des points d'injection avec le modèle XOR en introduisant une variable non initialisée (déclarée comme `extern`) par point d'injection. Cette étape est effectuée avec EVA qui utilise l'interprétation abstraite (et fonctionne donc par sur-approximation).

— 2) Génération des graphes de dépendances intra-procédural, à l'aide du plugin Program Dependency Graph (PDG) de Frama-C (basé sur EVA).

— 3) Calcul du graphe de dépendances inter-procédural à l'aide du plugin From.

— 4) Calcul de dépendances pour déterminer quels points d'injection peuvent impacter chaque assertion.

Ainsi, on ne conserve que les points d'injection qui ont un impact (d'après l'analyse de dépendance) sur les assertions non prouvées par EVA. Cette réduction de l'espace de faute préserve la complétude puisque celle-ci est effectuée en considérant toutes les fautes possibles.

La table 3.13 contient les résultats obtenus pour nos programmes de test. La colonne "Nom" indique le programme en question, "IPs" le nombre de points d'injection dans le programme et "Dets" le nombre de points de vérification de contre-mesure (déclenchant un arrêt de la carte si une attaque est détectée). La colonne "Lazart (seul)" indique le nombre d'attaques totales trouvées "AP", le nombre de chemins explorés ("EP") et la durée de l'analyse dans le cas où Lazart est lancé sur toutes les fautes générées par le modèle XOR. La colonne "AS" correspond à l'étape d'analyse statique (dépendances des propriétés) indiquant le temps d'exécution ("Temps") et le nombre de points d'injection restant après l'analyse ("IPs"). La colonne Heuristiques indique les heuristiques appliquées ("Type"), le temps d'exécution ("Temps") et le nombre de points d'injection restant après l'application de ces heuristiques ("IPs"). Enfin, la colonne "Lazart" présente les résultats obtenus avec les points d'injection restants.

Les programmes "vp0" et "vp7" ont été testés sans heuristique (les valeurs "-" indiquant qu'une étape n'est pas effectuée) et permettent d'illustrer un gain de temps non négligeable avec cette analyse de dépendance préliminaire. L'interprétation abstraite passe communément mieux à l'échelle et c'est pourquoi le temps d'analyse reste inférieur à une seconde pour ces exemples. Le gain est ainsi bien supérieur pour l'exemple "vp7" qui réduit d'un facteur 10 le nombre de points d'injection à analyser pour Lazart, pour un temps d'analyse trois fois moindre.

Table 3.13 – Résultats obtenus pour l'analyse de dépendance

Programme			Lazart (seul)			AS		Heuristique			Lazart		
Nom	IPs	Dets.	AP	EP	Temps	Temps	IPs	Type	Temps	IPs	Attaques	EP	Time
vp0	20	0	5	65	**14s**	1s	10	-	-	-	5	52	**11s**
vp7	142	31	6	3253	**1h43**	1s	15	-	-	-	6	1051	**30min**

3.5.2.3 Heuristiques simples

La méthodologie [Lacombe 2023] propose une heuristique de sélection basée sur le nombre maximal d'occurrences des points d'injection dans une exécution nominale (sans injection de faute). Cette heuristique est implémentée avec EVA, en amont de l'analyse statique de dépendance. Cette heuristique fait la supposition qu'un point d'injection qui se déclenche souvent en l'absence de faute, a plus de risque de provoquer une explosion combinatoire des chemins fautés et donc la non-terminaison de l'exploration concolique. Cette heuristique n'est valable qu'en faute unique puisque dans un contexte de fautes multiples, les occurrences des points d'injection peuvent fortement varier.

Une autre heuristique consiste à instrumenter le programme de manière à limiter l'exploration des exécutions. Lazart propose des macros d'instrumentation qui coupent l'exploration une fois qu'une limite fixée a été atteinte (par exemple pour limiter les itérations d'une boucle). Lazart n'a cependant pas de solution directe pour définir des limites de fautes différentes pour certains modèles de faute (seule une limite globale de fautes peut être définie). Cela permettrait de limiter l'occurrence de certains points d'injection sans pour autant les ignorer complètement, afin de ne pas passer à côté d'attaques combinées les impliquant.

La sélection par force brute (Brute Force Selection (BFS)) est une heuristique qui a été proposée dans [Lacombe 2021] et qui nécessite plusieurs appels à l'analyseur statique. BFS consiste à tester indépendamment si un point d'injection a un impact sur les propriétés, en désactivant tous les autres IPs. Les points d'injection qui n'ont pas d'impact sur les propriétés sont retirés. Cependant, cette approche n'est valide que dans un contexte de faute unique puisque chaque point d'injection est testé indépendamment. Une version multi-fautes nécessiterait d'effectuer autant de fois l'analyse qu'il existe de combinaisons de fautes possibles, ce qui revient à décaler la problématique de l'explosion combinatoire des fautes sur l'analyse de dépendances plutôt que sur l'exécution symbolique. La table 3.14 indique les résultats obtenus avec l'heuristique BFS pour l'exemple *iso7816* où près de 60% des points d'injection peuvent être retirés, en faute unique.

3.5.2.4 Heuristiques itératives

Les heuristiques itératives consistent à raffiner l'espace de faute à injecter au fur et à mesure des analyses. Celles-ci peuvent utiliser l'approche :

— par *élargissement du périmètre d'analyse* : le périmètre est étendu à chaque itération.
— par *réduction du périmètre d'analyse* : le périmètre est réduit à chaque itération.
— par *combinaison* de réductions et d'élargissement.

Nous avons ainsi proposé une approche par réduction appelée Strategy Shrinking (SS) qui consiste à interrompre l'analyse après un temps donné et retirer les points d'injection qui sont les plus déclenchés. D'autres métriques produites par l'analyse de points chauds pourraient être considérées pour déterminer les points d'injection à retirer, comme par exemple enlever les points d'injection qui se déclenchent plusieurs fois par attaque (cela étant principalement lié aux boucles). Si l'analyse est interrompue mais produit tout de même des attaques, il est possible de retirer les points d'injection apparaissant le plus dans les attaques maximales, puisqu'il s'agit à priori des attaques les moins intéressantes à explorer. Cependant, ces approches par réduction nécessitent de déterminer la durée de seuil du timeout, ce qui n'est pas évident. Dans les cas où seuls quelques points d'injection sont problématiques, ce type d'heuristiques peut fortement aider l'analyse.

L'approche par élargissement que nous avons proposée, appelée Strategy Widening (SW), vise à répéter l'analyse avec Lazart en ajoutant les points d'injection un par un. Si l'analyse dépasse le timeout spécifié lorsqu'un ajoute un point d'injection, alors celui-ci

est retiré et un autre point d'injection est testé. Il est nécessaire de prévoir une marge par rapport à la durée de l'itération précédente puisque l'ajout d'un point d'injection va nécessairement augmenter le nombre de chemins. Celle-ci a été fixée à 150% dans nos expérimentations.

La table 3.14 présente les résultats obtenus pour les exemples *iso7816* et *sudo*. L'exemple sudo a été testé avec la limite des occurrences de points d'injection (réduction) fixée à 1 ("O1") et à 5 ("O5"), et avec l'approche "SW" (élargissement). iso7816 a été expérimenté en faute unique avec BFS et SW, combinée à l'analyse de dépendances (section 3.5.2.2). L'approche BFS, qui utilise ici un timeout, est très performante car elle réduit très fortement le nombre de points d'injection.

TABLE 3.14 – Résultats obtenus pour la méthodologie

Programme		Lazart (seul)			AS		Heuristique			Lazart		
Nom	IPs	AP	EP	Temps	Temps	IPs	Type	Temps	IPs	Attaques	EP	Time
sudo	919	17	737	11	N/A	N/A	SW	23min	913	17	737	11s
	919	17	737	11s	1s	104	SW	3min	102	17	670	11s
	919	17	737	11s	-	-	SW+O1	3min	40	10	602	10s
	919	17	737	11s	1s	104	SW+O5	4min	47	10	302	10s
iso7816	153	-	-	-	-	-	SW	16min	660 → 655	1	192	19s
	153	-	-	-	153	3s	SW	6min	153 → 151	1	170	12s
	153	-	-	-	1	55s	BFS	-	-	1	45	1s

3.5.3 Périmètre d'analyse de Lazart et analyse partielle

La méthodologie présentée dans [Lacombe 2023] se concentre sur la sélection des points d'injection sur lesquels une faute pourra être injectée. Le modèle XOR englobant la combinaison des modèles d'inversion de test et de données symboliques non contraintes, celui-ci décale la problématique de la sélection des modèles de faute à appliquer vers la problématique de la sélection des points d'injection à activer. Comme cela a été évoqué précédemment, le périmètre d'analyse de Lazart inclut aussi d'autres paramètres, notamment le choix de l'objectif d'attaque ou encore de la surface de code à analyser.

Le choix du périmètre d'analyse initial reste une difficulté pour l'utilisateur de l'outil et par exemple, les centres d'évaluation s'appuient en grande partie sur l'expertise des évaluateurs. Des méthodes telles que les *stubs*[25] peuvent être utilisées pour aider à sélectionner les portions du programme à analyser. L'état de l'art des attaques en fautes permet d'aider à choisir les objectifs d'attaque et modèles de faute à étudier. Certains outils d'analyse non liés à l'analyse de fautes peuvent aider à la définition du périmètre d'analyse pour Lazart. Par exemple, l'analyse de dépendances d'EVA peut permettre ainsi de déterminer quelles fonctions du programme ciblé ont un impact sur les propriétés étudiées et de réduire l'analyse à cette portion. Lancer KLEE sur le programme sans injecter de faute (hors de Lazart donc) permet de déterminer si certaines portions du programme posent d'ores et déjà problème pour l'exécution concolique alors qu'aucune faute n'est injectée.

Lorsqu'une analyse est trop complexe pour terminer dans un temps imparti, il est néanmoins possible de récupérer des chemins d'attaques en laissant l'exécution symbolique s'exécuter avec un timeout. Dans ce cas, le choix de la stratégie d'exploration est d'autant plus importante que celle-ci peut fortement influer sur les chemins d'attaques qui seront trouvés

25. Le stub consiste à émuler l'effet d'une partie du programme. Par exemple en remplaçant le corps d'une fonction par un simple retour d'une valeur. Dans ce cas, l'opération est incomplète mais il est possible de préserver la complétude en retournant une variable symbolique dans le cas de Lazart, ce qui revient à faire une sur-approximation des comportements.

dans le temps imparti. La suite de cette section présente un cas d'utilisation d'une exécution concolique interrompue avec le programme RSA.

Les programmes RSA (du FISSC) n'utilisent pas d'entrées symboliques et utilisent un modèle de mise-à-0 plutôt que la mutation de données symbolique. Ces programmes implémentent des opérations modulaires, ce qui implique des boucles dont le nombre d'itérations dépend des entrées. Si les entrées ou la valeur d'une faute injectée sont symboliques, alors beaucoup de chemins peuvent être explorés, et potentiellement de longueurs élevées. Néanmoins, il est possible de borner la durée de l'exécution symbolique afin d'obtenir des chemins d'attaques.

La table 3.15 présente les résultats obtenus pour le programme d'exemple $rsa0$ avec le modèle d'injection de données en fonction de la stratégie d'exploration utilisée (colonne "Stratégie"). Par défaut, KLEE utilise la stratégie `nurs-cn` (Non-Uniform Random Search - Coverage-New), visant en priorité les chemins allant vers les instructions les moins couvertes. Les stratégies de recherche en profondeur (Deep First Search (DFS)) et recherche en largeur (Breadth First Search (BFS)) sont également présentées. Les colonnes suivantes indiquent le nombre d'attaque réussies trouvées pour chaque nombre de fautes. La colonne "EP" correspond au chemins explorés et la colonne "EI" au nombre d'instructions exécutées. La colonne "BCov" correspond à la couverture des blocs de base. La colonne "TDSE" indique combien de temps l'exécution symbolique a duré [26].

Table 3.15 – Comparaison de différentes stratégies d'exploration

Stratégie	1F	2F	3F	4F	EP	EI	BCov	TDSE
nurs-cn	0	0	0	0	-	-	-	-
dfs	54	71	100	75	419	3561393	96.8	33 :37
bfs	17	18	5	0	95	126315	33.62	57 :34 :00

On constate que la stratégie d'exploration a un fort impact sur les résultats qui sont obtenus. La stratégie par défaut ne parvient pas à trouver d'attaque en 30 minutes, mais les deux autres stratégies y parviennent, DFS trouvant le plus d'attaques dans ce cas. Pour `nurs-cn`, KLEE ne parvient pas à produire les ktests dans un temps imparti [27].

La table 3.16 présente les résultats obtenus pour $rsa0$ avec mutation de données symbolique en fonction du temps d'analyse, en utilisant la stratégie d'exploration DFS. La colonne "Délai" indique le temps accordé à l'exécution concolique pour l'analyse. Les autres colonnes ont la même signification que précédemment (table 3.15). Comme attendu, plus la durée limite est longue, plus l'analyse trouve de chemins d'attaque et plus la couverture est élevée. Cela étant, on peut observer que le passage de 1 minute à 5 minutes n'apporte que peu de nouveaux chemins d'attaques (4) tandis que le passage à 30 minutes en ajoute beaucoup.

Le choix de la stratégie d'exploration est un paramètre important pour Lazart, et dans le cas d'analyses qui ne terminent pas en un temps raisonnable, l'utilisation d'un timeout et d'une stratégie d'exploration adaptée permet de tout de même obtenir des résultats. La stratégie d'exploration peut aussi avoir un impact sur la consommation mémoire en fonction du nombre d'états symboliques explorés simultanément. Enfin, la stratégie d'exploration est aussi importante dans le cas où l'analyse termine, certaines stratégies pouvant rater des chemins, notamment lorsqu'elle incluent de l'aléatoire.

26. Le timeout est fixé à 30 minutes mais un temps supplémentaire est nécessaire à KLEE pour générer les ktests correspondant aux chemins en cours d'exploration.

27. Une limite de 1h pour l'arrêt de KLEE a été fixée, mais KLEE ne parvient pas à générer les cas de tests dans le temps imparti et ainsi aucune métrique d'exécution n'est disponible.

TABLE 3.16 – Attaques trouvées sur RSA en fonction du délai d'analyse spécifié

Délai	1F	2F	3F	4F	Chemins	Instrs.	BCov	TDSE
10s	10	5	0	0	35	306596	90.62	00 :11
1min	25	50	50	24	42	1962377	93.75	01 :00
5min	28	50	50	25	223	1965419	96.8	05 :01
30min	54	71	100	75	419	3561393	96.8	33 :37

3.6 Conclusion

La section 3.5 a présenté des solutions et méthodologies permettant de pallier les problématiques liées à l'utilisation de Lazart, notamment en ce qui concerne la sélection des points d'injection à évaluer. L'analyse statique permet de réduire l'espace de faute de Lazart de façon considérable sur certains exemples. La table 3.17 compare certaines des heuristiques présentées précédemment, leur nom étant indiqué dans la première colonne. La colonne "Type" détermine si l'heuristique est simple ou itérative. La colonne "Niveau" précise si l'heuristique est appliquée au niveau de l'analyse statique ou de Lazart. La colonne "Contexte de fautes" indique si l'heuristique est destinée à la faute unique ou aux fautes multiples et finalement la colonne "Complétude" précise si l'heuristique préserve la complétude.

TABLE 3.17 – Comparaison de différentes heuristiques

Heuristique	Type	Niveau	Contexte de fautes	Complétude
Limite d'occurence	simple	statique dynamique	simple multiple	Non
Selection bruteforce	iter	statique	simple	Oui
Strategy shrinking	iter	dynamique	simple	Non
Strategy widening	iter	dynamique	multiple	Non
Analyse manuelle	both	both	multiple	Non
Analyse partielle	both	dynamique	multiple	Non

Lazart propose donc des traitements des résultats visant à aider l'utilisateur dans la recherche de vulnérabilité et la protection du programme. Si la définition du périmètre initial d'une analyse reste délicate, l'approche itérative et l'utilisation d'analyse par sur-approximation permettent de guider l'utilisateur.

Le chapitre suivant revient sur ces problématiques du point de vue de l'implémentation. Elle précise certains choix qui ont été faits et qui résultent souvent d'un compromis entre accessibilité, passage à l'échelle et complétude.

Chapitre 4

Lazart - Implémentation et expérimentations

Ce chapitre s'intéresse à l'implémentation de l'outil Lazart et discute des choix et compromis qui ont été faits.

La section 4.1 décrit l'interaction entre Lazart et l'outil d'exécution concolique (KLEE). La section 4.2 présente Wolverine (phase de mutation), son architecture et son fonctionnement. La section 4.3 explique les choix d'implémentation concernant l'émulation des fautes. La section 4.4 détaille l'implémentation et la complexité des analyses d'attaques de Lazart. Finalement, la section 4.5 propose une synthèse des chapitres 3 et 4, ces deux chapitres sur l'outil Lazart et revient sur les contributions et les perspectives de l'outil.

Table des Matières

4.1 Interaction avec KLEE

KLEE est appelé après la phase de mutation avec pour entrées le mutant LLVM-IR et des arguments paramétrables depuis l'API Python (voir figure 4.1). Une fois que l'exécution symbolique est terminée (ou bien la fin d'un timeout spécifié par l'utilisateur), la phase

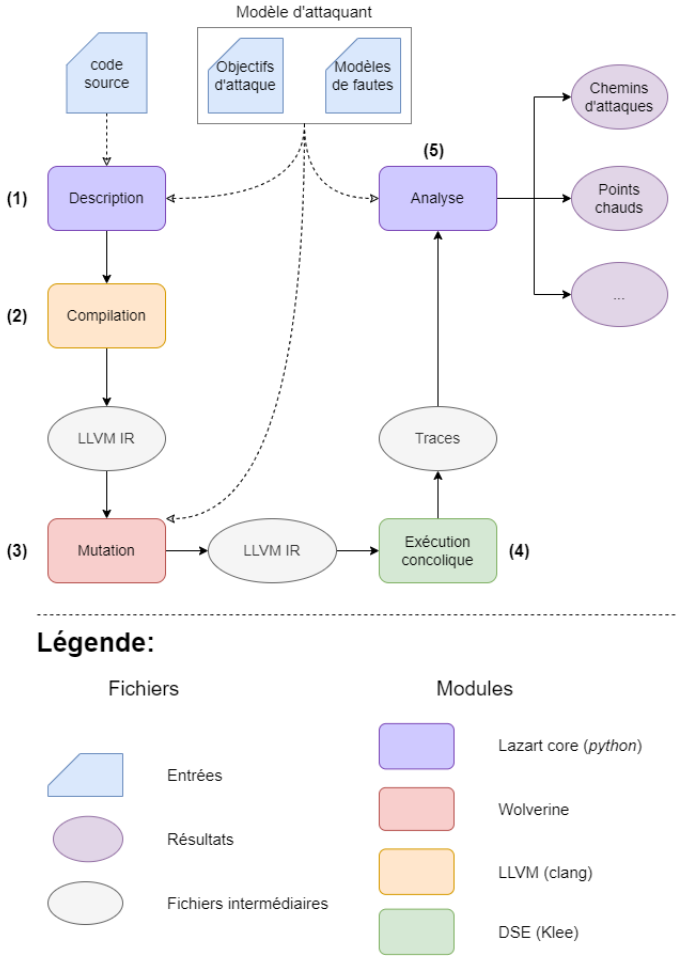

FIGURE 4.1 – Processus d'analyse avec Lazart (rappel)

de lecture des traces permet de récupérer les résultats de KLEE et les convertir dans la représentation de Lazart. L'usage de KLEE nécessite de prendre quelques précautions :

- Retarder au maximum l'introduction de variables symboliques et de disjonction de chemins dues aux fautes.
- Couper les traces qui ne nous intéressent pas au plus tôt.
- Choisir un paramétrage de KLEE adapté aux besoins de l'analyse et évitant d'introduire une concrétisation non souhaitée.

Ces aspects sont à prendre en compte par l'utilisateur lors de l'instrumentation du programme et du paramétrage de Lazart, comme cela a été abordé dans le chapitre précédent (section 3.5). Ce sont aussi des problématiques qui ont été considérées lors de l'implémentation de l'outil.

Les deux premiers points sont directement liés à la génération du mutant et à l'émulation des fautes, et seront abordés dans la section 4.3. Cette section se concentre sur la récupération des traces d'exécution à partir des cas de test et le paramétrage de KLEE. La section 4.1.1 présente la méthode de rejeu utilisée par Lazart pour récupérer les chemins de KLEE. La section 4.1.2 s'intéresse aux optimisations concernant l'exploration des chemins de KLEE et la lecture des ktests par Lazart. La section 4.1.3 décrit comment récupérer

des propriétés sur les traces, par exemple pour vérifier plusieurs objectifs d'attaque en une seule analyse.

4.1.1 Rejeu des traces

Pour récupérer les données d'une trace à partir d'un cas de test fourni par KLEE (fichier `.ktest`), chaque évènement d'une trace (entrée dans un bloc de base, déclenchement d'une faute, etc) est affiché sur la sortie standard avec un format spécifique. Lazart utilise la fonctionnalité *de rejeu* de KLEE (voir section 3.1.4) qui permet d'exécuter un ktest avec des entrées concrètes validant le prédicat de chemin de cette trace. La trace est ainsi reconstituée en analysant la sortie console de chaque rejeu de trace et transformée dans la représentation de Lazart.

```
1 [TRACE] bb2
2 [TRACE] bb3
3 [FAULT] [DL] [2] [12] [0]
4 [TRACE] bb5
5 [USER] tmp = 12
6 ...
```

Listing 4.1 – Exemple de rejeu d'un ktest de KLEE

Le listing 4.1 présente un exemple d'affichage console lors du rejeu d'une trace. Chaque type d'évènement suit une syntaxe précise. Les lignes préfixées par `[TRACE]` correspondent à l'entrée dans un bloc de base et la ligne préfixée par `[FAULT]` correspond à l'injection d'une faute dans laquelle le type de modèle est indiqué (ici mutation de données), suivi de l'identifiant du point d'injection (2) et finalement les valeurs spécifiques au modèle (ici la valeur attendue et la valeur fautée).

KLEE propose d'autres outils pour récupérer le chemin d'une trace. L'outil `ktest-tool` (voir section 3.1.4) permet de retrouver les valeurs de chaque variable symbolique d'un ktest donné et ainsi récupérer les fautes qui ont été injectées. Cette méthode est utilisée par Lazart lorsque le replay échoue [1]. Cette méthode permet de retrouver les fautes mais sans ordre et sans information sur le chemin [2] (fautes non-ordonnées et aucune information sur les blocs de base traversés par exemple).

4.1.2 Lecture des traces

L'étape de lecture des traces consiste donc à rejouer chaque ktest fourni par KLEE pour récupérer le chemin et les fautes injectées. La terminaison des traces est obtenue en cherchant les fichiers "*.err" associés à chaque ktest. KLEE génère un fichier par type de terminaison (*klee_assume* non validé, RTE par exemple). Si le chemin est complètement exploré, c'est-à-dire que le programme termine de façon nominale avec l'objectif d'attaque validé, alors aucun fichier d'erreur n'est associé au ktest.

Cette étape est très sensible à la performance des accès aux fichiers, d'une part pour le rejeu des traces et d'autre part pour la recherche de la terminaison. Plusieurs optimisations visant à limiter ces accès peuvent être mises en place. La récupération de tous les noms de fichiers permet de connaître la terminaison avant d'avoir rejoué la trace, le rejeu nécessitant un accès au fichier et un appel externe à ktest-tool pour chaque trace. Cela permet de savoir à l'avance si un ktest correspond à une trace d'attaque réussie et ainsi ne lire que les ktests nécessaires.

1. Par exemple en cas d'erreur de segmentation lors du rejeu ou bien lorsque le rejeu dépasse un délai fixé.
2. La récupération de ces informations depuis KLEE fait partie des perspectives d'amélioration de l'outil.

La lecture des traces ne lit par défaut que les traces validant l'objectif d'attaque. Ceci peut être paramétré par l'utilisateur s'il souhaite par exemple considérer les RTE comme des attaques, au prix d'une lecture des traces potentiellement plus longue. Néanmoins, KLEE par défaut ne génère qu'un ktest par point de code où une erreur de type RTE (ptr, div, free etc...), même si plusieurs erreurs se déclenchent dans différents chemins. Par exemple, il ne générera une erreur pour une division par zéro qu'une fois par point de code, même si plusieurs chemins d'attaques peuvent déclencher à ce point du programme. Cela permet à KLEE de ne pas effectuer les vérifications (et donc les appels au solveur) concernant les erreurs d'exécution pour chaque chemin. L'option *–emit-all-errors* doit être ajouté à KLEE pour générer un cas de test à chaque fois. Ainsi, si l'analyse nécessite de récupérer des traces correspondant à des cas d'erreur, il est nécessaire de spécifier l'option correspondante à KLEE et spécifier les ktests supplémentaires à traiter pour Lazart.

La table 4.1 propose une comparaison de plusieurs configurations de génération de ktests et de traces pour les collections de programmes `verify_pin` et `firmware_updater` présentés dans le chapitre précédent. La colonne "Version" indique les options de KLEE et le mode de génération de traces de Lazart. "eae" indique que tous les cas de tests d'erreur sont générés par KLEE, "standard" indique la lecture des traces standard (objectif d'attaque validé et pas de terminaison en erreur), "rte", la lecture des traces standard plus les erreurs d'exécutions et "full" indique que tous les ktests sont analysés. Les colonnes "Traces", "KTests", "EI" et "EP" correspondent respectivement au nombre de traces générées par Lazart, au nombre de Ktests, au nombre d'instructions exécutées et au nombre de chemins explorés. Les colonnes suivantes indiquent le temps d'analyse de KLEE ("TDSE"), de lecture des traces ("TTr") et la couverture des blocs de bases ("BCov"), des branches ("BRCov") et des instructions ("ICov").

Table 4.1 – Comparaison des méthodes de génération des traces

Programme	Version	Traces	KTests	EI	EP	TDSE	TTr	BCov	BRCov	ICov
vp5	standard	3291	3293	2172323	14213	15.56	7.63	98.7	96.43	92.85
	rte + eae	4875	14213	2172323	14213	23.65	23.146	98.7	96.43	92.85
	full + eae	14213	14213	2172323	14213	31.86	49.918	98.7	96.43	92.85
fu1	standard	9179	9183	36632126	137670	08 :05	31.976	97.06	91.67	84.31
	rte + eae	9727	137670	36632126	137670	20 :34	22 :20	97.06	91.67	84.31
	full + eae	54305	137670	36632126	137670	19 :14	02:14:65	97.06	91.67	84.31

Comme attendu, le mode standard est le plus performant, puisqu'il limite au maximum l'exploration. La couverture reste identique dans les exemples considéré quel que soit le mode utilisé, et il en va de même pour les attaques trouvées (qui ne sont pas indiquées ici). L'utilisation de l'option *eae* augmente considérablement le nombre de ktests produits par KLEE, ce qui influe sur le temps de l'exécution symbolique. Le temps de l'étape de récupération des traces ("TTr") devient proportionnellement plus important que celui de l'exécution symbolique lorsqu'on augmente le nombre de traces à lire. La couverture reste cependant assez proche de celle obtenue avec l'option *eae*, lorsqu'un ktest est généré pour chaque chemin d'erreur. Le cas *full + eae* implique la lecture de tous les chemins ne validant pas l'oracle, ce qui a peu d'intérêt en pratique mais est donné pour montrer le temps de lecture bien supérieur qu'on obtiendrait.

4.1.3 Objectifs d'attaque multiples et spécification par évènements

L'utilisation de *klee_assume* pour la vérification de l'objectif d'attaque permet de trier les traces suivant un prédicat binaire (satisfait / non-satisfait). Les *évènements utilisateur*

sont une autre méthode de vérification de propriété plus générales sur les traces. Cette méthode est notamment utile pour vérifier plusieurs objectifs d'attaque en une seule analyse.

Par exemple, les programmes *verify_pin* (voir section 3.5.1.1) peuvent être analysés avec différents objectifs d'attaques. On considérera ici les objectifs ϕ_{auth} (s'authentifier avec PIN faux) et ϕ_{ptc} (ne pas décrémenter le compteur d'essai avec un PIN faux) et leur conjonction $\phi_{auth \wedge ptc}$ et leur disjonction $\phi_{auth \vee ptc}$.

S'il est possible d'effectuer une analyse pour chacun des quatre objectifs d'attaque (et donc quatre exécutions concoliques), l'idée ici est d'ajouter des affichages pour le rejeu en fonction des propriétés *auth* (la fonction retourne vrai, l'utilisateur est authentifié) et *ptc* (le compteur d'essais n'a pas été décrémenté).

```
1  #define _LZ__EVENT(f_, ...) if(klee_is_replay()) { printf(("\n[USER_EVENT] " f_ "\n"), ##
       __VA_ARGS__); }
2
3  int main()
4  {
5      uint8_t user_pin[PIN_SIZE];
6      init(user_pin, card_pin); // make symbolic and not-equal
7
8      BOOL ret = verify_pin(user_pin);
9
10     if(ret == TRUE) {
11         _LZ__EVENT("AUTH");
12     }
13     if(try_counter >= TRY_COUNT) {
14         _LZ__EVENT("PTC");
15     }
16     return 0;
17 }
```

Listing 4.2 – Encodage des propriétés *auth* et *ptc* à l'aide d'évènements utilisateurs

Le listing 4.2 présente l'encodage des propriétés *auth* et *ptc* avec des *évènements utilisateur*, la non-égalité des entrées étant toujours vérifiée à l'aide d'un appel à `klee_assume` (dans la fonction `init`). Plutôt que couper les traces qui ne vérifient pas un objectif d'attaque binaire (avec `klee_assume`), des évènements utilisateurs (ici $AUTH$ et PTC sont ajoutés à la liste des transitions des traces en fonction des propriétés *auth* et *ptc*. La macro `_LZ__EVENT` englobe un appel à *printf* avec le bon format pour que l'évènement utilisateur soit récupéré lors du rejeu (l'implémentation de cette macro est indiqué ligne 1).

```
1  execute(a, no_analysis=True) # Compile, mutate, run DSE and parse traces
2
3  print(attacks_results(a, satisfies_fct=lambda trace: trace.satisfies() and trace.has_event
       ("AUTH"))
4  print(attacks_results(a, satisfies_fct=lambda trace: trace.satisfies() and trace.has_event
       ("PTC"))
5  print(attacks_results(a, satisfies_fct=lambda trace: trace.satisfies() and trace.has_event
       ("AUTH") and trace.has_event("PTC"))
6  print(attacks_results(a, satisfies_fct=lambda trace: trace.satisfies() and trace.has_event
       ("AUTH") or trace.has_event("PTC"))
```

Listing 4.3 – Encodage des propriétés de l'objectif d'attaque à l'aide d'évènements utilisateurs

Le listing 4.3 présente le code Python dans le script d'analyse permettant d'effectuer les analyses d'attaques sur les quatre objectifs d'attaque. Pour chaque analyse, les traces sont filtrées en fonction de la présence des évènements. L'analyse a est exécutée sans analyse d'attaque (*no_analysis=True*) puis une analyse d'attaque est exécutée pour chaque objectif d'attaque, en utilisant un filtrage des traces différent (*satisfies_fct*).

La table 4.2 présente les résultats d'une analyse du programme *vp3* pour les quatre objectifs d'attaque, en fonction de la méthode de vérification de l'oracle. Le programme est analysé avec le modèle d'inversion de test avec une limite de 5 fautes et des tableaux d'entrée symboliques. La colonne "Version" indique si l'analyse vérifie un objectif d'attaque "binaire" ou à l'aide d'évènements utilisateur ("uevent"). La colonne "objectif" indique l'objectif d'attaque étudié, "total" correspondant au total des analyses séparées pour la version "binaire". La colonne "chemins" correspond au nombre de chemins explorés et la colonne "instrs." correspond au nombre total d'instructions exécutées. Les colonnes "ICov" et "Bcov" indiquent respectivement la couverture des instructions et des blocs de base et "TDSE" et "TTot" indiquent respectivement la durée d'exécution concolique de l'analyse et la durée totale.

TABLE 4.2 – Résultats des l'analyses en fonction de la méthode de calcul de l'objectif d'attaque

Version	Objectif	Chemins	Instrs.	ICov	BCov	TDSE	TTot
séparée	ϕ_{auth}	1930	199238	98.01%	94.64%	1s 32	3s 305
	ϕ_{ptc}	1930	192652	97.99%	94.64%	1s 19	3s 124
	$\phi_{auth \wedge ptc}$	1930	205825	98.03%	94.64%	1s 47	3s 237
	$\phi_{auth \vee ptc}$	1930	205826	98.03%	94.64%	1s 19	3s 179
	total	7720	803541	-	-	5s 17	12s 75
uevent	tous	1930	206953	97.53%	92.19%	1s 37	5s 955

On constate que les *évènements utilisateurs* permettent d'améliorer grandement le temps d'analyse par rapport à l'utilisation d'analyse séparées avec des objectifs d'attaque binaires. Le surcoût de complexité de l'exécution concolique pour la vérification des différentes propriétés est largement compensé par l'usage d'une seule exécution concolique. On constate en effet que 1930 chemins sont explorés dans tous les cas, l'utilisation d'analyses séparées répète l'exploration de certains chemins. Dans cet exemple, les mêmes chemins d'attaque sont trouvés pour chaque objectif d'attaque binaire avec les deux méthodes. Néanmoins, il faut prendre en compte que la complexité supplémentaire de la vérification des différentes propriétés peut potentiellement faire rater des chemins d'attaque au moteur d'exécution concolique.

Les évènements utilisateurs permettent d'analyser plusieurs objectifs d'attaque en une seule exécution concolique, et plus généralement sont utilisés pour obtenir des propriétés plus complexes qu'un prédicat binaire (satisfait/non-satisfait) sur les traces.

4.2 Phase de mutation - Wolverine

L'étape de mutation (étape 3 de la figure 4.1) est réalisée avant la phase d'exécution concolique par le module *Wolverine* de Lazart. La représentation intermédiaire issue de la compilation est mutée de manière à y introduire la possibilité d'injecter des fautes. Cette phase de mutation est effectuée en fonction de la description du modèle d'attaquant et des opérations à effectuer dans le *fichier de mutation* au format YAML Ain't Markup Language (YAML). Le format de ce fichier est détaillé dans la section 4.2.1.

Wolverine utilise l'API C++ de LLVM permettant la manipulation de la représentation intermédiaire et s'appuie également sur une bibliothèque C liée au programme à analyser au moment de la compilation (étape 2). Il prend en entrée le programme sous la forme d'un bytecode LLVM (.bc) et le fichier de mutation YAML, et vise à transformer chaque

point d'injection du programme, par une transformation qui simule l'effet d'une faute (en fonction du modèle de faute spécifié) si le booléen symbolique d'injection est vrai.

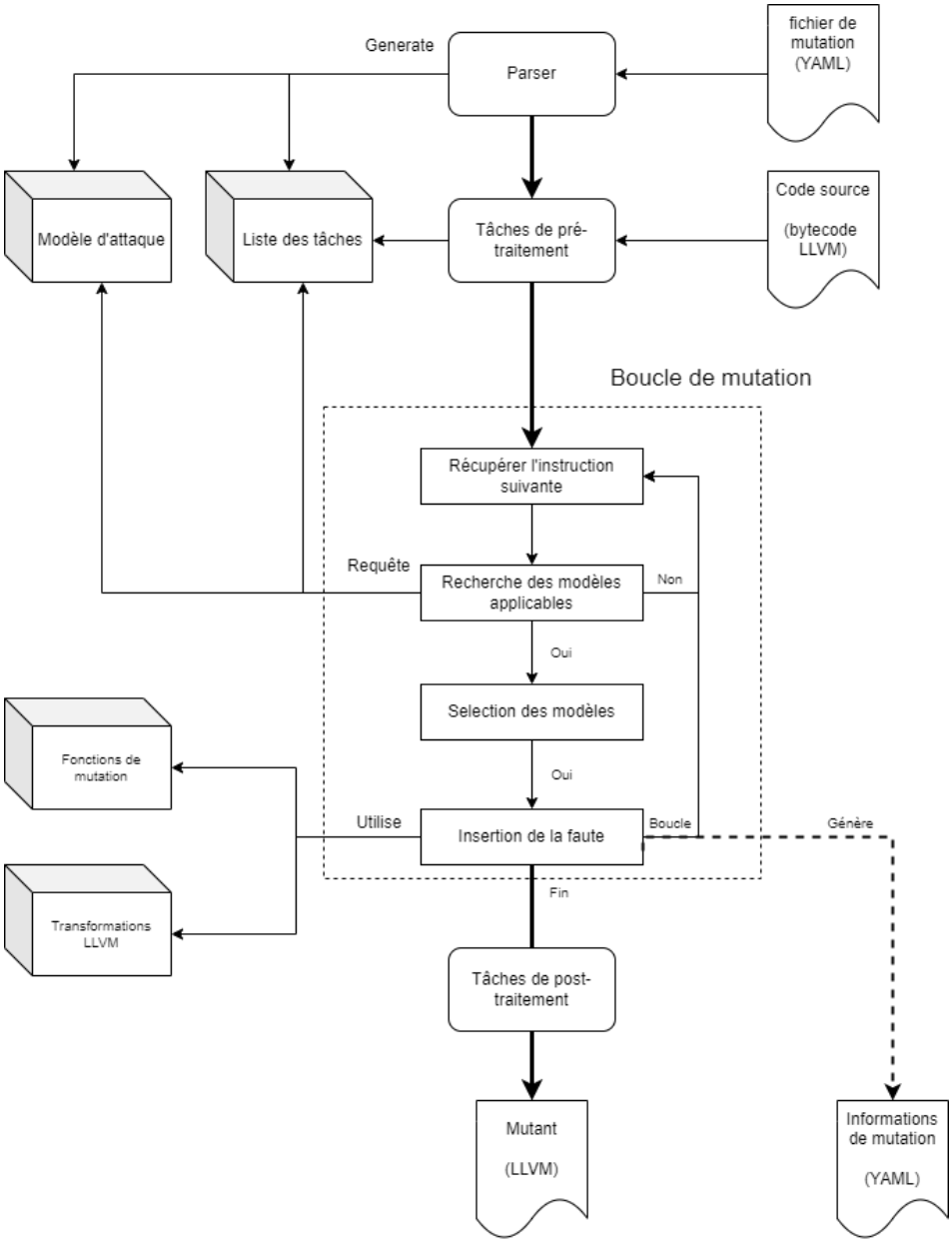

FIGURE 4.2 – Schéma général de *Wolverine*

La figure 4.2 présente le fonctionnement général d'une mutation. Après la génération du modèle d'attaquant à partir du fichier de mutation (étape 1), l'étape de *pré-traitement* (étape 2) consiste en plusieurs parcours du programme en fonction des opérations définies par l'utilisateur dans le fichier de mutation et l'instrumentation (détaillée en section 4.2.2).

La *boucle de mutation principale* parcourt chaque instruction du programme, fonction par fonction et bloc de base par bloc de base. Pour chaque instruction, le modèle d'attaquant et les informations des analyses sont interrogés afin d'obtenir la liste des modèles de faute

applicables sur cette instruction. Le programme est transformé en fonction des modèles de faute sélectionnés. Ces transformations sont détaillées dans la section 4.2.3. Enfin, Wolverine génère le fichier LLVM-IR mutant qui sera transmis à KLEE. Il génère aussi certaines informations utilisées ensuite par Lazart, telles que la liste des points d'injection.

4.2.1 Fichier de mutation

Le fichier de mutation contient les paramètres de la mutation sous la forme d'un fichier YAML qui est soit fourni par l'utilisateur, soit généré depuis Lazart Core lorsque ceux-ci sont décrits directement en Python. Le listing 4.4 montre un exemple de fichier de mutation.

```
1  ---
2  # Example of mutation file for Wolverine
3  tasks: # Preprocessing tasks
4    add_trace:
5      on: __mut__
6    rename_bb:
7      on: __all__
8    countermeasures:
9      - on: __mut__
10       type: load-duplication
11 ## General fault models, can be reused on specific location by ID.
12 fault-models:
13   - &data
14     type: data
15     all: 0 # All load are faulted to 0.
16 ## Fault spaces. Uses fault models on specific location of the program.
17 fault-space:
18   functions:
19     __all__:
20       models: [*data]
21     foo:
22       models:
23         - type: test-inversion
```

Listing 4.4 – Exemple de fichier de mutation

Le fichier de mutation contient trois parties : la section des tâches de pré-traitement (`tasks` ligne 3), la section des modèles de faute (`fault-models` ligne 12) et la section d'espace de faute (`fault-spaces` ligne 17). La section des tâches contient un ensemble d'actions effectuées en pré-traitement ou post-traitement. Dans l'exemple 4.4, les tâches `add-traces` (l'ajout des `printf` de trace d'exécution), `rename-bb` (renommage des blocs de base) et `countermeasures` (application d'une contre-mesure, ici la duplication des `load`) sont définies. Le champ `on` définit l'ensemble des fonctions sur lesquelles la tâche est effectuée, `__mut__` et `__all__` étant des mots clefs prédéfinis correspondants respectivement à l'ensemble des fonctions sur lesquelles au moins un modèle est défini et l'ensemble de toutes les fonctions.

La section de l'espace de faute suit une structure identique à celle du champ `attack_model` (modèle d'attaquant) de l'API Python (voir section 3.3.4.1) et contient une association entre des points du programme (ici des fonctions) et des modèles de faute. Ces modèles peuvent être définis en-place ou par référence YAML (comme ligne 20 dans l'exemple : `[*data]`).

4.2.2 Phase de pré-traitement

La phase de pré-traitement est effectuée au début de la passe de mutation. Celle-ci applique un ensemble d'opérations en fonction du fichier de mutation.

Les opérations utilitaires (renommage de blocs de base), de trace (ajout de `printf` pour tracer les blocs de base pour l'étape de rejeu) ou encore d'ajout de contre-mesures automatiques sont appliquées lors de cette phase avec une granularité de l'ordre de la fonction ou du bloc de base. Le pré-traitement effectue aussi les opérations déterminées par l'instrumentation du programme.

Cette phase pourrait aussi être utilisée pour appliquer d'autres analyses préliminaires (comme celles décrites dans la section 3.5). Dans la première version de l'outil [Potet 2014], le modèle d'inversion de test était précédé par une phase de "coloration de graphe" effectuée dans un module externe en Java permettant de réduire l'espace de faute. L'objectif d'attaque était alors défini en termes d'accessibilité à un bloc précis et cette analyse consiste à colorer chaque bloc pour savoir si une faute en inversion de test vers les branches then, else ou les deux peut mener au bloc ciblé. La combinaison de modèles de faute et la gestion de l'inter-procédural compliquent l'analyse dans les versions modernes de Lazart mais il s'agit aussi d'une analyse qui pourra être implémentée dans la phase de pré-traitement.

4.2.3 Boucle de mutation et points d'injection

La boucle de mutation effectue une passe sur toutes les instructions du programme dans les fonctions spécifiées par le modèle de faute comme présenté dans la figure 4.2. Elle maintient en parallèle l'état des indications de l'instrumentation en fonction des primitives rencontrées dans le programme, telles que l'activation ou la désactivation de modèles (voir section 3.3.4.3).

```
1  def mutation_loop(module: LLVMModule, istate: InstrumentationState, am: AttackModel):
2      for function in llvm_module:
3          istate.reset_function();
4          for bb in basic_block:
5              istate.reset_bb();
6              for instr in bb:
7                  consumed = istate.handle(instr)
8                  if consumed:
9                      continue
10
11                 models = am.get_appliable_models(instr, istate)
12                 if len(models) > 0:
13                     model = models[0]
14                     if len(models) > 1:
15                         warning("several models appliable, selecting: " + model)
16                     (instr, bb) = models.apply(instr)
```

Listing 4.5 – Boucle de mutation de Wolverine

Le listing 4.5 présente le pseudo-code de la boucle de mutation principale de Wolverine. Toutes les fonctions sont parcourues (à l'exception de primitives de LLVM et de Lazart) afin de pouvoir y détecter une activation de modèle par l'instrumentation [3]. Comme le montre l'avertissement à la ligne de 15, Lazart considère que plusieurs modèles applicables sur une même instruction correspond à une erreur de configuration (par exemple si deux modèles de mutation de donnée s'appliquent sur une même variable). La section 4.3.4 revient sur les cas où plusieurs fautes peuvent survenir sur un même point du programme.

Une difficulté de la combinaison de modèles de faute est qu'il ne faut pas qu'un modèle s'applique sur du code qui est relatif à l'analyse (code de simulation des fautes, `printf` pour le rejeu etc.) et non au programme analysé. L'utilisation d'une seule passe permet de ne jamais muter deux fois une instruction (dès lors que les transformations sont locales) mais Wolverine utilise par ailleurs un système d'étiquettes sur les instructions lui permettant

3. Le surcoût de performances est négligeable, toute l'étape de mutation étant elle-même négligeable.

d'identifier quelles instructions sont relatives à un point d'injection, à une contre-mesure ou à l'affichage du rejeu par exemple.

Wolverine fait le choix de limiter un maximum l'impact de la mutation sur le graphe de flot et le code original en englobant le code de déclenchement des points d'injection dans des *fonctions de mutation* (section 4.3). Ce choix permet aussi de simplifier le code muté s'il doit être utilisé par d'autres outils ou examiné manuellement, en permettant d'identifier simplement les parties du programme correspondant aux points d'injection. Cela étant, cette sur-couche d'un appel de fonction peut avoir des conséquences sur les performances de KLEE (discuté dans la section 4.3.3).

Le listing 4.6 contient un exemple de code LLVM-IR qui va permettre d'illustrer le fonctionnement de ces fonctions. Le listing 4.7 donne l'équivalent C de ce programme.

```
1 %1 = load i32* %val, align 4            ; lecture de val
2 %2 = call i32 @foo(i32 %1)              ; appel de foo
3 %3 = icmp ne i32 %2, 0                  ; comparaison
4 br i1 %4, label %else, label %then      ; branchement conditionnel
```

Listing 4.6 – IR LLVM source

```
1 if(foo(val)) {
2   // then:
3 } else {
4   // else:
5 }
6
```

Listing 4.7 – Équivalent en langage C

Dans cet exemple, pour un modèle de faute de mutation de donnée, la ligne 1 du programme est un point d'injection où la faute revient à charger une autre valeur que celle de `%val`. De la même manière, une faute dans le modèle d'inversion de test peut être représentée par la négation de la condition de l'instruction `br` ligne 4. Ces deux points d'injection sont nommés respectivement IP_{data} et IP_{ti}.

```
1  [...]
2  %6 = alloca i32
3  %7 = load i32* %val, align 4
4  %_LZ__w_call_ip_1 = call i32 @_LZ__mut_dl_i32(i32 %7, i8* getelementptr inbounds
       ([2 x i8], [2 x i8]* @_LZ__w_ip_1_id, i32 0, i32 0))
5  store i32 %_LZ__w_call_ip_1, i32* %6, align 4
6  %8 = load i32* %6, align 4
7  %9 = call i32 @foo(i32 %8)
8  %12 = icmp ne i32 %9, 0
9  %_LZ__w_call_ip_2 = call i32 @_LZ__mut_ti(i32 %12, i8* getelementptr inbounds ([2 x
       i8], [2 x i8]* @_LZ__w_ip_1_id, i32 0, i32 0), i8* getelementptr inbounds ([5
       x i8], [5 x i8]* @_LZ__w_ip_0_targetA, i32 0, i32 0), i8* getelementptr
      inbounds ([5 x i8], [5 x i8]* @_LZ__w_ip_0_targetB)
10 br i1 %_LZ__w_call_ip_2, label %bb26, label %bb25
11
```

Listing 4.8 – IR LLVM muté

Le listing 4.8 présente la transformation du programme 4.6 à l'aide de fonctions de mutation pour les points d'injection IP_{data} et IP_{ti}, respectivement représentés en rouge et en bleu-vert. Les fonctions de mutation (ici `_LZ__mut_dl_i32` et `_LZ__mut_ti`) prennent en argument la valeur originale de l'opérande cible, une chaîne de caractère correspondant à l'identifiant (unique) du point d'injection et d'éventuels arguments supplémentaires nécessaires au modèle. L'opérande de l'instruction ciblée est passée à la fonction de mutation dont le retour est utilisé comme nouvelle opérande.

Dans le cas de la mutation de donnée, une variable temporaire est créée puisque la valeur non fautée *%val* doit être chargée avant d'être passée à la fonction de mutation, et l'instruction `load` ligne 6 nécessite une adresse (`i32*`) comme opérande. Cette valeur temporaire est ainsi créée avec l'instruction *alloca* ligne 2, initialisée ligne 3, et finalement le retour de fonction `%_LZ__w_call_ip_1` est stocké dans la valeur temporaire avec une instruction `store` ligne 5, puis utilisé comme opérande pour l'instruction `load` ciblée par la faute ligne 6. Ainsi, il n'est pas nécessaire de modifier toutes les instructions du programme qui utilisent le temporaire `%8` puisque celui-ci reste inchangé.

Le modèle JMP n'est pas géré nativement par Wolverine et la transformation de leurs points d'injection est effectuée à la compilation via l'instrumentation. Le modèle est cependant reconnu durant la boucle de mutation afin de supporter les points d'injection générés par l'utilisateur dans les analyses (l'API Python ayant accès à ces points d'injections via le fichier `injection_points.yaml`).

4.3 Émulation des fautes

Cette section s'intéresse à l'implémentation de l'émulation des fautes, que ce soit dans les fonctions de mutation ou dans les transformations de points d'injection. L'objectif de ces transformations est de retarder l'introduction des variables symboliques ainsi que les disjonctions de chemins et de couper au plus tôt l'exploration des chemins.

La section 4.3.1 présente la structure des fonctions de mutation et propose une comparaison de plusieurs approches. La section 4.3.2 se concentre sur le modèle de mutation de données symboliques qui inclut une variable symbolique supplémentaire et discute des différentes versions de fonctions de mutation utilisées. La section 4.3.3 discute des problématiques induites par les fonctions de mutation par rapport à une transformation en-place et des différentes solutions. La section 4.3.4 parle de l'implémentation des points d'injection multiples générés par l'instrumentation.

4.3.1 Fonctionnement

Le programme présenté dans le listing 4.9 correspond au pseudo-code d'une émulation de faute sous la forme d'une fonction de mutation. Les entrées sont la valeur originale `orig`, potentiellement symbolique, le compteur de fautes `fcount` (symbolique). La limite de fautes `flimit` est globale. Les différents prédicats de chemins sont indiqués en marron et la mémoire symbolique (indiquée en rouge) considère `orig` comme concrète en entrée pour cet exemple.

```
1  def mutation_fct(orig, fcount): PC_0 ≡ PC_fcount    σ = { fcount → fcount_0 }
2      if fcount >= flimit: # No sym variable. PC_1 ≡ PC_0 ∧ fcount ≥ flimit
3          return orig
4      PC_2 ≡ PC_0 ∧ fcount < flimit
5
6      inject = sym_bool() # New sym variable.   σ = { fcount → fcount_0 ∧ inject = inject_0 }
7      if inject: PC_3 ≡ PC_2 ∧ inject = true
8          fcount++ # Update.  σ = { fcount → fcount_0 + 1 ∧ inject = inject_0 }
9          if klee_is_replay(): # Do not create new paths.
10             print(fault_str_with_params) # Trace fault trigger for replay.
11         return faulted_value
12     PC_4 ≡ PC_2 ∧ inject = false
13
14     return orig
```

```
15
```

Listing 4.9 – Pseudo-code de l'émulation des fautes

Cette implémentation vise à retarder au maximum la création de la variable symbolique booléenne `inject`. C'est pourquoi un premier test sort immédiatement de la fonction si la limite de fautes est atteinte. C'est aussi la raison pour laquelle *klee_assume* n'est pas utilisée ici, son appel n'étant pas possible avant la création d'`inject`. La fonction *klee_is_replay* est une primitive de KLEE permettant de ne pas exécuter le code lié au rejeu des traces durant l'exécution concolique, qui évite les risques de concrétisation [4].

4.3.2 Mutation de données

Cette section vise à présenter les différentes approches d'émulation de fautes en données. Le listing 4.10 présente le pseudo-code du retour de la valeur dans le cadre d'une faute des cinq approches d'implémentation de la fonction de mutation pour l'injection sur les données :

- `sym`, la variable symbolique injectée n'est pas contrainte et est retournée directement.
- `sym_pred`, la fonction retourne un booléen déterminant si la valeur est valide, et qui est vérifié par un appel à `klee_assume`.
- `sym_fun`, la valeur est transformée par la fonction qui retourne une nouvelle valeur contrainte.
- `fun`, consiste à faire la même opération sans passer par une valeur symbolique.
- `fixed` : est utilisée pour l'injection d'une valeur constante.

```
1  # Symbolic raw (sym)
2  return sym_int()
3
4  # Symbolic constrained (sym_pred)
5  value = sym_int()
6  klee_assume(pred(value, original))
7  return value;
8
9  # Symbolic constrained (sym_fun)
10 value = sym_int()
11 return f(value, original)
12
13 # Not symbolic, apply function (fun)
14 return f(original)
15
16 # Fixed value (fix)
17 return N
18
```

Listing 4.10 – Différentes versions de contraintes la mutation de donnée

4.3.3 Performance et inlining

Cette section compare les approches par fonction de mutation et par transformation en-place (c'est-à-dire sans appel de fonction, en transformant le graphe de flot local). Les fonctions de mutation ont l'avantage d'être plus simples pour un utilisateur qui souhaiterait créer de nouveaux modèles et parce qu'elles simplifient le code à analyser en séparant clairement le graphe de flot original de la partie mutation. Cependant, ces fonctions induisent

4. La documentation de KLEE indique que la fonction `printf` est gérée de façon spéciale, de manière à ne pas introduire de concrétisation. Mais nos expérimentations ont mit en évidence certains cas où la présence d'un appel à `printf` dépendant de variables symboliques fait manquer des chemins à KLEE.

une surcharge pour le moteur d'exécution symbolique, qui doit effectuer le changement de contexte dû à un appel de fonction et transmettre des paramètres statiques en argument (comme la chaîne de caractères correspondant à l'identifiant de point d'injection, ou l'entier `fcount` correspondant au nombre de fautes injectées par exemple), et les auteurs de REFINE [Georgakoudis 2017] indiquent que l'implémentation en-place permet d'obtenir des meilleures performances. Néanmoins, les expérimentations menées pour Lazart monstrent que ce surcoût n'est pas si important dans le cas d'une exécution concolique avec KLEE, l'explosion des chemins due aux fautes ayant un impact bien plus important que cette surcharge sur certains exemples.

Wolverine propose une option d'inlining automatique des fonctions de mutation via une passe LLVM qui est par défaut appliquée sur chaque appel de fonction de mutation (généré ou défini par l'utilisateur). Cette approche n'effectue pas un inlining en-place, mais place le code de la fonction de mutation dans la fonction courante (qui est donc réutilisé par d'autres injections dans la même fonction). Cette approche souffre de la surcharge de la transmission des arguments tout comme la version avec fonction de mutation, mais ne contient pas d'appel de fonction explicitement (instruction `call`). Une version en-place de la mutation de donnée symbolique a aussi été implémentée, qui elle consiste à avoir un code de mutation séparé pour chaque point d'injection.

La table 4.3 compare l'approche des fonctions de mutation sans optimisation (`mfct`), l'inlining automatique (`mfct+inl`) et la mutation en-place (`inplace`) sur un ensemble de programmes. Les exemples utilisés sont indiqués dans la colonne "Prgm.", *rsa*0 correspondant à l'implémentation de RSA en mise-à-0 et *loader* au bootloader (voir chapitre précédent). *loader* n'est évalué qu'en faute unique. L'approche utilisée est indiquée dans la colonne "Stratégie", la version en-place n'étant pas implémentée pour l'injection d'une valeur fixe, la version en-place pour *rsa*0 n'est pas présente. Les colonnes suivantes indiquent le nombre d'attaques trouvées en fonction du nombre de fautes. On constate que chaque version donne bien des résultats identiques.

Les colonnes "Chemins" et "Instrs." indiquent respectivement le nombre de chemins explorés et le nombre d'instruction exécutées. On constate que l'inlining automatique est moins performant. La colonne "TDSE" qui indique le facteur de temps d'exécution moyen indexé sur l'approche la plus faible. Les variations entre les versions `mfct` et `mfct+inl` sont trop faibles pour être mesurées dans le cas de *rsa*0 mais visibles sur *loader*. Le fait que l'inlining automatique soit plus coûteux tend à montrer que les changements de contexte lié à l'appel de fonction sont négligeables pour KLEE. La différence entre les deux étant expliquée par le fait que l'inlining automatique de LLVM complexifie légèrement le graphe de flot. En revanche, l'exemple *loader* indique des facteurs de temps plus significatifs. La version `inplace` étant la plus performante puisque les passages d'arguments ne sont pas présents.

Les colonnes "ICov" et "BCov" donnent des indications concernant la couverture des instructions et des blocs de base. La différence entre les approches est significative mais principalement parce que ces métriques ne mesurent pas la même chose suivant les versions. Dans le cadre d'une approche `mfct`, chaque point d'injection partage son code avec tous les points d'injection utilisant la même fonction de mutation. La couverture donnée par KLEE reflète donc davantage celle obtenue sur le programme analysé. Dans le cadre de l'inlining automatique `mfct+inl`, la couverture du code d'émulation est liée à la fois à la fonction de mutation utilisée, mais aussi à la fonction contenant le point d'injection. Pour la mutation en-place, le code d'émulation est séparé pour chaque point d'injection et la couverture retournée par KLEE est donc plus faible. On constate d'ailleurs que le nombre de chemins explorés est le même pour toutes les approches, seul le nombre d'instructions exécutées diffère.

Prgm.	Stratégie	1F	2F	3F	4F	Ktests	Chemins	Instrs.	ICov	BCov	TDSE
rsa0	mfct	10	33	65	92	202	913	52965525	100	100	**1**
	mfct+inl	10	33	65	92	202	913	52969338	98.55	93.02	**1**
loader	mfct	12	-	-	-	14	13592	218071560	59.7	45.26	1.15
	mfct+inl	12	-	-	-	14	13592	237171222	69.65	58.39	1.3
	inplace	12	-	-	-	14	13592	183262082	64.06	58.39	**1**

TABLE 4.3 – Expérimentations sur la mutation de données

La table 4.4 présente une comparaison des approches d'émulation des fautes. La colonne "Performance" classe les approches en fonction de la surcharge du temps d'exécution. La seconde colonne s'intéresse à l'impact de la méthode sur les métriques de couverture de KLEE. Les colonnes suivantes indiquent respectivement la simplicité avec laquelle un modèle peut être ajouté ou étendu (l'implémentation au niveau source étant considérée plus simple que la manipulation de la représentation LLVM), et si le code d'émulation est explicitement séparé du reste du programme. La dernière colonne indique l'état de l'implémentation de la méthode dans Lazart. La version en-place n'est disponible que pour la mutation de donnée non-contrainte [5]. Une version en-place pour les autres modèles de Lazart est une piste d'amélioration de l'outil.

Approche	Performance	Représentativité de la couverture	Extensibilité des modèles	Séparation de l'émulation	Implémenté
mfct	moyenne	par modèle	simple	forte	oui (défaut)
mfct+inl	pire	par modèle et par fonction	simple	moyenne	oui
inplace	meilleure	par IP	complexe	aucune	partielle (dl-sym)

TABLE 4.4 – Comparaison des approches de mutation de point d'injection

4.3.4 Points d'injection multiples

Un point d'injection multiple consiste à faire une disjonction entre le cas nominal et plusieurs cas fautés, comme indiqué dans le listing 4.11. Les points d'injection multiples sont plus performants qu'une succession de points d'injection simples, puisqu'ils n'introduisent qu'une seule fois le chemin correspondant à l'absence de faute. Wolverine ne génère pas de point d'injection multiple pour l'inversion de test et la mutation de données puisque ceux-ci n'entrent pas en collision lors de leur application (voir section 4.2.3). Il serait cependant possible de créer des points d'injection multiples si d'autres modèles étaient ajoutés. Le modèle JMP (uniquement défini par instrumentation) est un exemple de point d'injection multiple dans Lazart, lorsque plusieurs étiquettes de destination sont spécifiées.

```
1 int select = sym_int()
2 if select == 0:
3   normal_behavior()
4 else if select == 1:
5   model_1_behavior()
6 else if select == 2:
7   model_2_behavior()
8 ...
```

Listing 4.11 – Pseudo-code d'un point d'injection multiple

5. Hors Wolverine, le modèle JMP est un exemple d'émulation en-place.

La table 4.5 présente les résultats normalisés (sur la valeur la plus faible indiquée en gras) sur un ensemble de programmes `verify_pin` ($vp0$ à $vp6$) avec deux modèles de fautes JMP : le saut vers deux étiquettes possibles et le saut vers trois étiquettes possibles. La colonne "Mode" indique si le point d'injection est émulé par une succession de points d'injection simples ("unique") ou bien par un point d'injection multiple ("multiple"). On peut constater que la version avec IPs multiples est plus performante en explorant moins de chemins, sans différence sur les résultats de l'analyse d'attaque obtenus.

Mode	TDSE	Traces	Chemins explorés (EP)	Attaques
unique	1.43	1.25	1.80	**1**
multiple	**1**	**1**	**1**	**1**

TABLE 4.5 – Métriques de performance entre IP simple et multiple

4.4 Implémentation des traitements

Cette section se focalise sur l'implémentation et la complexité des traitements proposés par Lazart (présentés en section 3.4). La section 4.4.1 se concentre sur les analyses d'attaques et de points chauds, et la section 4.4.2 sur l'analyse de redondance-équivalence.

4.4.1 Attaques et points chauds

L'analyse d'attaque et l'analyse de points chauds sont des analyses linéaires puisqu'elles ne s'appliquent qu'une fois sur chaque trace. Leur complexité au pire cas est donc $O(n*m)$, avec n le nombre de traces à parcourir et m la taille moyenne des traces (en nombre d'évènements à analyser). L'analyse de points chauds est cependant plus coûteuse en temps parce qu'elle maintient plusieurs valeurs pour chaque point d'injection.

```
def attack_analysis(a: Analysis, s_fct):
    attacks = []
    for order in range(a.max_order()):
        for trace in traces_list(a, order,
      s_fct):
            attacks.append(trace)
    return attacks
```

Listing 4.12 – Pseudo-code de l'analyse d'attaque

```
def hotspots_analysis(a: Analysis, s_fct):
    ips_data = dict<ip_id: str, data:
     IPData>

    for order in range(a.max_order()):
        for trace in traces_list(a, order,
      s_fct):
            data = compute_local_hs(trace)
      # metrics on ip in the traces
            ips_data.update(data) # update
      metrics

    return ips_data
```

Listing 4.13 – Pseudo-code de l'analyse de points chauds

FIGURE 4.3 – Pseudo-code pour les analyses d'attaques et de points chauds

Le listing 4.12 (de la figure 4.3) correspond au pseudo-code de l'analyse d'attaque, qui peut se réduire à un filtrage des traces suivant le prédicat `s_fct` fourni par l'utilisateur (par défaut les traces satisfaisant l'objectif d'attaque, sans terminaison en erreur).

Le listing 4.13 présente le pseudo-code du calcul des points chauds qui correspond à une mise-à-jour des valeurs (nombre total de déclenchements, nombre maximum par trace etc...) pour chaque trace, l'ensemble de traces étant aussi paramétré par `s_fct`.

Les deux analyses pourraient en théorie être effectuées toutes deux en une seule passe mais la séparation permet d'appliquer l'analyse de points chauds sur d'autres ensembles de traces (comme les attaques minimales par exemple).

4.4.2 Analyse de redondance-équivalence

L'analyse de redondance-équivalence a une complexité plus élevée que les analyses d'attaques et de points chauds. Le temps d'analyse peut avoir un impact non négligeable sur les performances pour certains exemples, comme cela a été illustré dans la section 3.5.1.

L'analyse d'équivalence nécessite de comparer pour chaque ordre o (chaque nombre de fautes) toutes les traces du groupe de traces entre elles, ce qui implique une complexité au pire cas pour un ordre donné en $O(n * n * m)$, avec n le nombre de traces pour un ordre donné et m la moyenne de la taille des traces dans cet ensemble. Plus il existe de classes d'équivalence, plus l'analyse est rapide (pour chaque groupe d'équivalence, une seule trace est traitée par l'analyse de redondance). La complexité à l'ordre o serait plutôt $O(n*e*m)$ avec e correspondant au nombre de classes d'équivalence pour o.

L'analyse de redondance nécessite de comparer chaque trace d'ordre o à toutes les traces de chaque ordre supérieur. La complexité correspond donc à $O(n * log(n))$. Si on ne s'intéresse qu'à trouver les attaques minimales (voir section 3.4.4), il n'est pas nécessaire de calculer toutes les relations de redondance entre les traces, et les traces déjà déterminées comme redondantes n'ont pas à être comparées aux autres traces. L'option `lazy` permet de limiter l'analyse de redondance à la recherche de traces minimales.

Le listing 4.14 présente le pseudo-code de l'analyse de redondance-équivalence. Le graphe de redondance (`graph`) représente les relations de redondance entre les traces ainsi que les classes d'équivalence. Les classes d'équivalence sont d'abord calculées pour chaque nombre de fautes et l'analyse de redondance peut ainsi s'appliquer au niveau de ces classes plutôt que sur chaque trace. Les métriques, comme le nombre total d'attaques minimales, de classes d'équivalence minimales ne sont pas présentées ici mais sont calculées et stockées durant l'analyse (afin de ne pas avoir à retraverser le graphe de redondance pour récupérer ces valeurs).

```
1     def equivalence_then_redundancy(a: Analysis, do_eq: bool, do_red: bool, eq_s_fct,
       red_s_fct, lazy: bool):
2     graph = TraceGraph()
3
4     if do_eq:
5         for order in range(a.max_order()):
6             trs = traces_list(a, order, eq_s_fct):
7             for i in range(trs.size() - 1):
8                 tr1 = trs[i]
9                 for j in range(i, trs.size()):
10                    tr2 = trs[j]
11                    if equivalence_rule(tr1, tr2):
12                        graph.set_equivalent(tr1, tr2)
13
14    for order in range(1, a.max_order() - 1):
15        trs = traces_list(a, order, red_s_fct)
16        for tr1 in trs:
17            if graph.redundant(tr1) and lazy: # lazy mode
18                continue
19
20            for target_order in range(order + 1, a.max_order()):
```

```
21                 for tr2 in traces_list(a, order, red_s_fct):
22                     if redundancy_rule(tr1, tr2):
23                         graph.set_redundant(tr1, tr2)
24
25     return graph
26
```

Listing 4.14 – Pseudo-code de l'analyse de points chauds

Cette approche, consistant à réduire la complexité de la redondance en calculant préalablement les classes d'équivalence, est efficace si peu de classes d'équivalence sont créées. Il est possible à l'inverse de commencer par calculer la redondance des traces puis ne calculer l'équivalence que sur les attaques minimales. Si le nombre de classes d'équivalence est élevé pour les attaques maximales et centrales, alors cela peut accélérer le calcul de l'équivalence.

L'implémentation de Lazart permet de paramétrer l'ordre dans lequel l'équivalence et la redondance sont effectuées ainsi que les définition d'équivalence et de redondance utilisées. L'utilisateur peut spécifier les prédicats déterminant l'équivalence et la redondance de deux traces, les plages d'ordres sur lesquelles les analyses s'appliquent, et les plages d'ordres de comparaison (voir la liste des options dans l'annexe A.1).

4.5 Conclusion et perspectives

4.5.1 Contributions sur Lazart

La reprise et le développement de l'outil Lazart ont été réalisés tout au long de cette thèse. L'outil a été réécrit pratiquement en entier afin de répondre aux besoins des différents utilisateurs académiques et industriels. L'API Python a été développée pour simplifier la création d'analyses complexes et Wolverine (phase de mutation) a été entièrement réécrit de manière à supporter la mutation inter-procédurale et la combinaison de modèles de fautes, ainsi que faciliter la récupération d'informations sur la mutation (comme les détails sur les points d'injection dans le mutant final). Le projet a part ailleurs été migré vers une version plus récente de LLVM (version 9) et de KLEE (version 2).

L'analyse de redondance a été revue au cours de la thèse puisque dans sa version originale [Potet 2014] celle-ci n'utilisait qu'une version partielle de la définition sous-mot. Les concepts d'équivalence, de faute-équivalence et de redondance ont été implémentés, ainsi que l'analyse de points chauds et les analyses de contre-mesures présentées dans les chapitres suivants.

L'analyse inter-procédurale et la combinaison de modèles de fautes ont été ajoutées, ainsi que les fonctionnalités de mutation par instrumentation du programme. Le projet compte environ 20k lignes de codes réparties entre les différents modules (API Python, Wolverine et outils utilitaires). De plus, la collection FISSC a été mise à jour au fur et à mesure du développement de Lazart.

4.5.2 Pistes d'amélioration

Outres des améliorations techniques, certaines pistes sont envisagées pour le futur de Lazart. Les modèles de fautes supportés par l'outil pourraient être étendus. En effet, les modèles JMP, TI et DL permettent de simuler la plupart des modèles haut-niveaux. Par exemple, il est possible de représenter *test-falltrought* (exécuter les deux branches d'un test) à l'aide du modèle JMP. Le modèle de mutation de pointeurs pourrait aussi être simulé [6] en autorisant l'injection d'une valeur contenue dans une variable locale de la même fonction ou

6. Si on exclut la solution qui serait de rendre symbolique les adresses, non supportée par KLEE.

bien encore en actualisant la mauvaise variable (ou la mauvaise adresse) lors de l'injection. Des travaux sont en cours dans cette direction.

Les fautes permanentes ne sont pas traitées par Lazart, et pourraient être représentées avec l'utilisation de fonctions de mutation spécifiques par point d'injection (ou une mutation en-place). Le point d'injection serait ainsi activé définitivement une fois déclenché et la valeur retourné serait toujours retournée. Le support des fautes permanentes, en mémoire, ainsi que des limites de fautes spécifiques à un modèle de faute ou un point d'injection particulier (voir section 3.5.2.3) sont aussi des perspectives d'extension pour l'outil.

L'implémentation d'une version inter-procédurale de la coloration de graphe [Potet 2014] pour le modèle d'inversion de test (ou plus généralement des modèles orientés flot de contrôle) afin d'écarter les fautes sur des points du programme n'ayant pas d'accessibilité avec l'objectif d'attaque pourrait améliorer les performances de ces modèles. Cela s'avère cependant plus complexe à mettre en place dans le cadre de la combinaison de modèles si les fautes peuvent ajouter des arrêtes ou des blocs. Plus généralement, la mises en place d'analyses préliminaires pour Lazart (voir sections 3.5 et 4.2.2) fait partie des projets futurs pour l'outil.

Plutôt que l'approche de type *swifi* de Lazart qui consiste à émuler les fautes en mutant le programme avant de le transmettre à l'outil d'exécution symbolique, il est possible de modéliser les fautes dans le moteur d'exécution symbolique. En fonction d'un modèle d'attaquant, le moteur d'exécution peut diviser l'exécution symbolique en un comportement fauté et non fauté lorsqu'il rencontre une instruction sur laquelle une faute peut être injectée (voir l'exploration symbolique fautée décrite dans la section 3.1.5). Dans le cas de LLVM-IR, cela rendrait notamment la simulation des fautes qui impliquent une modification du CFG (ajout de blocs de base et ou d'arêtes) plus simple à simuler puisque cela reviendrait à modifier l'instruction suivante à lire pour le moteur d'exécution concolique, sans nécessiter une reconstruction d'un CFG incluant tous les sauts possibles comme c'est le cas actuellement. Les modèles de faute touchant spécifiquement des parties de la modélisation du système seraient aussi plus faciles à simuler. Par exemple, un modèle d'injection de données dans la mémoire [7] reviendrait à rendre symbolique une portion de mémoire touchée. Bien entendu, la problématique d'explosion combinatoire en fautes multiples reste présente.

L'implémentation dans le moteur d'exécution symbolique est probablement plus efficace en termes de performance en raison de l'économie des lectures / écritures de fichiers (voir section 4.1) et des changements de contexte dûs à l'appel externe de l'outil concolique. Les analyses sur l'accessibilité ou la dépendance des données pourraient être faites au fur et à mesure de l'exécution symbolique. En effet, si une faute a déjà été injectée et qu'une limite de fautes est fixée, alors certains chemins fautés peuvent devenir impossibles en raison de cette limite (pareillement pour des analyses de teinte ou de dépendance). De plus, l'accès à la représentation du système (représentation de la mémoire, système de fichiers, temporaires etc...) de l'outil d'exécution permettrait d'implémenter plus simplement des extensions de cette représentation pour certains modèles spécifiques (par exemple un registre multiplieur spécifique à une architecture [Laurent 2019]). Des outils de ce type existent au niveau binaire mais aucun au niveau LLVM-IR. Une version étendue de KLEE supportant l'injection de fautes est donc une piste de recherche intéressante.

7. Dans Lazart ce type de faute peut être simulée avec une succession d'injections de mutation de donnée (fautes multiples en lecture). Une autre solution serait de développer un modèle d'injection de données sur l'instruction `store`.

Une autre piste d'amélioration à explorer consiste à ne pas encoder l'objectif d'attaque dans le programme analysé à l'aide de `klee_assume`, mais de le vérifier avec le prédicat de chemin. Cela permettrait d'éviter la problématique d'une exécution concolique non disjointe (voir section 3.4.3.1) à cause de cette vérification, les contraintes sur les entrées et l'objectif d'attaque étant simplement des contraintes supplémentaires au prédicat de chemin. Ne pas explorer les chemins dans le code d'encodage de l'objectif d'attaque permettrait d'améliorer le passage à l'échelle pour certaines analyses.

Chapitre 5

Placement de contre-mesures logicielles

Ce chapitre se concentre sur les contre-mesures logicielles et présente une méthodologie de placement de contre-mesures en fautes multiples combinant une analyse en isolation des schémas de protection et différents algorithmes de placement automatique.

La section 5.1 présente les problématiques de l'analyse de contre-mesures et comment les méthodologies proposées dans ce chapitre peuvent être utilisées pour y répondre. Cette section s'intéresse aussi à quelques contre-mesures logicielles et leurs caractéristiques, ainsi que les approches existantes dans le domaine de l'évaluation et la comparaison de programmes protégés. La section 5.2 traite des différentes contre-mesures automatiques proposées dans Lazart et introduit certaines propriétés nécessaires aux algorithmes de placement. La section 5.3 présente les algorithmes de placement automatique de contre-mesures et discute des garanties qu'ils apportent en fonction des contre-mesures considérées. La section 5.4 discute des expérimentations qui ont été réalisées pour ces différents algorithmes. La section 5.5 s'intéresse à la notion de "protégeabilité" des modèles de faute et propose une classification correspondante. Finalement, la section 5.6 discute des perspectives de ces travaux.

Table des Matières

5.1 Chaîne de développement logiciel et analyse de contre-mesures

Les chapitres précédents ont introduit les notions de contre-mesures et d'analyse de contre-mesures (section 2.3). La suite de ce manuscrit s'intéresse plus particulièrement à la protection de programmes au niveau logiciel.

Définition 5.1. *Une contre-mesure est une transformation d'un système (programme, algorithme, composant) qui préserve son comportement observable en l'absence d'attaque et vise à augmenter la sécurité en présence d'attaques.*

L'*analyse de contre-mesure* a déjà été abordée dans les chapitres précédents, et plusieurs problématiques ont été énoncées :

— *le placement :* comment sélectionner les contre-mesures et leurs positions dans un programme.
— *la comparaison :* comment comparer et évaluer des contre-mesures et des programmes protégés.
— *l'optimisation :* comment déterminer si les protections ajoutées sont effectivement utiles ou si certaines peuvent être retirées.

Ces différentes problématiques sont liées les unes aux autres. Entre autres :

— la capacité de comparer les contre-mesures, pour un modèle d'attaquant donné, permet d'aider au placement des contre-mesures,
— l'optimisation de programme peut être utilisée pour placer des contre-mesures en les appliquant systématiquement sur le programme et en calculant a posteriori une version réduite de celui-ci.

La section 5.1.1 s'intéresse à quelques contre-mesures logicielles de la littérature et discute de leurs caractéristiques. La section 5.1.2 présente une revue des techniques d'analyse de contre-mesures proposées dans la littérature. Finalement, la section 5.1.3 propose une synthèse de cette section.

5.1.1 Contre-mesures

La section 2.3 a présenté les caractéristiques des contre-mesures en fonction du niveau de représentation considéré. Cette section se concentre sur les contre-mesures logicielles (ce qui inclut les transformations au niveau assembleur/binaire) et vise à présenter différentes caractéristiques, qui permettront notamment de classifier les contre-mesures pour s'intéresser aux garanties apportées par les algorithmes de placement. Les contre-mesures visant les attaques par canaux-auxiliaires ou visant spécifiquement la protection d'implémentations cryptographiques ne sont pas considérées ici.

La table 5.1 présente différentes contre-mesures logicielles proposées dans la littérature. La colonne "Nom" indique la contre-mesure en question et "Date" l'année de la publication. Les contre-mesures préfixée par "LZ" correspondent aux contre-mesures automatiques implémentées par Lazart : multiplication de tests ("LZ TM"), multiplication de loads ("LZ LM") et une implémentation de SecSwift [de Ferrière 2019] ("LZ SSCF").

Les colonnes suivantes correspondent aux différentes classes de caractéristiques qui sont présentées dans cette section : "Modèle" (caractéristiques relatives au modèle de faute et au niveau de représentation), "Méthode" (caractéristiques liées à l'implémentation et le fonctionnement de la contre-mesure) et "Application" (caractéristiques concernant l'application de la contre-mesure sur un programme).

Deux grandes *familles* de contre-mesures se dégagent au niveau logiciel (colonne "Famille"), en fonction de si celles-ci visent la protection du flot de contrôle (CFI) ou du flot de

Général			Modèle			Méthode		Application			
Nom	Famille	Date	Dom.	Modèle	MF	Etat	Tech.	Niveau	Granu.	Auto.	Syst.
Booléen durcis	Data	-	FA	Data	1	non	L	-	V	oui	F
CFCSS [Oh 2002]	Both	2002	FT	bit-flip	1	oui	D / I	Compil.	B	oui	P
[Nicolescu 2004]	Both	2003	FT	bit-flip	1	oui	D / I	Compil.	I/B	oui	P
Swift [Reis 2005]	Both	2005	FT	CF	1	oui	D / I	LLVM	G	oui	P
Lalande	CFI	2014	FA	Saut	1	oui	D / I	C	SL	oui	F
[Moro 2014b]	Both	2014	FA	CF/Data	1	oui	D / I	Binaire	I	oui	P
SecSwift [de Ferrière 2019]	CFI	2019	FA	CF	X	oui	D / I	LLVM	F/B	oui	F
LZ TM	CFI	2019	FA	TI	X	non	D	LLVM	IP	oui	-
LZ LM	Data	2022	FA	DL	X	non	D	LLVM	IP	oui	-
LZ SSCF	CFI	2020	FA	CF	X	oui	D / I	LLVM	IP	oui	-

TABLE 5.1 – Comparaison de quelques contre-mesures logicielles

données (Data). Certaines contre-mesures visent les deux à la fois [Reis 2005]. La colonne "Modèle" indique le modèle de faute visé par la contre-mesure et la colonne "MF" précise si la contre-mesure vise les fautes multiples ("X") ou la faute unique ("1"). La colonne "Domaine" indique si la contre-mesure est issue du domaine des attaques en fautes ("FA") ou des fautes accidentelles ("FT").

La colonne "État" indique si la contre-mesure propage un *état interne* (comme un compteur [Heydemann 2019] ou des variables [Reis 2005, de Ferrière 2019]) ou si chaque application n'utilise que des états du programme initial (comme la duplication de test). La colonne "Tech." indique quelles *techniques* sont utilisées pour la contre-mesure, à savoir :

— *détection (D) :* l'état du programme est vérifié à certains points de contrôle et une réponse est déclenchée (arrêt du programme, arrêt du système ou signalement de l'erreur par exemple).
— *infection (I)* : les sorties du programmes sont rendues inutilisables ou invalides en cas de faute.
— *dilution (L)* : l'attaque est rendue plus difficile à réaliser, sans la bloquer complètement.

Pour la catégorie "Application", le *niveau de représentation* (colonne "Niveau"), précise à quel niveau la contre-mesure est appliquée (langage source, représentation intermédiaire ou compilation). La colonne "Granu." correspond à la *granularité* de l'application de la contre-mesure sur le programme, c'est-à-dire quels types de structures peuvent être protégées indépendamment. Il peut s'agir d'une granularité globale ("G"), de l'ordre d'une fonction ("F") ou du bloc de base ("B"), de l'ordre du point d'injection ("IP"), des instructions ("I") ou encore au niveau de structures logicielles ("SL") telles que les boucles ou les conditions. La colonne "Automatisation" indique si la contre-mesure est destinée à être appliquée manuellement ou automatiquement (par exemple dans un compilateur). La colonne "Syst." indique si l'application automatique est appliquée sur l'ensemble du programme [Lalande 2014] ("F") ou si les auteurs proposent une méthode pour réduire cette application en fonction de certains critères ou analyses [Reis 2005] ("P").

La *performance* (c'est-à-dire le surcoût de temps d'exécution, de consommation mémoire ou de taille de code) est aussi un paramètre capital pour le choix des contre-mesures à appliquer. Cette caractéristique n'est pas présentée dans la table ci-dessus.

5.1.2 Évaluation de contre-mesures

L'évaluation de contre-mesures vise à répondre à deux questions principales :

— Déterminer quelle contre-mesure est la plus efficace pour un modèle de faute donné parmi un ensemble de contre-mesures.

— Vérifier l'efficacité des protections insérées dans un programme protégé.

Deux grandes classes d'approches se dégagent de la littérature : l'*évaluation expérimentale* à l'aide d'outils d'analyse de robustesse et l'utilisation des *méthodes formelles*.

5.1.2.1 Évaluation expérimentale de la robustesse

Une première approche consiste à comparer les résultats d'analyse de robustesse d'un programme protégé par différentes contre-mesures. En fonction de la méthode d'évaluation de la robustesse utilisée, les métriques obtenues peuvent être très variées. Dans un article de 2013, Thießing et al [Theissing 2013] proposent une comparaison de différents schémas de protection au niveau assembleur. Ils évaluent un ensemble de 19 contre-mesures combinant protection du flot de contrôle et des données en comparant leur surcoût en terme de mémoire et de performance, ainsi que la couverture de détection pour des fautes injectées dans les programmes d'exemples. Les différentes versions protégées sont comparées à l'aide d'un simulateur sur une architecture ARM en effectuant des fautes bit-flip sur les instructions et la mémoire. Dans [Moro 2014c], Moro et al. proposent aussi une comparaison expérimentale de deux contremesures contre le saut d'instruction et le remplacement d'instructions au niveau binaire. Cette évaluation est réalisée à l'aide d'un simulateur. Dans [Dureuil 2016b], les auteurs présentent la collection FISSC, dont les exemples *verify_pin* et RSA présentés précédemment sont issus. Les versions protégées de chaque programmes sont évaluées en inversion de test avec Lazart [Potet 2014] au niveau LLVM ainsi qu'avec le simulateur Celtic [Werner 2022] au niveau binaire, qui utilise un modèle de mutation de données sur les instructions.

Une problématique majeure réside dans le fait que l'ajout de contre-mesures logicielles implique une augmentation de la surface d'attaque du programme, les portions ajoutées pouvant elles aussi être attaquées. Ce *paradoxe de dilution* [Dureuil 2016a], implique qu'il est difficile de comparer les métriques sorties par les outils pour différents programmes protégés.

De plus, ces approches visent généralement la faute unique. Comparer des programmes en fautes multiples est plus coûteux en termes de temps d'analyse.

5.1.2.2 Méthodes formelles

Des solutions basées sur les méthodes formelles ont été proposées pour l'analyse de programmes protégés. Cette section propose quelques exemples de solutions s'appuyant sur les méthodes formelles.

SymPFLIED [Pattabiraman 2008] (déjà abordé dans la section 2.5) permet à l'utilisateur de spécifier un ensemble de détecteurs (de tests) sur le programme. Ceux-ci sont composés de morceaux de code écrits dans le programme à analyser à l'aide de l'instruction spécifique `assert(p)` et d'éventuelles variables fantômes nécessaires pour vérifier un prédicat p, qui est quant à lui spécifié en tant qu'expression dans Maude. Dans [Pattabiraman 2012], un détecteur est un morceau de code externe vérifiant une propriété à l'aide d'une instruction `check`. Le détecteur est considéré comme non fautable.

Dans [Lalande 2014, Heydemann 2019], les auteurs proposent une vérification formelle d'une contre-mesure, basée sur le model checking. Ils parviennent ainsi à prouver que la

contre-mesure proposée protège effectivement contre le modèle de faute de saut d'instructions niveau C. De la même manière, Goubet et al [Goubet 2015] parviennent à vérifier l'efficacité d'une contre-mesure contre le remplacement d'instructions au niveau binaire. Cette approche garantit qu'aucune attaque n'est possible avec le modèle de faute considéré, contrairement à des approches basées sur le test qui considèrent des entrées fixées.

Martin et al. [Martin 2022] ont proposé une méthode d'analyse dans laquelle des oracles sont ajoutés manuellement pour vérifier que les contre-mesures protègent effectivement le programme contre le modèle de l'inversion de test. Les inversions de tests sont représentées par un appel à une fonction *mutated()* qui produit ou non une faute qui sera ajoutée à la condition par disjonction. Cette fonction vérifie que la limite de faute fournie par l'utilisateur n'est pas atteinte et retourne dans ce cas une valeur non contrainte. Des annotations (ACSL) sont appliquées automatiquement, vérifiant qu'aucune attaque ne peut atteindre la fin du programme sans qu'une contre-mesure ne soit déclenchée. Les annotations sont ensuite vérifiées à l'aide du plugin Weakest Precondition (WP) de Frama-C. Les auteurs parviennent notamment à détecter une erreur dans une des contre-mesures du projet Wookey, qui avait été découverte précédemment dans [ANSSI 2020].

5.1.3 Synthèse

Différentes approches ont été proposées en ce qui concerne l'analyse et l'évaluation de contre-mesures. La section 5.1.1 a montré quelques propriétés sur les contre-mesures logicielles proposées dans la littérature. Néanmoins les solutions de protection sont très diverses et leur comparaison n'est pas toujours facile, sans compter le *paradoxe de dilution* énoncé précédemment qui rend difficile la comparaison des métriques de robustesse, une contre-mesure pouvant ajouter de la surface d'attaque.

L'approche empirique par essai / erreur est fréquemment utilisée afin de mettre en place des protections dans un programme. Cette approche ne résiste cependant pas au passage à l'échelle dans le cadre de fautes multiples. En effet, les contre-mesures pouvant être attaquées, l'explosion combinatoire des chemins d'attaque rend difficile l'approche empirique dans ce contexte. De plus, les contre-mesures et les méthodes d'analyse sont très souvent évaluées vis-à-vis d'un modèle de faute particulier, rendant le rejeu de la méthodologie difficile dans un contexte différent.

La suite de ce chapitre s'intéresse à différentes stratégies de placement de contre-mesures, guidées par une recherche préalable des chemins d'attaques réussies avec Lazart. Ces algorithmes de placement peuvent être utilisés au sein d'outils de placement automatique de contre-mesures ou servir d'aide au développeur pour un placement manuel des contre-mesures.

5.2 Définitions et notions pour le placement de contre-mesures

Cette section s'intéresse à quelques contre-mesures automatiques proposées dans Lazart, présente leurs schémas de protection, et définit certaines notions qui seront nécessaires pour la définition des algorithmes de placement.

La section 5.2.1 présente les notions de localité et de pondération. La section 5.2.2 définit la notion d'adéquation d'une contre-mesure par rapport à un modèle de faute. La section 5.2.3 définit la notion de complétude de contre-mesures, elle aussi définie par rapport à un modèle de faute. Enfin, la section 5.2.4 propose une synthèse de cette section.

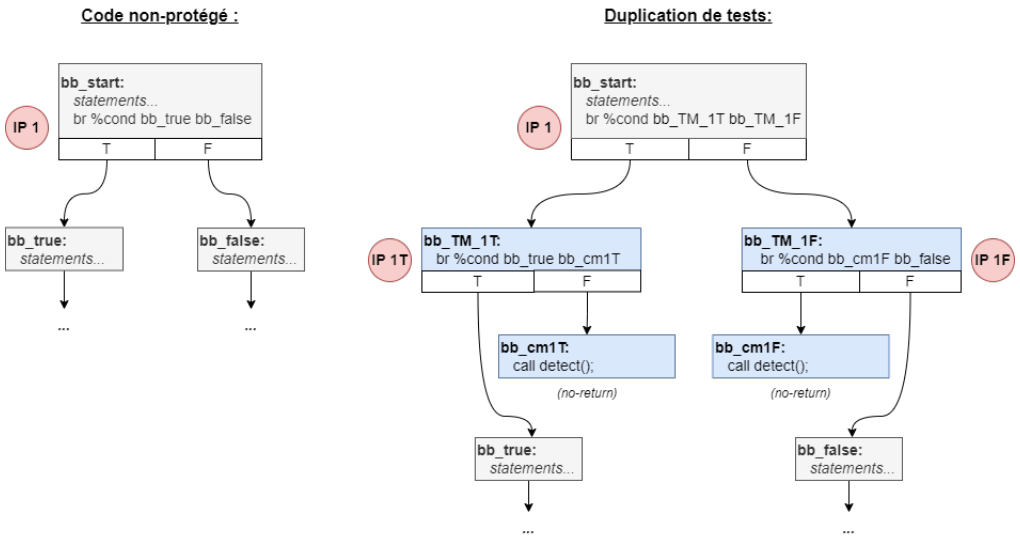

FIGURE 5.1 – Schéma de protection de la duplication de test

5.2.1 Localité et pondération

Les notions de *localité* et de *pondération* seront illustrées respectivement par la duplication de tests (Test Duplication (TD)) et la duplication de loads (Load Duplication (LD)), qui sont des contres-mesures locales, et la multiplication de tests (Test Multiplication (TM)) et la multiplication de loads (Load Multiplication (LM)) qui sont des contre-mesures locales pondérées. Ces contre-mesures ont une granularité de l'ordre du point d'injection et on s'intéressera à leur robustesse vis-à-vis des modèles de faute TI et DL.

5.2.1.1 Contre-mesures locales au point d'injection

La *duplication de test* (TD) est un exemple de Contre-mesure Locale (CL) relativement au modèle d'inversion de test. La figure 5.1 présente le schéma de transformation pour la protection d'un branchement conditionnel avec cette contre-mesure dans Lazart. Le *schéma de protection* correspond à la règle de transformation du programme lors de l'application de la contre-mesure sur ce point d'injection. Les blocs en gris correspondent au programme original et les blocs en bleu aux blocs ajoutés par la contre-mesure. Les points d'injection (pour le modèle TI) sont indiqués en rouge, à côté de l'instruction correspondante (`br`). Le branchement conditionnel est dupliqué pour chaque branche (`true` et `false`) et le programme est arrêté (appel à `detect`) si le résultat n'est pas celui attendu. La condition (`%cond`) n'est évaluée qu'une seule fois et est réutilisée dans les tests redondants.

Définition 5.2. *Soit P un programme et C une contre-mesure, on note $C(P)$ l'application de C sur P.*

Définition 5.3. *Soit C une contre-mesure à granularité d'un point d'injection, on note $C(P, IP)$ l'application de C localement sur le point d'injection IP.*

Définition 5.4. *On dit que deux contre-mesures C_1 et C_2 sont indépendantes (noté $\perp$) ssi, pour tout programme P, leur ordre d'application n'a pas d'incidence [1] sur le programme protégé obtenu : $C_1 \perp C_2 \leftrightarrow C_2(C_1(P)) = C_1(C_2(P))$.*

1. En d'autres termes, les programmes obtenus sont syntaxiquement équivalents quel que soit l'ordre d'application des contre-mesures.

Définition 5.5. *Pour un modèle de faute m, une contre-mesure C est dite locale, ssi, pour tout programme P, toutes les applications de C pour chaque point d'injection sont indépendantes :* $\forall IP_a, IP_b \in IP(P), C(P, IP_a) \perp C(P, IP_b)$.

TD est une contre-mesure locale (définition 5.5), puisqu'elle s'applique sur les points d'injection d'inversion de test (granularité IP) et chaque application est indépendante (définition 5.4). Cette notion de *localité* est importante pour le placement de contre-mesures, puisque dans le cas de contre-mesures locales, il est possible de représenter l'application de la contre-mesure comme un ensemble non ordonné de points d'injection.

5.2.1.2 Contre-mesures locales pondérées

La multiplication de tests est un exemple de contre-mesure locale pouvant être appliquée avec un niveau de pondération différent pour chaque point d'injection. On parle alors de Contre-mesure Locale Pondérée (CLP), et on appelle *profondeur* le coefficient d'application d'une contre-mesure pour un point d'injection donné (définition 5.6).

Définition 5.6. *Une contre-mesure locale est dite pondérée s'il est possible de l'appliquer avec une profondeur p positive. On note $C_p(P, IP)$ l'application d'une contre-mesure locale pondérée C sur un point d'injection IP avec une profondeur $p \in \mathbb{N}$.*

Ainsi, la *duplication de test* correspond à une *multiplication de test* de profondeur 1. La figure 5.2 présente le schéma de transformation de la multiplication de tests pour une profondeur 2 (Test Triplication (TT)). Les points d'injection sont indiqués dans les cercles rouges à côté de l'instruction de branchement correspondante. La multiplication de tests peut être vue comme l'application successive de duplications de tests sur les points d'injection introduits par l'itération précédente, en ignorant la protection des branches allant vers un bloc de détection (c'est-à-dire contenant un appel à *detect()* dans les schémas de protection présentés).

Les contre-mesures locales pondérées peuvent être appliquées avec une profondeur variable sur chaque point d'injection. L'intérêt de tels schémas dépend du fait qu'une application de profondeur $n + 1$ apporte plus de garanties de sécurité qu'une application de profondeur n, pour un modèle de faute m, ce qu'on va pouvoir quantifier dans la suite de ce chapitre.

5.2.1.3 Multiplication de loads

La multiplication de tests vise le modèle de l'inversion de test (TI). La multiplication de loads (LM) est une contre-mesure locale pondérée visant le modèle de la mutation de données lors les *loads* (DL).

La figure 5.3 présente le schéma de protection de la multiplication de loads avec une profondeur 0 (code non protégé), 1 (LD) et 2 (Load Triplication (LT)). Les points d'injection en rouge correspondent au modèle de faute TI tandis que les violets correspondent au modèle DL. Les blocs de base en jaune indiquent des blocs comportant à la fois des instructions du programme original et du code de contre-mesure. Dans le cas de la protection de profondeur 1, l'instruction `load` est dupliquée et les deux valeurs sont comparées. Si celles-ci sont différentes, le programme est arrêté, sinon le flot de contrôle revient à l'instruction suivant le `load` ciblé (ici dans un nouveau bloc `bb_LM_tail`). Dans le cas de protection de profondeur 2 ou plus, comme indiqué dans la partie droite de la figure, les `load`s dupliqués sont comparés à la valeur du `load` original, générant ainsi plusieurs branchements conditionnels successifs.

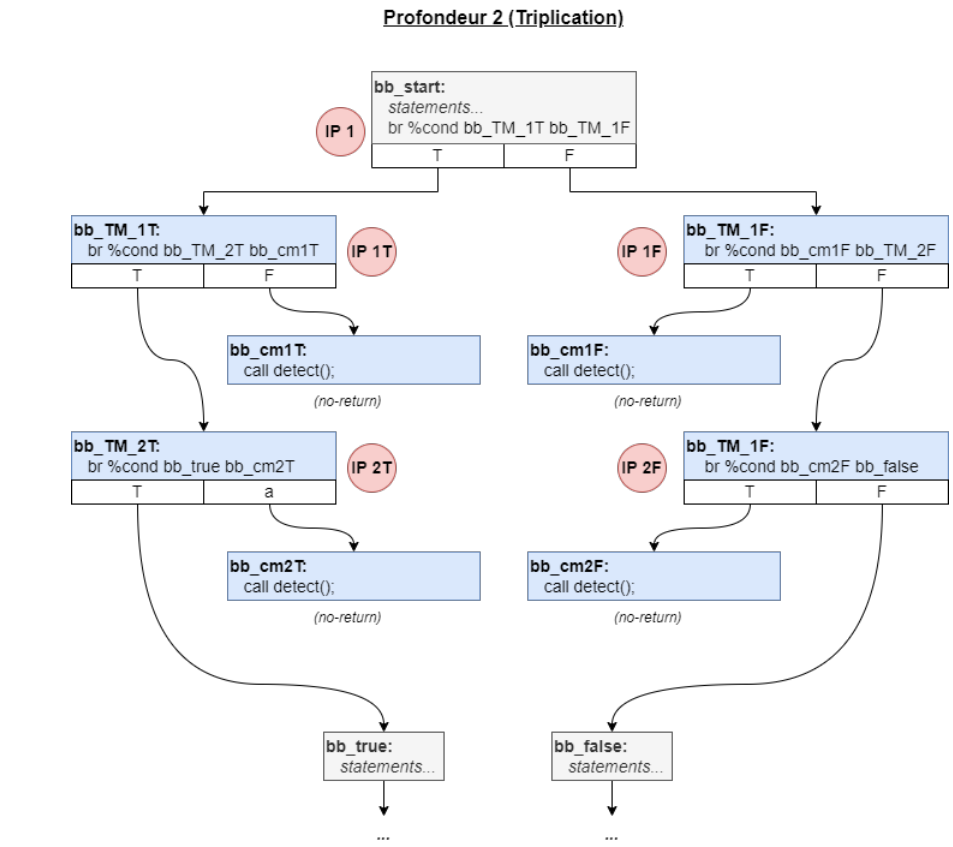

FIGURE 5.2 – Schéma de protection de la triplication de test

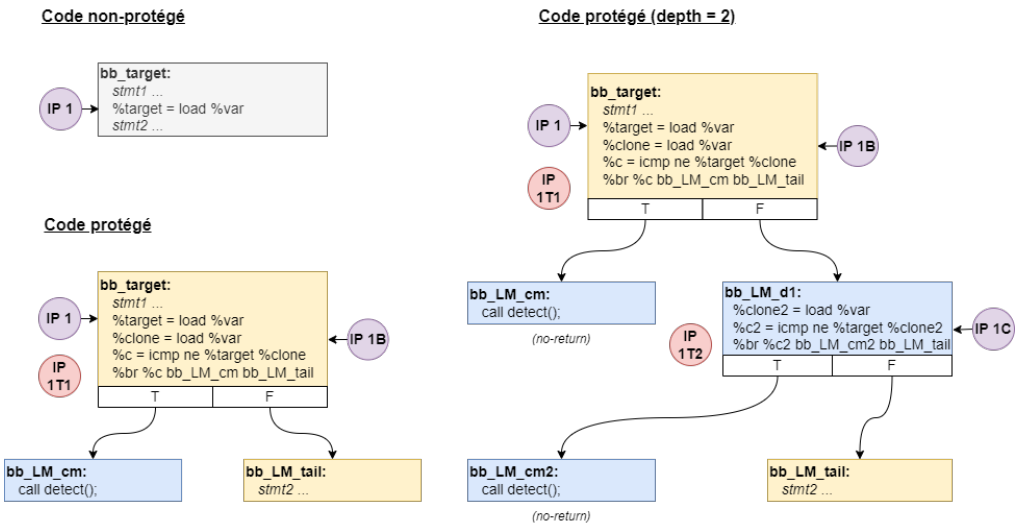

FIGURE 5.3 – Schéma de protection de la multiplication de load

5.2.2 Adéquation et coefficient de protection

Cette section s'intéresse à la propriété d'adéquation d'une contre-mesure, définie par rapport à un modèle de faute. Cette propriété est nécessaire pour établir des garanties de robustesse pour les différents algorithmes de placement. La section 5.2.2.1 présente une approche d'analyse de contre-mesures *en isolation*, visant à étudier les propriétés du schéma de protection en dehors d'un programme particulier. La section 5.2.2.2 détaille le cas d'une analyse dans le scénario où une entrée est déjà corrompue. La section 5.2.2.3 définit l'adéquation d'une contre-mesure locale à l'aide de cette analyse.

5.2.2.1 Analyse en isolation de schémas de protection

L'approche présentée ici consiste à s'intéresser aux comportements d'un schéma de protection en isolation, relativement à un modèle de faute. L'idée est d'explorer tous les chemins d'exécution fautée en prenant pour entrée le point d'injection (ici une instruction LLVM) en fonction du modèle de faute m afin de déterminer le nombre de fautes nécessaires pour le corrompre, appelé *coefficient de protection* (définition 5.7). Puisqu'un point d'injection est par définition attaquable en une faute, un schéma de protection avec $K > 1$ apporte des garanties sur la robustesse du schéma de protection.

Définition 5.7. *Soit S un schéma de protection sur un point d'injection, on appelle coefficient de protection K le nombre minimum de fautes nécessaires pour produire un comportement incorrect, pour un modèle de faute donné.*

L'analyse en isolation d'un schéma de protection pour un modèle de faute m nécessite de déterminer :

— Les *points d'entrée* et les *points de sortie* du schéma de protection.
— Les *entrées* du schéma de protection.
— Les *sorties* du schéma de protection.
— La *surface d'attaque* du schéma de protection par rapport au modèle m.

Points d'entrée et de sortie. Les *points d'entrée et de sortie* sont définis par le schéma de protection, mais également par le modèle de faute m. Pour un modèle de faute de saut par exemple, l'exploration exhaustive des comportements du schéma de protection nécessite de prendre en compte comme points d'entrée chaque point d'arrivée de saut possible et comme points de sortie les points d'injection de saut. Les schémas de LM et TM (voir figure 5.1, 5.2 et 5.3) ne contiennent qu'un seul point d'entrée correspondant à l'instruction sur laquelle le schéma de protection s'applique (respectivement `br` et `load`). Le schéma LM n'a qu'un seul point de sortie (le bloc `bb_LM_tail`) tandis que TM en possède deux (les blocs cibles `bb_true` et `bb_false`). Notez qu'on ne considère pas les blocs de détection comme des points de sorties.

Entrées. Les *entrées* correspondent à tout état du programme qui est utilisé par le schéma de protection. Dans le cadre de la contre-mesure TM, l'unique entrée est la valeur de condition `%cond`, utilisée par les instructions de branchement. Pour la contre-mesure LM, l'unique entrée est la valeur de la variable lue par les instructions `load`.

Sorties. Les *sorties* correspondent aux états du programme qui sont susceptibles d'être modifiés par le schéma de protection. Pour TM, il s'agit du bloc d'arrivée (bb_true ou bb_false). Pour LM, la sortie correspond à la valeur lue par l'instruction `load`[2]. Les sorties sont nécessaires pour déterminer l'objectif d'attaque de l'analyse en isolation, une attaque réussie du schéma TM correspondant à un branchement vers la mauvaise branche en sortie et

2. C'est à dire la valeur qui sera stockée dans le registre temporaire LLVM $\%target$.

pour LM au chargement de la mauvaise valeur lors du load (sans qu'aucune fonction `detect` ne soit déclenchée). Il est possible d'avoir plusieurs sorties si on considère des contre-mesures avec *état*, telles que *SecSwift CF* [de Ferrière 2019] citée précédemment.

Surface d'attaque. La *surface d'attaque* du schéma de protection correspond à tous les points d'injection du modèle m, que ce soit le point d'injection nominal (celui sur lequel le schéma s'applique) mais aussi les points d'injection introduits par la contre-mesure.

Définition 5.8 (Surface d'attaque)**.** *Soit C une contre-mesure et IP un point d'injection, on note $IPs(C(P, IP))$ l'ensemble des points d'injection introduits par l'application $C(P, IP)$.*

A partir de ces quatre paramètres, il est possible d'explorer les exécutions fautées du schéma de protection. Cette exploration peut être effectuée avec Lazart [3], comme montré dans les listings 5.1 et 5.2 qui correspondent à l'encodage en langage C des schémas de protection de la multiplication de tests, respectivement pour une profondeur de 1 et de 2. Les noms des blocs de base (correspondant à ceux des figures 5.1 et 5.2) sont indiqués en tant qu'étiquettes (labels). L'entrée du schéma de protection est représentée par l'entier `cond` et la branche de sortie est représentée sous la forme d'un booléen (branche *true* ou branche *false*). Les appels à `detect()` correspondent à une détection de l'attaque.

Le listing 5.3 correspond à la fonction principale de l'analyse avec Lazart. La condition d'entrée est déclarée comme variable symbolique (ligne 3) et l'objectif d'attaque consiste à retourner la mauvaise branche (vérifié ligne 7). La recherche des chemins d'attaque est effectuée en fautant la fonction correspondant au schéma de protection (ici `tm_scheme`) avec le modèle m (ici TI).

La table 5.2 présente les résultats de l'analyse en isolation pour différentes profondeurs de protection de la multiplication de tests. La colonne "Contre-mesure" indique le schéma de protection : P correspondant à une profondeur 0 (code non protégé), TD à la duplication de test (TM profondeur 1) et TT à la triplication de test (TM de profondeur 2). Les colonnes suivantes indiquent le nombre d'attaques trouvées pour chaque nombre de fautes et la dernière colonne "K_n" correspond au *coefficient de protection nominal*, c'est-à-dire le nombre minimum de fautes nécessaires pour valider l'objectif d'attaque.

Contre-mesure	0 faute	1 faute	2 fautes	3 fautes	4 fautes	K_n
TM_0 (P)	0	1	0	0	0	1
TM_1 (TD)	0	0	1	0	0	2
TM_2 (TT)	0	0	0	1	0	3

TABLE 5.2 – Évaluation en isolation de TM

5.2.2.2 Scénario corrompu

Afin d'être exhaustif sur l'exploration des comportements, il est nécessaire de se pencher sur le cas où les entrées de la contre-mesure sont déjà corrompues (par une faute précédente). Ainsi, la méthodologie d'analyse en isolation distingue deux scénarios, comme indiqué dans la figure 5.4 :

— Scénario Nominal (N) : toutes les entrées sont saines (traité dans la section précédente).
— Scénario Corrompu (C) : une entrée au moins est corrompue en amont du schéma de protection.

```
int tm_scheme(int cond) {
    bb_start:
    if(cond) {
        bb_TM_1T:
        if(cond) {
            bb_true:
            return true;
        } else {
            bb_cm1T:
            detect();
        }
    } else {
        bb_TM_1F:
        if(cond) {
            bb_cm1F:
            detect();
        } else {
            bb_false:
            return false;
        }
    }
}
```

Listing 5.1 – Encodage de TD (TM_1)

```
int tm_scheme(int cond) {
    bb_start:
    if(cond) {
        bb_TM_1T:
        if(cond) {
            bb_TM_2T:
            if(cond) {
                bb_true:
                return true;
            } else {
                bb_cm2T:
                detect();
            }
        } else {
            bb_cm1T:
            detect();
        }
    } else {
        bb_TM_1F:
        if(cond) {
            bb_cm1F:
            detect();
        } else {
            bb_TM_2T:
            if(cond) {
                bb_cm2F:
                detect();
            } else {
                bb_false:
                return false;
            }
        }
    }
}
```

Listing 5.2 – Encodage de TT (TM_2)

```
int main() {
    int entry;
    klee_make_symbolic(&entry, sizeof(entry), "cond");

    int br = tm_scheme(entry);

    _LZ__ORACLE((br != (entry ? true : false)));
}
```

Listing 5.3 – Fonction principale de l'analyse en isolation (scénario nominal)

Contre-mesure	0 faute	1 faute	2 fautes	3 fautes	4 fautes	K_c
TM_0 (P)	1	0	0	0	0	0
TM_1 (TD)	1	0	0	0	0	0
TM_2 (TT)	1	0	0	0	0	0

TABLE 5.3 – Évaluation en isolation de TM (scénario corrompu)

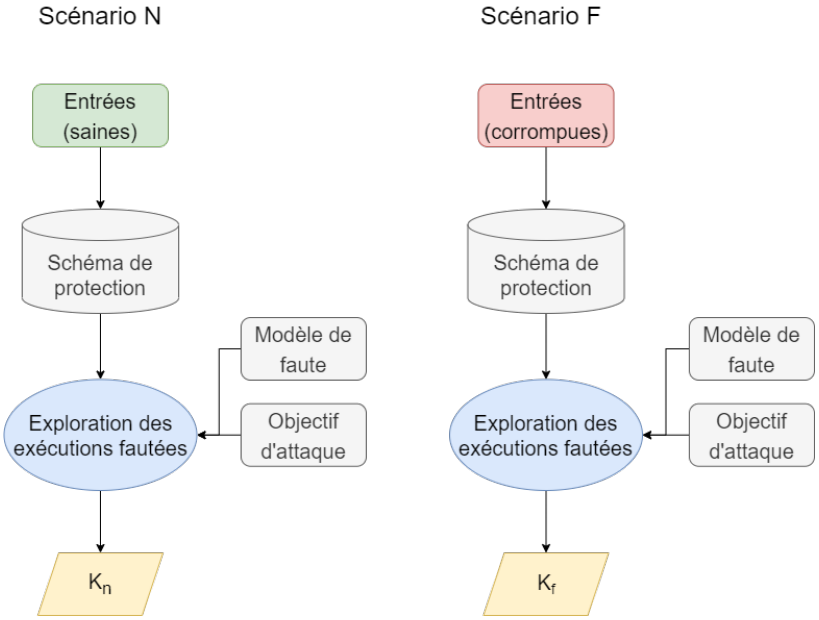

FIGURE 5.4 – Évaluation de schéma de protection en isolation

La table 5.3 présente les résultats (après analyse de redondance-équivalence) du scénario corrompu (C) pour l'exemple précédent, la condition d'entrée étant falsifiée (en changeant donc l'appel ligne 5 en `int br = tm_scheme(!entry);` [4]). On appellera respectivement *coefficient de protection nominal* (K_n) et *coefficient de protection corrompu* (K_c) les coefficients de protection obtenus pour respectivement les scénarios N et C. Pour les contre-mesures TM et LM et leurs modèles de faute respectifs TI et DL avec une pondération d'application p, on obtient $K_n = 1 + p$ et $K_c = 0$. Cela implique que, si les entrées d'un schéma de protection TM ou LM sont nominales, alors on a la garantie que le programme sera dans un état nominal en sortie du schéma de protection en moins de $1 + p$ fautes injectées localement. En revanche, dans le scénario corrompu, tous les schémas de protection produisent un comportement incorrect même en l'absence de faute, de la même manière que le point d'injection non protégé.

5.2.2.3 Adéquation des contre-mesures

Définition 5.9 (Adéquation)**.** *Pour un modèle de faute m, une contre-mesure locale C est dite adaptée ssi, pour tout programme P, son coefficient de protection nominal K_n est*

3. Le nombre de chemins à explorer dans une analyse en isolation est généralement petit. Ainsi, il est possible de vérifier manuellement que l'exploration symbolique est correcte et complète.

4. Dans le cas d'une entrée non booléenne, celle-ci serait contrainte comme étant différente de la valeur nominale (`_LZ__ORACLE(entry != expected_value);`.

strictement supérieur à 1.

Définition 5.10 (Perfection)**.** *Pour un modèle de faute m, une contre-mesure locale C est dite parfaite ssi, pour tout programme P, son coefficient de protection $K = min(K_n, K_c)$ est strictement positif.*

Une contre-mesure parfaite (définition 5.10) garantit que, quelles que soient les entrées d'une contre-mesure, tout état invalide du programme sera détecté. La section 5.5 revient sur l'existence de telles contre-mesures, mais on peut observer que ce n'est pas le cas des contre-mesures TM et LM, qui ont K_c nul. En effet, si les entrées sont déjà corrompues, alors le comportement du point d'injection protégé restera incorrect (mauvaise branche ou valeur incorrecte) sans qu'aucune faute ne soit injectée. En revanche, TM et LM sont des contre-mesures adaptées (définition 5.9) pour respectivement les modèles TI et DL (propriétés 5.1 et 5.2). Comme cela a été expliqué dans la section précédente, l'analyse en isolation avec l'exécution symbolique (via Lazart) génère peu de chemins pour nos exemples et il est donc possible de vérifier la complétude de l'exploration des chemins manuellement. La preuve de la propriété d'adéquation d'un schéma de protection par rapport à un modèle de faute pourrait aussi être effectuée à l'aide d'un outil d'analyse statique ou d'un assistant de preuve.

Propriété 5.1. *La multiplication de tests est une contre-mesure locale pondérée adaptée pour le modèle de l'inversion de test (TI).*

Propriété 5.2. *La multiplication de loads est une contre-mesure locale pondérée adaptée pour le modèle de la mutation de données sur les loads (DL).*

5.2.3 Complétude d'une contre-mesure et catalogue

Les contres-mesures TM et LM sont chacune des contres-mesures adaptées pour respectivement les modèles TI et DL. TM et LM sont aussi adaptées pour le modèle combiné de l'inversion de test et de la mutation de données ($TI + DL$). En effet, chacune a un coefficient de protection nominal K_n strictement supérieur à 1 lorsqu'on les étudie en fonction du modèle $m = TI + DL$.

Néanmoins, ni TM, ni LM, ne permettent indépendamment la protection de l'ensemble des points d'injection du modèle $TI + DL$. On dit alors que chacune de ces contre-mesures sont *incomplètes* pour le modèle $TI + DL$ (définition 5.11). Ainsi, on appellera Test and Load Multiplication (TLM) la contre-mesure locale pondérée correspondant à l'application de TM sur les points d'injection TI et LM pour les points d'injection DL. Cette contre-mesure est complète et adaptée pour le modèle $TI + DL$ (propriété 5.3).

Définition 5.11. *Une contre-mesure locale est dite complète pour un modèle de faute m, si elle propose une schéma de protection pour tous les points d'injection générés par le modèle de faute m.*

Propriété 5.3. *La multiplication de tests et de loads (TLM) est une contre-mesure locale pondérée adaptée et complète pour le modèle $TI + DL$.*

Les algorithmes de placement qui seront présentés par la suite considèrent un *catalogue* de contre-mesures. Ce catalogue associe un schéma de protection en fonction d'un point d'injection suivant son modèle de faute et un coefficient de protection. TLM peut être vu comme un catalogue de contre-mesure associant TM aux points d'injection TI et LM aux points d'injections DL, le schéma de protection précis qui sera retourné dépendant du

coefficient de protection demandé [5]. Les notions d'adéquation et de complétude peuvent être étendues à un catalogue de contre-mesures (définitions 5.12 et 5.13). Les algorithmes de placement présentés par la suite considèrent les catalogues totaux [6] (définition 5.14).

Définition 5.12 (Catalogue adapté)**.** *Un catalogue de contre-mesures $\mathcal{C}$ est adapté pour un modèle de faute m, si toutes les schémas de protection sont adaptés pour le modèle m.*

Définition 5.13 (Catalogue complet)**.** *Un catalogue de contre-mesures $\mathcal{C}$ est complet pour un modèle de faute m, s'il associe un schéma de protection pour tous les points d'injection du modèle de faute m.*

Définition 5.14 (Catalogue total)**.** *Un catalogue de contre-mesures $\mathcal{C}$ est dit total pour un modèle de faute m, s'il possède un schéma de protection pour tout points d'injection et pour tout coefficient de protection k, avec $1 \leq k \leq n+1$, pour m, avec $n \in \mathbb{N}^*$.*

Un catalogue de contre-mesures n'est donc finalement qu'une forme particulière de contre-mesure, permettant de garantir qu'un point d'injection est protégé qu'une fois par un schéma de protection (quelle que soit la profondeur d'application choisie). De cette manière, plusieurs contre-mesures (de l'ordre du point d'injection) qui ne sont pas indépendantes peuvent être appliquées de manière indépendante. En effet, les contre-mesures TM et LM ne sont pas indépendantes. LM introduit des points d'injections TI qui peuvent ainsi être protégés par une application de TM (en d'autres termes, $TM(LM(P)) \neq LM(TM(P))$). Il est donc nécessaire de s'assurer qu'un point d'injection ne soit pas protégé plusieurs fois de manière à garantir l'indépendance de l'application.

5.2.4 Synthèse

La table 5.4 présente un résumé des caractéristiques des différentes contre-mesures abordées dans cette section. La colonne "CM" indique le nom de la contre-mesure. Les colonnes "Adéquation" et "Complétude" indiquent respectivement si la contre-mesure est adaptée et complète, par rapport aux différents modèles de faute. Pour la complétude, la valeur "-" indique que la contre-mesure ne permet de protéger aucun point d'injection du modèle de faute. La colonne "Surface d'attaque" indique le nombre de points d'injection générés par la contre-mesure pour une profondeur p respectivement pour les modèles TI et DL (c'est-à-dire $IPs(C_p(P, IP))$). Ces valeurs ne sont pas indiquées pour TLM puisqu'elle correspond à celles de TM ou LM en fonction du modèle du point d'injection protégé.

CM	Adéquation			Complétude			Surface d'attaque	
	TI	DL	TI+DL	TI	DL	TI+DL	TI	DL
TM	Oui	Oui	Non	Oui	-	Non	$2p$	0
LM	Oui	Oui	Non	-	Oui	Non	p	$1+p$
TLM	Oui	Oui	Oui	Oui	Oui	Oui	-	-

TABLE 5.4 – Comparaison des contre-mesures locales pondérées

Les notions de contre-mesures *locales* et *pondérées* sont nécessaires pour les algorithmes de placement. Ces contre-mesures peuvent être représentées par l'ensemble des pondérations associées à chaque point d'injection de fautes du programme, une pondération de 0

5. Par exemple, pour un point d'injection TI et un coefficient de protection de 2, le catalogue retournerait la duplication de test.
6. Un catalogue total est complet et adapté.

correspondant à ne pas protéger le point d'injection. Si on considère des contre-mesures locales mais non pondérées (CL), leur application est aussi un ensemble de pondérations dont les valeurs sont comprises entre 0 et 1. Enfin, dans le cas de contre-mesures à granularité du point d'injection mais non indépendantes (C), alors l'ordre dans lequel les protections sont appliquées est important. Dans ce cas, l'application de la contre-mesure peut être représentée par une liste ordonnée de points d'injection à protéger. Les notions d'*adéquation* et de *complétude* sont quant à elles utiles pour déterminer les garanties en terme de robustesse pour un algorithme de placement.

Contre-mesure	Acronyme	Autre nom
Multiplication de tests	TM	-
Duplication de tests	TD	TM_1
Triplication de tests	TT	TM_2
Multiplication de loads	LM	-
Duplication de loads	LD	LM_1
Triplication de loads	LT	LM_2
Multiplication de loads et de tests	TLM	-

TABLE 5.5 – Abréviation pour les différentes contre-mesures locales de Lazart

Localité / Pondération	Adéquation / Complétude			
	Inadaptée (-I)	Adaptée (-A)	Complète (-C)	Parfaite (-P)
CM non locale (C)	C-I	C-A	C-C	C-P
CM Locale (CL)	CL-I	CL-A	CL-C	CL-P
CM Locale Pondérée (CLP)	CLP-I	CLP-A	CLP-C	CLP-P

TABLE 5.6 – Abréviation pour les différentes catégories de contre-mesures à granularité IP

Les tables 5.5 et 5.6 proposent un résumé des abréviations qui seront utilisées dans la suite de ce chapitre. La table 5.5 indique les abréviations utilisées pour les contre-mesures étudiées ici. La table 5.6 indique les abréviations concernant les différentes classes de contre-mesures. Par exemple, CLP-AC correspond à une Contre-mesure Locale Pondérée, Adaptée et Complète (pour un modèle de faute donné).

5.3 Placement de contre-mesures adaptées

L'approche de placement de contre-mesures présentée dans ce chapitre se base sur l'analyse en isolation des schémas de protections et sur une exploration des traces d'attaques réussies, par rapport à un modèle de faute et un objectif d'attaque. L'analyse en isolation fournit des garanties sur le comportement d'un point d'injection protégé, et les traces d'attaques permettent de déterminer quels sont les points d'injection à protéger. Les algorithmes présentés ici prennent ainsi en entrées :

— Le catalogue de contre-mesures $\mathcal{C}$.
— Le modèle d'attaquant $M = \{m, \phi\}$, avec m le modèle de faute et ϕ l'objectif d'attaque.
— Le nombre de fautes n pour lequel on souhaite être robuste.

Le catalogue $\mathcal{C}$ est un catalogue total (définition 5.14) pour le modèle m. Les garanties de robustesse sont données en supposant que l'énumération des chemins d'attaque est *correcte*

et *complète* (définitions 3.5 et 3.6). Les algorithmes génèrent un programme P' protégé dont les garanties dépendent de l'algorithme et du catalogue, et visent :

— La *robustesse* du programme P' en n fautes.
— L'*optimalité* du placement.

La section 5.3.1 présente trois algorithmes de placement systématique. La section 5.3.2 décrit le placement par bloc, dans lequel seul un IP par trace d'attaque est protégé. La section 5.3.3 traite du placement réparti où différents IPs peuvent être protégés avec des coefficients de protection différents. Finalement, la section 5.3.4 discute des garanties lorsque le catalogue ne dispose pas d'un schéma avec un coefficient de protection suffisant et discute de certaines variantes des algorithmes précédents.

5.3.1 Placement systématique

Lorsqu'on s'intéresse à un catalogue total pour le modèle de faute m, appliquer un schéma de protection avec un coefficient de protection nominal $K_n = n + 1$ sur tous les points d'injection garantit une robustesse en au moins n fautes du programme protégé. En effet, l'analyse en isolation garantit que le comportement du schéma de protection est nominal en moins de $n+1$ fautes si les entrées sont nominales. Si tous les points d'injection sont protégés avec $K_n = n + 1$, il faudra $n + 1$ fautes pour casser indépendamment chaque schéma et aucun schéma de protection ne pourra être corrompu en n fautes.

On distinguera trois versions de l'algorithme de placement systématique :

— Le placement systématique naïf (`naïf`) : tous les points d'injection du programme sont protégés avec $K_n = n + 1$.
— Le placement systématique sur les attaques (`atk`) : tous les points d'injection intervenant au moins une fois dans une attaque sont protégés avec $K_n = n + 1$.
— Le placement systématique sur les minimales (`min`) : seuls les points d'injection intervenant dans les attaques minimales sont protégés avec $K_n = n + 1$.

Si l'énumération des chemins fautés est correcte et complète, alors protéger les points d'injection intervenant dans les attaques est suffisant. De plus, la protection des attaques minimales est suffisante, puisque si une attaque a' est redondante par rapport à une attaque a, protéger a pour n fautes implique que a' sera robuste en $n + 1$ fautes au minimum.

Le listing 5.4 correspond au pseudo-code de l'algorithme de placement systématique sur les attaques minimales (`min`) prenant en entrées le catalogue $\mathcal{C}$, le programme P, le modèle d'attaquant M et le nombre de fautes n pour lequel le programme P' doit être robuste. Les attaques minimales sont obtenues à partir d'une analyse d'attaque et une analyse de redondance (lignes 3 et 5) puis chaque point d'injection intervenant au moins une fois dans une attaque minimale se voit assigner un coefficient de protection de $n+1$ (lignes 11 à 13). Le programme P' est généré à partir de l'ensemble des coefficients de protection (lignes 16 à 19) en sélectionnant le schéma de protection à appliquer à partir du catalogue (représenté par la fonction `get_cm` ligne 18).

```
1  def placement_min(C: Catalog, P: Program, M: AttackModel, n: int):
2      # Get sucessfull non-detected attacks.
3      attacks = T_s(P, M, n)
4      # Filter with minimals attacks.
5      minimals = RedundacyAnalysis(attacks).minimals()
6
7      # Initial protection factors Kn at 1 for all IP.
8      required_kn = { IPA: 1, IPB: 1, ..., IPN: 1 }
9
10     # Apply ponderation of n for all IP in traces
11     for attack in minimals:
12         for IP in attack:
```

```
13              required_kn[IP] = n + 1 # Make IP robust en n faults.
14
15      # Generation of P'
16      P` = P
17      for IP, kn in required_kn:
18          S = C.get_cm(IP.model(), kn) # Select protection scheme from catalog
19          P` = S(P`, IP) # Apply local protection
20      return P`
```

Listing 5.4 – Algorithme de placement systématique `min`

5.3.2 Placement par bloc

L'algorithme de placement systématique sur les attaques minimales protège l'ensemble des points d'injection intervenant dans une attaque. L'algorithme de placement par bloc (`bloc-h`) présenté dans cette section vise à protéger au moins un point d'injection par attaque minimale, avec $K_n = n + 1$.

```
1   def placement_bloc_h(C: Catalog, P: Program, M: AttackModel, n: int):
2       # Get sucessfull non-detected attacks.
3       attacks = T_s(P, M, n)
4       # Filter with minimals attacks.
5       minimals = RedundacyAnalysis(attacks).minimals()
6
7       # Initial protection factors Kn at 1 for all IP.
8       required_kn = { IPA: 1, IPB: 1, ..., IPN: 1 }
9
10      # For all attacks by faults count.
11      for order in 1 to n:
12          # Loop trought order-faults attacks by number of associated redundant attacks.
13          for attack in minimals.where(order=order).sort_by(Minimals):
14              if is_protected(attack, required_kn):
15                  continue
16              # Make attack robust in n faults
17              IP = select IP in attack with most occurence
18              required_kn[IP] = n
19
20      # Generation of P'
21      P` = P
22      for IP, kn in required_kn:
23          S = C.get_cm(IP.model(), kn) # Select protection scheme from catalog
24          P` = S(P`, IP) # Apply local protection
25      return P`
```

Listing 5.5 – Algorithme de placement par bloc avec heuristique `bloc-h`

Le listing 5.5 présente le pseudo-code de l'algorithme de placement par bloc avec heuristiques. Pour chaque nombre de fautes, toutes les attaques minimales sont parcourues. Si une attaque n'est pas protégée (c'est-à-dire qu'il n'y a pas au moins un IP protégé au niveau n, vérifié par la fonction `is_protected` ligne 14), un IP est sélectionné pour être protégé (ligne 17). La pondération de protection obtenue dépend ainsi de deux leviers :

- *l'ordre dans lequel les traces sont protégées* : l'algorithme utilise une heuristique consistant à parcourir les attaques minimales uniquement, par ordre de leur nombre de fautes, et par nombre de traces redondantes associées ($Red(a)$) décroissant [7].
- *le point d'injection sélectionné lors de la protection d'une attaque* : ici l'heuristique consiste à sélectionner les IPs en fonction de leur nombre d'occurrences dans l'attaque

7. C'est-à-dire que si deux traces ont le même nombre de fautes et sont minimales, on privilégiera les attaques avec le plus grand nombre d'attaques redondante associées.

considérée, puis en fonction de leur nombre de déclenchements total au sein des attaques minimales (obtenu à l'aide d'une analyse de point chauds, voir section 3.4.5).

Dans le cas du placement par bloc, protéger un IP avec un coefficient de protection $n+1$ dans chaque attaque garantit que chaque attaque nécessitera au moins $n+1$ fautes pour être réussie. Toute attaque en moins de $n+1$ fautes sera soit bloquée par un IP protégé, soit une attaque non réussie (les sorties produites par l'IP protégé étant nominales), ce qui est garanti par l'exploration des chemins d'exécution fautés de P.

5.3.3 Placement réparti

Le placement par bloc présenté précédemment impose la protection avec un coefficient de protection $K_n = n+1$ pour au moins un IP par trace d'attaque minimale. Le placement réparti s'autorise à protéger avec $K_n < n+1$ les IPs, de manière à répartir les protections sur plusieurs IPs.

La section 5.3.3.1 explique pourquoi le placement peut être réparti dans une attaque sans risquer d'introduire des nouveaux chemins d'attaque. La section 5.3.3.2 ramène la problématique du calcul optimal du placement réparti à un problème d'optimisation linéaire en nombre entier. Finalement, la section 5.3.3.3 présente l'algorithme de placement réparti optimal.

5.3.3.1 Protection des traces d'attaque

La figure 5.5 présente le principe de la combinaison de l'analyse en isolation et l'exploration des traces d'attaques. Les blocs "IP X" correspondent aux déclenchements de fautes sur un IP X. L'analyse en isolation (en haut) vise à abstraire les comportements possibles d'un point d'injection, que celui-ci soit protégé ou non. Tous les différents chemins ajoutés par la protection du point d'injection sont explorés dans l'analyse en isolation, permettant de déterminer combien de fautes doivent être dépensées pour obtenir une sortie invalide. Dans l'exemple présenté sur la figure 5.5, deux chemins existent pour l'IP non protégé (à gauche) : le chemin d'exécution nominale produisant une sortie nominale (en vert) et le chemin fauté produisant une sortie invalide (en rouge). Pour la version protégée avec $K_n = 2$ (à droite), l'attaque de IP_{B2} introduit par la contre-mesure permet d'obtenir un comportement fauté en deux fautes (le chemin avec une seule faute étant bloqué).

L'exploration des traces d'attaque donne des informations sur les relations entre les entrées et les sorties des différents points d'injection. L'objectif est de déterminer si un IP peut être protégé partiellement (avec un coefficient de protection inférieur à $n+1$), sans que la sortie invalide qu'il produira n'introduise de nouveaux chemins fautés qui n'auraient pas été pris en compte. Dans la figure 5.5, l'exploration des chemins fautés (en bas), contient des informations sur les effets possibles d'une sortie corrompue d'un point d'injection. Les losanges verts indiquent que l'objectif d'attaque n'est pas validé (attaque non réussie), et les losanges rouges indiquent une attaque réussie.

Par exemple, si on s'intéresse à la trace 2, comportant la séquence de points d'injection déclenchés $\{IP_C, IP_D\}$, on peut déterminer qu'un coefficient de protection de 3 pour IP_C serait suffisant pour obtenir une robustesse en 3 fautes [8]. On peut alors se poser la question de si une attaque en 3 fautes est possible après la sortie invalide de l'IP IP_C, qui ne demande que 3 fautes pour être corrompue. La réponse est donnée par l'exploration complète des traces d'attaques. La trace 4 est une attaque non réussie et n'est donc pas à prendre en

8. Pour $K_n = 3$, la Trace 4 deviendrait une attaque en 4 fautes, une faute étant nécessaire pour casser le point d'injection IP_D non protégé.

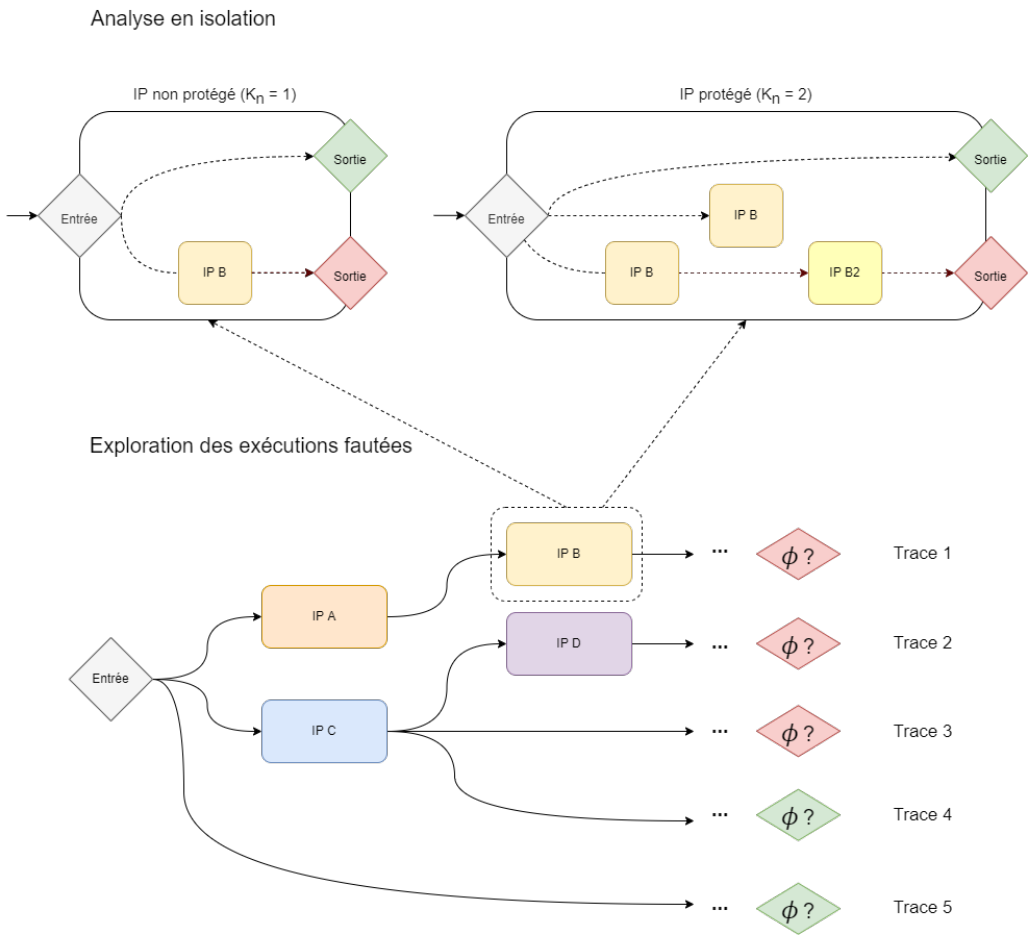

FIGURE 5.5 – Principe général du placement avec analyse en isolation

compte. En revanche la trace 3 indique que si IP_C n'est protégé qu'avec un coefficient de protection de 3, alors la sortie invalide de IP_C peut produire une attaque en 3 fautes. Un coefficient de protection d'au moins 4 est nécessaire sur IP_C pour garantir la robustesse en 3 fautes.

L'objectif est de pouvoir substituer les comportements des IPs protégés à ceux des IPs non protégés qui sont analysés dans l'exploration des exécutions fautées sur P, de manière à pouvoir prévoir comment un IP protégé va se comporter dans le contexte du programme. L'analyse en isolation abstrait les comportements fautés dans l'IP, mais il est nécessaire de garantir que les comportements produits par un point d'injection fauté sont bien inclus dans les modèles de faute utilisés dans l'exploration préliminaire des exécutions fautées. Dans le cadre des modèles de faute TI et DL, les sorties invalides d'un IP fautés sont en effet incluses dans les modèles de Lazart :

— Pour TI, les différents schémas de protection de TM peuvent produire une sortie vers la mauvaise branche.
— Pour DL, les différents schémas de protection de LM peuvent produire une valeur erronée lue par le `load`, qui correspond au modèle DL.

Ce ne serait pas le cas avec tous les modèles. Par exemple une mutation de donnée en mise-à-zéro, pourrait potentiellement produire des comportements différents (comme une

mise-à-un) en fonction du schéma de protection. Pour garantir la possibilité de substituer les IP protégés et non protégés, il serait nécessaire de montrer que les schémas de protection ne peuvent produire que le comportement nominal ou des mises-à-zéro, ou bien utiliser un modèle de mutation de donnée général (symbolique) dans l'exploration des chemins fautés afin de garantir cette inclusion.

5.3.3.2 Un problème d'optimisation linéaire

La recherche des meilleurs placements répartis (dont le total des pondérations de protection appliquées et minimal) correspond à la recherche des ensembles de coefficients de protection minimaux tels que toute attaque réussie dans $T(P', M)$ nécessite au moins $n+1$ fautes. L'idée est de couvrir toutes les attaques, chaque attaque donnant un ensemble de contraintes sur les coefficients de protection à appliquer à chaque IP pour obtenir la garantie de robustesse. On peut notamment remarquer :

— si plusieurs IPs apparaissent dans une attaque, alors chaque IP peut être protégé indépendamment pour obtenir au final une attaque en plus de n fautes.
— si un point d'injection ip apparaît x fois dans une attaque, alors une protection sur ip ajoute la nécessité de $x * K_n$ fautes supplémentaires.

Les contraintes imposées par une attaque a peuvent se traduire par une inégalité C_a de la forme $\alpha x_1 + ... + \gamma x_n \geq n + 1$, avec n le nombre de fautes pour lequel P doit être robuste, $\{x_1, ..., x_n\}$ les coefficients de protection des IPs $\{1, ..., n\}$ et $\{\alpha, ..., \gamma\}$ les coefficients correspondants au nombre de fois où chaque point d'injection se déclenche dans l'attaque. La figure 5.6 présente des exemples de traces d'attaques à partir desquelles les inégalités sont générées (indiquées à droite), où *faute X* désigne le déclenchement d'une faute sur le point d'injection X.

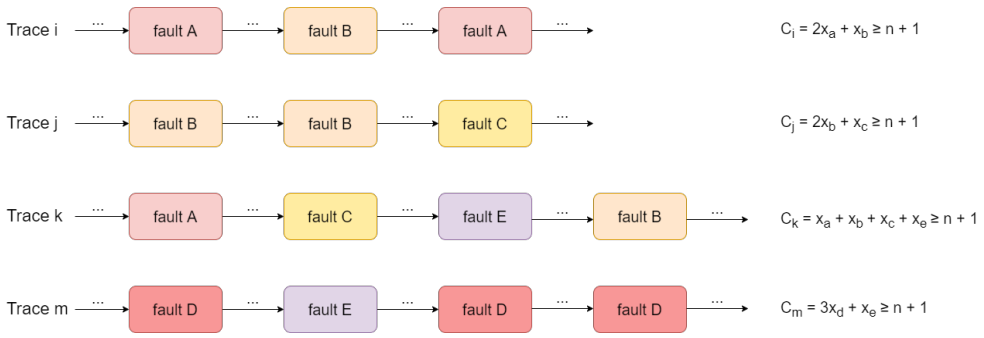

FIGURE 5.6 – Inégalités sur les coefficients de protection à partir des attaques

L'objectif est de trouver les ensembles de coefficients ($\{x_1, ..., x_n\}$) validant les contraintes C_i et minimisant la fonction $x_1 + x_2 + ... + x_n$. Ce problème est une instance du problème d'optimisation linéaire en nombre entier, ILP, qui est NP-complet.

5.3.3.3 Algorithme de placement réparti optimal

Le listing 5.6 présente l'algorithme de placement réparti optimal (*rep-opt*) qui s'appuie aussi sur les attaques minimales. Dans un premier temps, les contraintes du problème d'optimisation linéaire sont générées à partir des attaques (fonction `compute_constraint` ligne 12) et le problème d'optimisation linéaire est résolu dans la fonction `solve_ilp` (ligne 14), qui fournit ainsi l'ensemble des coefficients minimisant la somme des protections appliquées.

Le programme P' est ensuite généré en appliquant les contre-mesures avec les coefficients de protection obtenus.

```
def placement_rep_opt(C: Catalog, P: Program, M: AttackModel, n: int):
    # Get sucessfull non-detected attacks.
    attacks = T_s(P, M, n)
    # Filter with minimals attacks.
    minimals = RedundacyAnalysis(attacks).minimals()

    # Initial protection factors Kn at 1 for all IP.
    required_kn = { IPA: 1, IPB: 1, ..., IPN: 1 }

    constraints = [] # constraints for ILP
    for attack in minimals:
        constraints += compute_constraint(attack)

    required_kn = solve_ilp(constraints, required_kn)

    # Generation of P'
    P` = P
    for IP, kn in required_kn:
        S = C.get_cm(IP.model(), kn) # Select protection scheme from catalog
        P` = S(P`, IP) # Apply local protection
    return P`
```

Listing 5.6 – Algorithme de placement réparti optimal `rep-opt`

Dans le cadre d'un catalogue adapté et complet pour le modèle m, P' est robuste en au moins n fautes et la somme des coefficients de protection appliqués est minimale.

5.3.4 Autres algorithmes de placement

Trois approches de placement ont été abordées précédemment : le placement systématique (algorithmes *naïf*, *atk* et *min*), le placement par bloc par heuristiques (*bloc-h*) et le placement réparti optimal (*rep-opt*). Cette section discute de certaines variantes possibles de ces algorithmes.

Catalogue non total Les sections précédentes ont considéré des catalogues qui possèdent toujours un schéma de protection avec le coefficient de protection demandé. On considérera ici un catalogue non total, qui dispose d'une fonction *catalog.get_ cm(IP, k)*, retournant le couple (S, k'), avec S le schéma de protection et k' le coefficient de protection de S. Deux cas peuvent être distingués :

— *Sur-protection* : le catalogue fournit le schéma de protection avec k' le plus petit, tel que $k' > k$.

— *Sous-protection* : le catalogue fournit un schéma avec $k' < k$.

La *sur-protection* préserve la robustesse du programme protégé. Dans le cas de l'algorithme optimal néanmoins, il devient nécessaire d'ajouter une contrainte supplémentaire sur les coefficients de protection requis pour certains IPs. Cela peut donc faire sortir le problème du cadre de l'ILP (ce qui ne le rend pas pour autant indécidable), en ajoutant des contraintes du type $X_a \neq 2$ (pour représenter l'absence de schéma pour $K_n = 2$). Dans le cas d'une *sous-protection*, le problème de sortie du cadre du problème ILP se pose aussi. Plus encore, la robustesse du programme peut être remise en question. Si un autre IP peut être protégé avec un coefficient de protection suffisant, alors il est possible de garantir la robustesse du programme (quelle que soit l'approche de placement utilisée). S'il existe des traces d'attaques pour lesquelles il n'est pas possible de protéger pour n fautes en raison d'un manque de schéma de protection adapté, alors la robustesse en n fautes du programme

P' est perdue. Ces traces d'attaques sont cependant connues au moment du placement et peuvent être retournées à l'utilisateur, permettant d'anticiper les traces d'attaques du programme P' sans avoir à en explorer les chemins fautés et de déterminer le nombre de fautes pour lequel P' est robuste (en calculant la somme des coefficients de protection des IP de chaque attaque non protégée).

Placement réparti par heuristique On peut construire l'algorithme *rep-h* correspondant au placement réparti utilisant des heuristiques plutôt qu'un problème ILP. Cet algorithme fonctionne comme *bloc-h* (listing 5.5), en combinant une heuristique de parcours des traces et une heuristique de sélection des IPs à protéger dans une trace. Cette approche peut être préférée à l'approche *rep-opt* (listing 5.6) dans le cas où :

— le problème ILP de l'approche optimale est trop complexe pour être résolu.
— certaines contraintes sur le catalogue sortent du cadre du problème ILP standard.

Placement optimal par bloc L'algorithme *bloc-h* ne garantit pas l'optimalité du placement par bloc. Un algorithme de placement optimal par bloc (visant à protéger le moins d'IPs possible avec un $K_n = n + 1$) est envisageable. Le placement *bloc-h* nécessite cependant des contraintes qui ne sont pas de l'ordre du problème ILP (les coefficients n'étant plus bornés entre 1 et $n+1$ mais ne pouvant prendre que l'une des deux valeurs). Cet algorithme correspond à rechercher l'ensemble minimum d'IP qui couvrent toutes les traces d'attaques minimales au moins une fois, ce qui se rapproche de l'algorithme de sélection présenté dans le chapitre suivant (voir section 6.4.1.3).

5.4 Expérimentation

Cette section présente quelques expérimentations qui ont été menées sur les algorithmes de placement présentés précédemment :

— *naïf* (section 5.3.1) : protection avec $K_n = n + 1$ de tous les IPs du programme.
— *atk* (section 5.3.1) : protection avec $K_n = n + 1$ de tous les IPs intervenant dans les attaques.
— *min* (section 5.3.1) : protection avec $K_n = n + 1$ de tous les IPs intervenant dans les attaques minimales.
— *bloc-h* (section 5.3.2) : protection avec $K_n = n + 1$ d'au moins un IP par attaque minimale avec heuristiques.
— *rep-opt* (section 5.3.3) : algorithme optimal pour la protection répartie entre les IPs.

La figure 5.7 présente la méthodologie expérimentale utilisée. Chaque algorithme prend en entrée les traces d'attaques générées par Lazart, et retourne les protections à appliquer pour générer le programme P'. Une analyse d'attaque est effectuée sur P', permettant ainsi de déterminer si le programme est robuste en n fautes (ce qui n'est garantit que si l'exécution symbolique est complète et correcte). Pour l'algorithme de placement optimal (*rep-opt*), la résolution du problème d'optimisation linéaire a été réalisée avec un outil en ligne [9].

La section 5.4.1 décrit les catalogues de contre-mesures utilisés en fonction du modèle de faute considéré. La section 5.4.2 présente les programmes étudiés et les objectifs d'attaques et la section 5.4.3 discute des résultats obtenus.

9. L'annexe C.1 présente les fichiers de contraintes fournis à l'outil en ligne (`https://online-optimizer.appspot.com`).

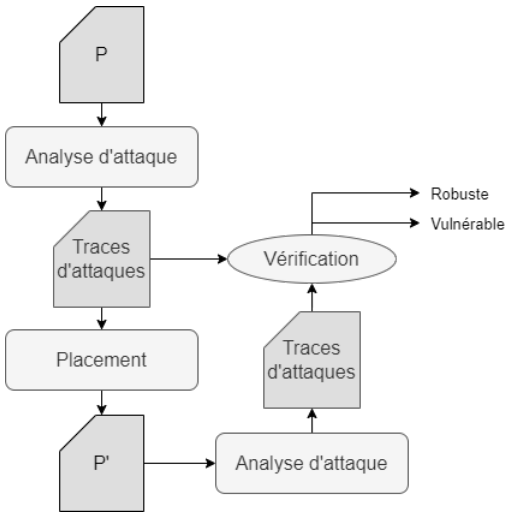

FIGURE 5.7 – Méthodologie expérimentale pour les algorithmes de placement

5.4.1 Catalogue de contre-mesures et inversion de branches

La section 5.2 a présenté la contre-mesure de multiplication de tests qui est adaptée pour le modèle de l'inversion de test. Ses schémas de protection protègent à la fois la branche *vrai* et la branche *faux* de chaque test.

Dans ces expérimentations, on s'intéressera à une variation de cette contre-mesure qui vise à protéger une seule branche d'un test, permettant ainsi de protéger chaque branche avec une pondération différente. Cela nécessite de considérer un autre modèle que l'inversion de test, qu'on appellera *inversion de branche* (Branch Inversion (BI)), et pour lequel les points d'injection de la branche *vrai* et la branche *faux* sont distincts.

Lazart ne supporte pas directement le modèle de l'inversion de branche, mais il est possible d'appliquer une analyse en inversion de test et de distinguer à posteriori les fautes concernant la branche *vrai* et la branche *faux*.

Les expérimentations utilisent un catalogue de contre-mesures complet et adapté pour le modèle de faute considéré. Ce catalogue retourne le schéma de protection à appliquer pour un point d'injection et un coefficient de protection donnés. Dans le cadre de BI, on utilisera donc la multiplication de test sur la branche correspondant au point d'injection. Pour le modèle DL, on utilisera la multiplication de load. Le catalogue garantit ainsi qu'un schéma de protection de profondeur n offre un coefficient de protection de $n + 1$.

5.4.2 Programmes utilisés

Les programmes étudiés dans cette section sont les suivants :
— *verify pin 2b* (`vp2b`).
— *firmware updater 1* (`fu1`).
— *memcmps* (`memcmps3`).

`vp2b` correspond à une version modifiée de *verify_ pin* 2 (voir annexe B.1), qui comprend les booléens endurcis et la boucle en temps constant mais ne possède pas de contre-mesure vérifiant le compteur de boucle (l'idée étant de placer les schémas de protection). `vp2b` est analysé uniquement avec le modèle BI et avec l'objectif d'attaque visant à s'authentifier avec un PIN faux. Les tableaux d'entrées sont symboliques, de taille 4 et différents sur au moins un octet.

`fu1` (voir section 3.5.1.1) à l'inverse contient une duplication de test systématique et considère les modèles BI et DL. L'objectif d'attaque vise soit à charger un micro-programme corrompu, soit à ne pas mettre à jour le micro-programme.

`memcmps3` correspond à une version durcie du programme présenté dans la section 3.3.1 (voir annexe B.2), dans laquelle quatre appels successifs à `memcmp` sont effectués avec différents masques. Les tableaux d'entrées sont symboliques et différents (sur au moins un octet) et l'objectif est de retourner *vrai* malgré l'inégalité des tableaux. Le programme est analysé avec les modèles BI et DL.

5.4.3 Comparaison des algorithmes de placement

La table 5.4.3 présente les résultats obtenus pour les différents programmes. La colonne "Expé" indique les paramètres de l'expérimentation avec "P" le programme utilisé, "m" le modèle de faute considéré et "IPs" le nombre total de points d'injection dans le programme. La colonne "Algos" indique quel algorithme de placement est utilisé. La colonne "$\sum$ de protections" indique la somme des pondérations de protection [10] appliquées par le placement. Enfin la colonne "Robuste" indique si le programme protégé P' est robuste en n fautes après une analyse de vérification avec Lazart.

Comme attendu, l'approche par bloc (*bloc-h*) offre de meilleurs résultats que les approches systématiques. De la même manière, le placement réparti (*rep-opt*) produit un poids de protection plus faible.

Pour l'exemple `vp2b`, on constate que les algorithmes *min* et *atk* sont équivalents, tous les IPs des attaques intervenant aussi dans les attaques minimales. Les algorithmes *bloc-h* et *rep-opt* fournissent les mêmes résultats, ce qui s'explique par un grand nombre d'attaques en 1 faute qui imposent une pondération de $n \in \mathbb{N}^*$ aux points d'injection concernés pour être résistant en n fautes.

Pour `fu1`, seuls deux points d'injection sur les données sont présents et l'un d'entre eux intervient dans une attaque en une faute. Ce même point d'injection est donc à protéger avec un coefficient de protection $K_n = n + 1$ dans le cas du modèle $BI + DL$. On observe pour le modèle $BI + DL$ que les algorithmes proposent tous des résultats différents, mettant en évidence la relation d'ordre entre les résultats de chaque algorithme.

Pour l'exemple `memcmps3`, on peut constater un net avantage des algorithmes prenant en compte les attaques pour le modèle BI. En effet, une seule attaque est possible avec ce modèle, et consiste à inverser chacun des quatre appels à `memcmp`. L'algorithme optimal ne nécessite qu'un coefficient de protection $K_n = 2$ puisque déjà quatre fautes sont nécessaires pour l'attaque. Dans cet exemple, les résultats pour l'algorithme *rep-opt* sont meilleurs que *bloc-h* quel que soit le modèle de faute.

On peut remarquer que le programme P' est robuste pour toutes les expérimentations réalisées. Cela peut faire penser que l'exploration des chemins est complète pour tous les programmes. Cependant, on peut postuler que si l'exploration rate des chemins pour la génération des attaques sur P, il est probable que les chemins correspondant dans P' sont eux aussi manqués.

5.5 Protégeabilité et contre-mesure parfaite

Les différents algorithmes de placement présentés dans ce chapitre posent des conditions fortes sur les propriétés des contre-mesures du catalogue (adéquation et coefficient de pro-

10. Correspondant au coefficient de protection appliqué pour chaque IP moins 1 (un IP non protégé ayant un coefficient de protection de 1).

Expé			Algo.	$\sum$ des protections				Robuste
P	m	IPs		1 faute	2 fautes	3 fautes	4 fautes	
`vp2b`	BI	8	naïf	8	16	24	32	✓
			atk	**3**	8	12	16	✓
			min	**3**	8	12	16	✓
			bloc-h	**3**	**6**	**9**	**12**	✓
			rep-opt	**3**	**6**	**9**	**12**	✓
`fu1`	BI	42	naïf	42	84	126	168	✓
			atk	**0**	28	42	88	✓
			min	**0**	28	42	72	✓
			bloc-h	**0**	14	21	28	✓
			rep-opt	**0**	**7**	**14**	**21**	✓
	DL	2	naïf	2	4	6	8	✓
			atk	**1**	4	6	8	✓
			min	**1**	**2**	**3**	**4**	✓
			bloc-h	**1**	**2**	**3**	**4**	✓
			rep-opt	**1**	**2**	**3**	**4**	✓
	BI+DL	44	naïf	44	88	132	176	✓
			atk	**1**	32	60	96	✓
			min	**1**	32	60	80	✓
			bloc-h	**1**	16	24	32	✓
			rep-opt	**1**	**9**	**17**	**25**	✓
`memcmps3`	BI	12	naïf	12	24	36	48	✓
			atk	**0**	**0**	**0**	16	✓
			min	**0**	**0**	**0**	16	✓
			bloc-h	**0**	**0**	**0**	4	✓
			rep-opt	**0**	**0**	**0**	**1**	✓
	DL	15	naïf	15	30	45	60	✓
			atk	**1**	6	15	32	✓
			min	**1**	6	15	32	✓
			bloc-h	**1**	4	6	8	✓
			rep-opt	**1**	**3**	**4**	**7**	✓
	BI+DL	27	naïf	27	54	81	108	✓
			atk	**1**	8	24	56	✓
			min	**1**	8	24	56	✓
			bloc-h	**1**	6	9	12	✓
			rep-opt	**1**	**3**	**4**	**9**	✓

TABLE 5.7 – Comparaison des différents algorithmes de placement

tection suffisant) lorsqu'on recherche une robustesse en n fautes. Cette section propose une classification des modèles de faute en fonction du type de protection qu'il est possible d'y appliquer.

On notera $\mathcal{M}$ l'ensemble des modèles de faute. Il existe des modèles pour lesquels il est possible de protéger un programme pour le rendre résistant en n fautes, on dit alors que le modèle de faute est *protégeable* (comme les modèles TI et DL). A l'inverse, il existe des modèles pour lesquels il n'est pas possible de se protéger en n fautes. Given et al [Given-Wilson 2017] ont montré l'existence de tels modèles en se ramenant à des machines de Turing. C'est le cas par exemple d'un modèle de remplacement d'instruction arbitraire, permettant de remplacer le programme en un autre de taille arbitraire. On divisera donc l'ensemble des modèles de faute en fonction de leur *protégeabilité* :

— $\mathcal{P}$ (*modèles Protégeables*) : $m \in \mathcal{P} \iff \exists C$ tel que $\forall$ *programme* $P, \forall n \in \mathbb{N}$, $T_s(C(P), m_n) = \emptyset$ [11].

— $\mathcal{N}$ (*modèles Non protégeables*) : $m \in \mathcal{P} \iff \nexists C$ tel que $\forall$ *programme* $P, \forall n \in \mathbb{N}$, $T_s(C(P), m_n) = \emptyset$

L'ensemble des modèles protégeables $\mathcal{P}$ peut être divisé en deux sous-ensembles. Les modèles pour lesquels il existe une Contre-mesure Locale Pondérée et Adaptée (CLP-A) sont dits *localement protégeables*, notés $\mathcal{L}$ et ceux pour lesquels il n'en existe pas sont appelés *globalement protégeables*, notés $\mathcal{G}$. Les modèles pour lesquels il existe une contre-mesure locale parfaite sont dits *parfaitement protégeables*, notés $\mathcal{F}$.

Dans le cas des modèles non protégeables ($\mathcal{N}$), on peut là encore diviser entre ceux pour lesquels il est possible de rendre les attaques plus difficiles à réaliser, qu'on appellera *modèles diluables* $\mathcal{D}$ et ceux pour lesquels cela n'est pas possible, appelés *modèles strictement non protégeables*, notés $\mathcal{S}$.

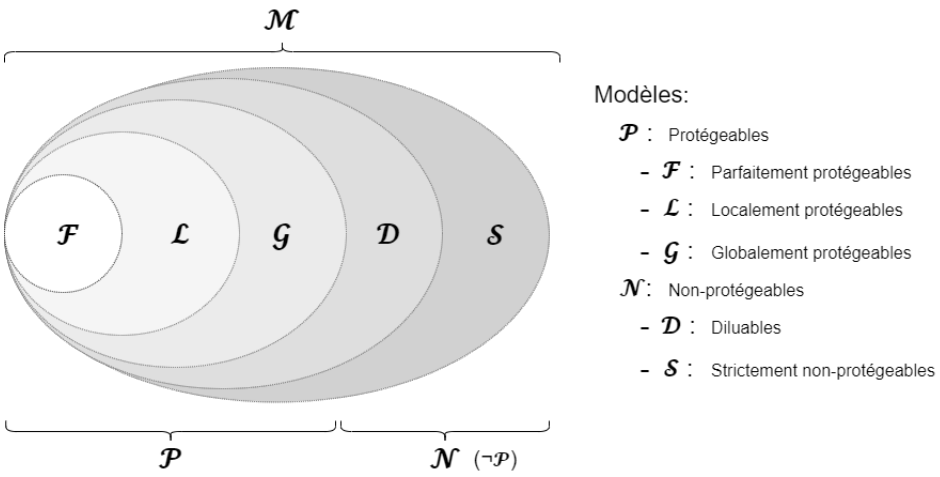

FIGURE 5.8 – Classification des modèles d'attaquant en fonction de leur *protegeabilité*

La figure 5.8 présente une classification des modèles d'attaquant en fonction de leur protégeabilité. Plusieurs questions restent en suspens concernant l'existence de certaines sous-classes. Les sections précédentes ont présenté des modèles dans $\mathcal{L}$ (*localement protégeables*), avec les exemples des modèles TI, DL et $TI + DL$, pour lesquelles les contre-mesures locales TM, LM et TLM ont été proposées. Les modèles *globalement protégeables* ($\mathcal{G}$) sont des modèles pour lesquels il existe une transformation du programme (non locale) permettant de

11. Avec $T_s(P, m_n)$ correspondant à l'ensemble des attaques réussies pour P jusqu'à n fautes.

rendre le programme robuste en n fautes. L'existence de cette classe de modèles est hypothétique puisque nous n'avons pas trouvé d'exemple, mais ne semble pas impossible pour autant. Il semble difficile de construire une contre-mesure parfaite pour les modèles TI et DL par exemple, mais l'existence de modèles dans $\mathcal{F}$ semble tout de même probable, si l'on s'intéresse à des modèles de faute simples. Formulé autrement : $\mathcal{L} = \mathcal{G} = \mathcal{P}$? (considérant que $\mathcal{G}$ n'inclut pas $\mathcal{L}$).

Une autre question survient avec cette classification : est-ce qu'un modèle peut changer de classe de *protégeabilité* si on s'intéresse qu'à un sous-ensemble des programmes possibles à protéger et/ou à un sous-ensemble des objectifs d'attaque. Il est aussi possible que cette classification n'ait de sens que si on exclut effectivement certains programmes et objectifs d'attaque qui sont toujours *protégeables* ou *non protégeables* quel que soit le modèle de faute [12]. Cette question n'a pas été explorée davantage. On peut aussi noter que ce n'est pas parce qu'un modèle de faute est non protégeable qu'il n'existe pas de protection contre ce type d'attaques en considérant un autre niveau de représentation.

Enfin, il est raisonnable de se demander à quel point les modèles *localement protégeables* ($\mathcal{L}$) sont suffisamment réalistes pour être utilisés pour le placement de contre-mesures en pratique. On peut arguer à cette question que, même s'il s'avère que les modèles dans $\mathcal{L}$ sont en réalité trop rares ou trop spécifiques, il reste envisageable que cette méthodologie puisse être généralisée pour le calcul du placement de contre-mesures pour des modèles dans $\mathcal{G}$.

Cette classification des modèles en fonction de leur protégeabilité contient donc encore des classes qui sont hypothétiques et des travaux futurs pourraient viser à préciser cela.

5.6 Conclusion et perspectives

Cette section présente une conclusion de ce chapitre et discute des perspectives possibles de ces travaux.

5.6.1 Conclusion

La table 5.8 présente une comparaison des algorithmes de placement présentés dans ce chapitre (identifiés en colonne "Algo") :

- *naïf* : protection systématique de tous les IPs avec $K_n = n + 1$.
- *atk* : protection des IPs des attaques avec $K_n = n + 1$.
- *min* : protection des IP des attaques minimales avec $K_n = n + 1$.
- *bloc-h* : algorithme par bloc avec heuristiques.
- *rep-opt* : algorithme optimal réparti.
- *bloc-opt* : algorithme par bloc optimal.
- *rep-h* : algorithme réparti avec heuristiques.

La colonne "Type" indique le type de placement de l'algorithme : *systématique*, par *bloc* ou *réparti*. Les colonnes "Garanties P'" indiquent les garanties pour le programme P' en ce qui concerne la robustesse du programme P' pour un catalogue total. L'optimalité du placement indique si le placement est optimal par rapport au type de placement considéré. La colonne "Complexité" indique la complexité de l'algorithme de placement (donc sans prendre en compte les différentes analyses d'attaques, de points chauds ou de redondance éventuelles), t correspondant au nombre d'attaques minimales obtenues par l'analyse d'attaque. Les

12. Par exemple si un modèle m est non protégeable dans le cas général, mais protégeable si on exclut le programme vide, peut-être faudrait-il le considérer comme protégeable et simplement exclure ce cas extrême de la définition.

colonnes "Analyses" indiquent les analyses de Lazart utilisées pour l'algorithme : analyse d'attaques (AA), analyse de redondance (Red) et analyse de points chauds (HS).

Algorithme	Type	Garanties P'		Complexité	Analyses requises		
		Robuste	Optimal		AA	Red	HS
naïf	syst.	✓	-	$O(t)$	✓	-	-
atk	syst.	✓	-	$O(t)$	✓	-	-
min	syst.	✓	-	$O(t)$	✓	✓	-
bloc-h	bloc	✓	-	$O(t)$	✓	✓	✓
rep-opt	réparti	✓	✓	NP-Complet[1]	✓	✓	-
bloc-opt	bloc	✓	✓	NP-Complet[1]	✓	✓	-
rep-h	réparti	✓	-	$O(t)$	✓	✓	✓

TABLE 5.8 – Comparaison des différentes approches de placement

Les expérimentations ont montré que les différents algorithmes parviennent à rendre un programme robuste en n fautes lorsque le catalogue est total. On peut constater la relation d'ordre attendue entre les poids de protections proposés par les différents algorithmes de placement : *naïf* $\geq$ *atk* $\geq$ *min* $\geq$ *bloc-h* $\geq$ *bloc-opt* $\geq$ *rep-h* $\geq$ *rep-opt*. Les algorithmes de placement systématiques produisent un grand nombre de protections superflues par rapport aux autres approches, mais les résultats dépendent grandement du programme considéré, *min* trouvant le placement optimal dans l'exemple `fu1` avec DL par exemple. De la même manière, *bloc-h* peut parfois donner les mêmes résultats que *rep-opt*, comme pour `vp2b`.

La complexité du placement optimal (section 5.3.3) rend son passage à l'échelle difficile pour des programmes contenant des combinaisons de placements possibles élevées. Néanmoins, les expérimentations tendent à montrer que l'ensemble des contraintes générées pour le problème d'optimisation linéaire est souvent simple, même dans le cas de programmes complexes. Cela est du aux attaques en une faute imposant un coefficient de protection maximal (c'est-à-dire de $n + 1$) à certains points d'injection.

L'analyse en isolation (section 5.2.2.1) des schémas de protection permet d'obtenir des garanties en fautes multiples concernant l'efficacité d'une contre-mesure sur un point d'injection, en fonction d'un modèle de faute. En combinaison avec l'exploration des chemins fautés, il est possible d'anticiper les comportements des programmes protégés en fonction des coefficients de protection appliqués. Dans le cas de catalogue non totaux, il est possible de prévoir si un placement va produire un programme P' robuste si un cas de sous-protection est rencontré (section 5.3.4), et dans ce cas de fournir les attaques de P'. Si l'exploration des chemins d'attaque n'est pas complète, on a tout de même la garantie d'avoir protégé certaines attaques sans en avoir introduit de nouvelles, ce qui permet à P' d'être *au moins aussi robuste* que P, sans garantie de robustesse en n fautes.

5.6.2 Perspectives

Différentes perspectives peuvent être envisagées pour les travaux présentés dans ce chapitre. Dans un premier temps, il serait intéressant d'étendre les exemples de programmes et de contre-mesures pour comparer plus en profondeur les diverses approches. Les algorithmes *bloc-opt* et *rep-h* pourraient être implémentés afin de pouvoir les comparer expérimentalement avec les autres algorithmes, mais ils n'offriront pas de meilleures garanties que *rep-opt*. Pour terminer sur le plan implémentation, l'algorithme de placement optimal utilise pour

1. L'ILP est NP-Complet dans le cas général mais les contraintes générées dans le cas des placements optimaux restent relativement simples.

le moment une intervention manuelle de l'utilisateur pour l'appel à un outil d'optimisation linéaire et automatiser ce processus serait un plus.

De la même manière, d'autres modèles de faute pourraient être considérés, notamment avec des modèles tels que le saut inconditionnel impliquant plusieurs points d'entrée et de sortie dans le schéma de protection, dans le cas de l'analyse de contre-mesures en isolation. Substituer les IPs protégés aux IPs non protégés dans l'exploration des chemins fautés nécessiterait de couvrir les comportements possibles de ces différents points d'entrée ou de sortie.

L'analyse en isolation de contre-mesures propageant des états internes [Oh 2002, Lalande 2014] nécessite de prendre en compte ces états comme entrées et sorties du schéma de protection. Cela nécessiterait potentiellement de diviser les scénarios de l'analyse en isolation en séparant les entrées liées au point d'injection (comme la condition du branchement) et les entrées correspondants aux états internes.

Les algorithmes de placement pourraient être étendus à des contre-mesures à granularité plus hautes que le point d'injection. Il s'agirait par exemple de sélectionner une protection au niveau d'un bloc ou d'une fonction plutôt qu'une protection de plusieurs points d'injection. Cependant il est difficile de déterminer dans quel cas chaque protection devrait être utilisée. Une analyse en isolation mais appliquée sur des structures plus grosses qu'un point d'injection pourrait aider à la définition de tels algorithmes.

Le placement de contre-mesures non locales fait aussi partie des pistes d'exploration. Cela implique que l'ordre dans lequel les protections sont appliquées est important pour le placement, les applications n'étant pas indépendantes. Il est possible de construire un algorithme de placement basé sur les mêmes heuristiques que *bloc-h*, en sélectionnant une contre-mesure adaptée pour protéger les points d'injection pour chaque attaque. Les garanties de robustesse sur un tel placement sont difficiles à définir, les protections ajoutées étant modifiées par l'application des contre-mesures suivantes. Une solution pourrait consister à étudier en isolation des schémas de protection combinés des différents schémas de protection du catalogue.

La classification théorique des modèles de faute en fonction de leur protégeabilité laisse encore quelques points en suspens, notamment en ce qui concerne l'existence de certaines classes de contre-mesures. Prouver l'existence ou la non-existence de ces groupes pourrait permettre d'aider à la définition de schémas de protection.

Les algorithmes de placement présentés dans ce chapitre se basent tous sur une analyse des chemins d'attaques non détectées validant l'objectif d'attaque, et visent à sélectionner les portions du programme à protéger. Le chapitre suivant s'intéresse à une approche inverse, visant à déterminer quelles portions de contre-mesures peuvent être retirées dans un programme protégé, en se basant sur une analyse recherchant les chemins d'attaques détectés.

Chapitre 6

Optimisation de contre-mesures à détecteurs

Ce chapitre présente la méthodologie d'analyse de contre-mesures présentée à la conférence FDTC [Boespflug 2020]. Cette méthodologie s'intéresse à vérifier que les protections ajoutées dans un programme sont effectivement utiles. Cela permet d'aider au placement de contre-mesures, notamment lorsque celles-ci sont placées par un outil automatique.

Dans un premier temps, la section 6.1 revient sur la problématique de l'optimisation de contre-mesures et présente la méthodologie proposée. Cette approche vise un type spécifique de contre-mesures basées sur des points de vérification de la forme `if(cond) detect();`, appelés *détecteurs*. La section 6.2 présente le fonctionnement de l'algorithme d'optimisation de contre-mesures qui repose sur un type particulier d'exécution utilisant des détecteurs non bloquants. La section 6.3 s'intéresse aux garanties apportées par l'algorithme d'optimisation de détecteurs. La section 6.4 présente l'implémentation de cette méthodologie dans Lazart et discute des résultats obtenus sur un ensemble d'expérimentations. Finalement, la section 6.5 conclut de ce chapitre.

Table des Matières

6.1 Méthodologie d'optimisation de détecteurs

Cette section présente la problématique de l'optimisation de contre-mesures décrite dans ce chapitre. Il s'agira dans un premier temps de présenter la structure des contre-mesures considérées (section 6.1.1), appelée *contre-mesure à détecteurs*. La section 6.1.2 présente la problématique du calcul de l'ensemble optimal de détecteurs dans un programme protégé.

6.1.1 Contre-mesure à détecteur

La méthodologie présentée ici repose sur une forme particulière de contre-mesures basées sur les tests, où des points de vérification (appelés *détecteur*) sont de la forme `if(condition) detect();`. La définition 6.1 présente la structure d'un programme protégé par une contre-mesure à détecteurs en trois parties : la contre-mesure (le corps et les détecteurs) et le programme non protégé (corps du programme).

Définition 6.1. *Dans un programme protégé par des contre-mesures logicielles basées sur les tests, le programme peut être décomposé en trois parties :*

- *Les détecteurs : des points de vérification sur l'état du programme qui visent à détecter un comportement anormal du programme. Les détecteurs sont de la forme* `if(condition) detect();`*, l'appel à* `detect` *correspondant à la détection de l'attaque.*
- *Le corps de contre-mesure : l'ensemble des variables et instructions supplémentaires ajoutées de manière à pouvoir effectuer les vérifications dans les détecteurs.*
- *Le corps de programme : correspondant au reste du programme (non lié à la contre-mesure).*

Définition 6.2. *On notera $\mathcal{D}(P)$ l'ensemble des détecteurs d'un programme P.*

Contre-mesure	Corps de contre-mesure	Détecteurs	Modèle de faute	Niveau
multiplication de tests	variables de conditions	tests ajoutés	inversion de test	LLVM
multiplication de load/store	instructions redondantes	vérification des valeurs des load/store	données (load/store)	LLVM binaire
redondance des données	variables redondantes opérations redondantes	points de comparaison	données et flôt de contrôle	
ST's SecSwift CF [de Ferrière 2019]	variables GSR, RTS calculs pour GSR, RTS identificateurs statiques de blocs	vérification (`GSR == ID`)	flôt de contrôle	LLVM
CNT [Heydemann 2019]	compteurs (et paramètres des fonctions) opérations sur les compteurs	macros de vérification (`CHECK_**`)	saut d'instructions n > 2	C

TABLE 6.1 – Correspondance des corps et détecteurs pour quelques contremesures

Un certain nombre de contre-mesures présentées dans la section 5.1 suivent ce schéma. Les contre-mesures telles que la multiplication de test (TM) en sont l'exemple naturel. Pour les schémas de duplication d'instructions, ou de redondance de données, les variables/registres ajoutées et les calculs supplémentaires font partie du corps de la contre-mesure, tandis que les tests redondants correspondent aux détecteurs. La contre-mesure *CNT* [Lalande 2014, Heydemann 2019] est aussi une contre-mesure à détecteurs. Les compteurs et leurs initialisations et incrémentations font partie du corps de contre-mesure et les détecteurs correspondent aux macros de vérification (telles que `CHECK_END_LOOP`).

La table 6.1.1 présente la décomposition en corps et détecteurs de certaines contre-mesures basées sur les tests de la littérature, identifiées par leur nom en première colonne. Les colonnes 2 et 3 indiquent respectivement ce qui correspond au corps de contre-mesure et aux détecteurs. La colonne "Modèle de faute" correspond aux modèles de faute pour laquelle la contre-mesure est destinée et la colonnes "Niveau" précise le niveau d'abstraction sur laquelle la contremesure s'applique classiquement (ou défini par les auteurs de l'article correspondant).

6.1.2 Problématique et ensemble optimal de détecteurs

L'objectif est de s'intéresser aux différentes variations du programme protégé P (définition 6.3), de manière à déterminer si certains détecteurs peuvent être retirés, sans introduire de nouvelles attaques (problématique 6.1).

Définition 6.3. *On notera $\mathcal{V}(P)$ l'ensemble des variations du programme P obtenues en retirant 0 à $|\mathcal{D}(P)|$ détecteurs (ce qui inclut P lui même).*

Définition 6.4. *Étant donné un programme P comportant les détecteurs $\mathcal{D}(P)$. On notera $P_{D_1,...,D_n}$ la variation du programme P conservant uniquement l'ensemble de détecteurs $\mathcal{D}(P_{D_1,...,D_n}) = \{D_1, ..., D_n\}$.*

Problématique 6.1. *Étant donné un programme protégé par un ensemble de détecteurs, et un modèle d'attaquant $\mathcal{M} = \{m, \phi\}$, avec m un modèle de faute et ϕ un objectif d'attaque, peut-on supprimer certains de ces points de vérification en gardant le même niveau de sécurité (c'est-à-dire sans introduire de nouvelles attaques) ?*

La problématique 6.1 revient à trouver l'ensemble de programmes dans $\mathcal{V}(P)$ qui n'introduisent pas de nouveaux chemins d'attaques réussies. Si comparer les attaques de programmes protégés différents est difficile dans le cas général, puisque deux programmes protégés peuvent avoir des structures et donc des chemins d'attaques très différents, les programmes dans $\mathcal{V}(P)$ ont tous une structure similaire, différenciés uniquement par la présence ou non de certains détecteurs [1]. Ainsi, la notion de "ne pas introduire de nouvelles attaques" correspond à s'assurer que chaque chemin d'attaque ayant pour préfixe un chemin d'attaque détectée dans P, reste en effet bloqué par un détecteur.

Si on choisit une fonction $\mathcal{W}$ associant un poids à un ensemble de détecteurs $\mathcal{D}_i$, on peut s'intéresser aux ensembles de détecteurs minimaux qui conservent le même niveau de sécurité (problématique 6.2). Si on s'intéresse à minimiser le nombre de détecteurs, avec un poids égal à 1 pour chaque détecteur du programme, on prendra $\mathcal{W}(\mathcal{D}_i) = |\mathcal{D}_i|$.

Problématique 6.2. *Étant donné un programme protégé par un ensemble de détecteurs, et un modèle d'attaquant $\mathcal{M} = \{m, \phi\}$, avec m un modèle de faute et ϕ un objectif d'attaque, peut-on trouver un ensemble de détecteurs minimal $\mathcal{D}_i$, tels que $P_{\mathcal{D}_i}$ n'introduit pas de nouvelles attaques par rapport à P, et $\mathcal{D}_i$ minimise une fonction $\mathcal{W}(\mathcal{D})$, fournie par l'utilisateur ?*

6.2 Optimisation de détecteurs

Cette section présente plus en détail le fonctionnement de la méthodologie d'optimisation de détecteurs. Cette méthodologie prend en entrée un programme P, protégé par un

1. Notez que les variations de P conservent toutes les portions du corps de contre-mesure, seuls les détecteurs peuvent être retirés. La section 6.2.5 discute du retrait des corps de contre-mesure.

ensemble de détecteurs $\mathcal{D}(P)$, un modèle d'attaquant $M = \{m, \phi\}$ (avec m le modèle de faute et ϕ l'objectif d'attaque), et recherche le ou les ensembles de détecteurs minimaux $\mathcal{D}_i$ (déterminés par une fonction de pondération $\mathcal{W}$) tels que les programmes P_{D_i} conservent le même niveau de sécurité que P. Cette analyse est paramétrée par une borne du nombre de fautes n, et autorise que les contre-mesures (détecteurs et corps de contre-mesure) soient attaquées.

L'idée derrière l'approche présentée ici est de s'intéresser aux chemins des attaques détectées ($T_c(P, M)$), en continuant l'exécution lorsqu'un détecteur se déclenche. Ainsi, on peut étudier le comportement du programme dans le cas où le détecteur n'est pas présent et chercher à savoir si un autre détecteur permettrait de détecter le chemin d'attaque. Si l'exploration des chemins est correcte et complète (définitions 3.5 et 3.6), alors les ensembles de détecteurs $\mathcal{D}_i$ calculés sont garantis comme étant minimaux [2].

La section 6.2.1 présente le principe de l'exécution avec détecteurs non bloquants, et l'ensemble des chemins d'attaques réussies et détectées. La section 6.2.2 montre que la recherche des ensembles optimaux de détecteurs, qui n'introduisent pas de nouvelles attaques, peut être ramenée aux ensembles qui couvrent chaque trace par au moins un détecteur. La section 6.2.3 présente l'étape de classification, une première optimisation sur la recherche de l'ensemble de détecteurs, et la section 6.2.4 montre l'étape de sélection permettant de calculer les ensembles minimaux. Finalement, la section 6.2.5 conclut cette section.

La méthodologie sera illustrée avec une version du programme `verify_pin`, appelée *verify_pin_2c*, et protégée avec une duplication de test systématique (TD). Ce programme sera étudié en prenant pour modèle de faute l'inversion de test (TI) et l'objectif d'attaque est de s'authentifier avec un PIN d'entrée incorrect. Les listings 6.1 et 6.2 (figure 6.1) présentent l'équivalent C du programme $vp2c + TD$, et correspondent respectivement aux fonctions `verify_pin` et `compare`. Les détecteurs sont indiqués en violet et le corps de contre-mesure en orange, celui-ci correspondant ici aux variables temporaires associées aux résultats des conditions des tests. Chaque détecteur est identifié par un entier permettant ainsi de différencier les déclenchements des détecteurs dans les traces. Cet identifiant est passé en paramètre de la fonction *detect*.

6.2.1 Exécution avec détecteurs non bloquants

Lorsqu'on s'intéresse à trouver les ensembles de détecteurs minimaux qui n'introduisent pas de nouvelles attaques, une première solution consiste à comparer chacune des variations du programme P. Soit n le cardinal de $\mathcal{D}(P)$, il existe $\mathcal{V}(P) = \sum_{k=1}^{n} \frac{n!}{k!(n-k)!} + 1$ variations du programme P correspondant aux combinaisons possibles des détecteurs dans P. La méthodologie présentée ici vise à calculer ces ensembles de détecteurs minimaux en une seule exploration des exécutions fautées. Pour cela, on étudie les exécutions du programme P, contenant l'ensemble des détecteurs, en continuant l'exécution lorsqu'un détecteur se déclenche, de manière à explorer les comportements dans les variations du programme où les détecteurs déclenchés ne seraient pas présents.

Définition 6.5. *On note $T^{nb}(P, M)$ l'ensemble des traces d'exécution obtenues avec une analyse avec détecteurs non bloquants, pour un modèle d'attaquant M.*

Lorsque le programme atteint un détecteur D_i, le déclenchement du détecteur est enregistré et l'exécution continue comme si le détecteur n'était pas présent. La figure 6.2 présente un exemple de traces avec détecteurs bloquants et de traces avec détecteurs non bloquants associées. Dans cet exemple, la trace bloquante 2 correspond aux deux traces

2. Une exception concernant certaines traces d'attaque à écarter sera précisée dans la section 6.3.4.

```
bool verify_pin(uint_t* user_pin) {
    if(bool c_1 = try_counter > 0) {
        if(!c_1)
            detect(0);

        if(bool c_2 = compare(user_pin,
     card_pin, PIN_SIZE) == true) {
            if(!c_2)
               detect(8);
            try_counter = 3;
            return true;
        } else {
            if(c_2)
                detect(9);
            try_counter--;
            return false;
        }
    } else
        if(c_1)
            detect(1);









    return false;
}
```

Listing 6.1 – Fonction `verify_pin`

```
bool compare(uint_t* a1, uint_t* a2,
     size_t size) {
    bool result = true; int i;
    bool c_1 = false;

    for(i = 0; c_1 = i < size; i++) {
        if(!c_1)
            detect(2);
        if(BOOL c_2 = a1[i] != a2[i]) {
            if(!c_2)
                detect(4);
            result = false;
        } else
            if(c_2)
                detect(5);
    }
    if(c_1)
        detect(3);

    if(bool c_3 = i != size) {
        if(!c_3)
            detect(6);
        detect(10);

    } else
        if(c_3)
           detect(7);

    return result;
}
```

Listing 6.2 – Fonction `compare`

```
bool verify_pin(uint_t* user_pin) {
    if(bool c_1 = try_counter > 0) {
        if(!c_1)
            detect(0);

        if(bool c_2 = compare(user_pin,
     card_pin, PIN_SIZE) == true) {
            if(!c_2)
               detect(8);
            try_counter = 3;
            return true;
        } else {
            if(c_2)
                detect(9);
            try_counter--;
            return false;
        }
    } else
        if(c_1)
            detect(1);









    return false;
}
```

Listing 6.3 – Résultats `verify_pin`

```
bool compare(uint_t* a1, uint_t* a2,
     size_t size) {
    bool result = true; int i;
    bool c_1 = false;

    for(i = 0; c_1 = i < size; i++) {
        if(!c_1)
            detect(2);
        if(BOOL c_2 = a1[i] != a2[i]) {
            if(!c_2)
                detect(4);
            result = false;
        } else
            if(c_2)
                detect(5);
    }
    if(c_1)
        detect(3);

    if(bool c_3 = i != size) {
        if(!c_3)
            detect(6);
        detect(10);

    } else
        if(c_3)
           detect(7);

    return result;
}
```

Listing 6.4 – Résultats `compare`

FIGURE 6.1 – Application systématique et application optimale pour $vp2c + TD$

non bloquantes 2.1 et 2.2, en continuant l'exécution après le déclenchement du détecteur D_A. Les cercles en bas indiquent le type de terminaison de la trace : attaque réussie T_s ou attaque détectée T_c pour les traces avec détecteurs bloquants ainsi que les attaques réussies et détectées T_{cs} pour les traces non bloquantes [3].

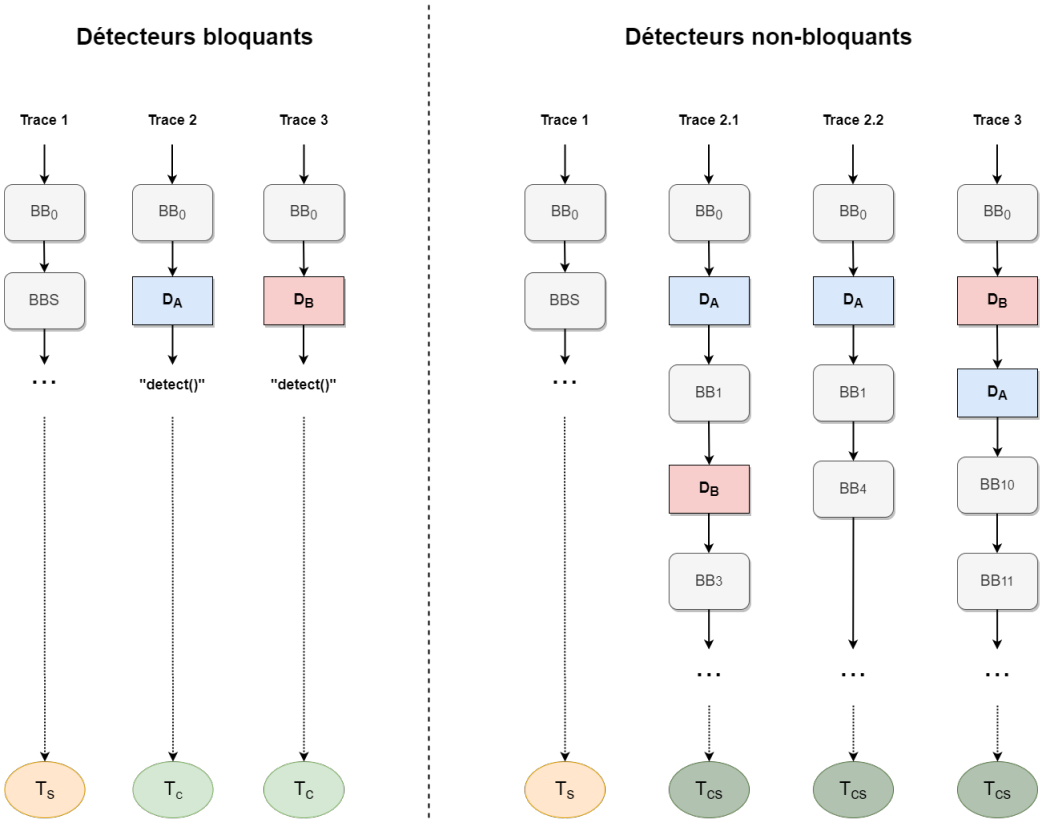

FIGURE 6.2 – Exemples de traces avec détecteurs bloquants (à gauche) et non bloquants (à droite)

6.2.2 Problématique de la couverture de $T_{cs}^{nb}(P, M)$

Dans la figure 6.2 précédente, on constate que si le détecteur D_B est retiré (trace 3), alors toutes les traces d'exécution non bloquantes sont couvertes (définition 6.6) par le détecteur D_A (à l'exception de la trace 1 qui n'est de toute façon bloquée par aucun détecteur). En revanche, si on retire le détecteur D_A, alors il existe un chemin supplémentaire (trace 2.2) qui ne sera pas bloqué, le programme P_{D_B} introduit donc au moins une nouvelle attaque par rapport aux programmes P_{D_A,D_B} et P_{D_A}. L'ensemble de détecteurs optimal sera donc dans cet exemple $\{D_A\}$.

Définition 6.6. *Soit $Det(t)$ l'ensemble des détecteurs déclenchés dans une trace d'exécution t. Un ensemble de détecteurs $\mathcal{D}_i$ couvre un ensemble de traces $\mathcal{T}$ si chaque trace de $\mathcal{T}$ contient le déclenchement d'au moins un détecteur de $\mathcal{D}_i$:*

$$Couvre(\mathcal{D}_i, \mathcal{T}) \equiv t \in \mathcal{T} \mid \exists d \in \mathcal{D}_i \mid d \in Det(t)$$

3. L'indice c indique qu'un détecteur au moins est déclenché (c pour *contre-mesure*) et l'indice s indique que l'objectif d'attaque est *satisfait*.

L'objectif étant de ne pas introduire de chemins d'attaques réussies, on s'intéresse aux attaques réussies et détectées (noté $T_{cs}(P, M)$). Dans le cadre d'une exécution non bloquante, une attaque détectée termine en contre-mesure. La notion d'attaque réussie pour des traces non bloquantes a du sens puisqu'on s'autorise à continuer après le déclenchement d'un détecteur afin de vérifier si l'objectif d'attaque aurait été validé dans le cas où les détecteurs ne seraient pas présents. Ainsi, les ensembles de détecteurs qui n'introduisent pas de nouvelles attaques sont ceux qui couvrent l'ensemble des traces d'attaques *détectées et réussies* d'une exécution non bloquante (noté $T^{nb}_{cs}(P, M)$). La problématique 6.2 revient donc à répondre à la problématique 6.3, qui est un problème d'optimisation [4].

Problématique 6.3. *Trouver les ensembles $\mathcal{D}_i$ de détecteurs minimaux tels que chaque trace d'attaques détectées et réussies $T^{nb}_{cs}(P, M)$, pour un programme P et un modèle d'attaquant M, soit couverte par au moins un détecteur de $\mathcal{D}_i$. Les ensembles minimaux sont ceux qui minimisent la fonction $\mathcal{W}(\mathcal{D})$ choisie par l'utilisateur.*

Pour la suite de cette section 6.2, on posera $\mathcal{T} = T^{nb}_{cs}(P, M)$. La recherche des ensembles $\mathcal{D}_i$ minimaux qui couvrent l'ensemble de trace $\mathcal{T}$ est réalisée en deux étapes :

— L'étape de classification (section 6.2.3) effectue une première passe sur les traces d'entrées afin de réduire l'espace de recherche.
— L'étape de sélection (section 6.2.4) consiste en la recherche des ensembles $\mathcal{D}_i$ optimaux sur cet espace réduit.

6.2.3 Classification des détecteurs

L'étape de classification consiste à faire un premier tri des traces d'entrées dans $\mathcal{T}$ pour trouver les détecteurs qui sont :

— *inactifs* : ne se déclenchent jamais dans $\mathcal{T}$ et peuvent donc être écartés des ensembles minimaux.
— *nécessaires* : se déclenchent au moins une fois seuls dans une trace d'exécution de $\mathcal{T}$ (sans qu'un autre détecteur ne soit déclenché), et doivent donc être conservés pour ne pas introduire de nouvelles attaques.

Tous les autres détecteurs sont dit *répétitifs* et c'est sur ceux-ci qu'il faut effectuer la recherche d'ensembles minimaux. L'étape de *classification* réduit l'espace d'exploration de la sélection des ensembles minimaux. En effet, seuls les détecteurs *répétitifs* $\mathcal{D}_R$ doivent être considérés et seules les traces contenant uniquement des détecteurs répétitifs $\mathcal{T}_R$ (définition 6.7) doivent être couvertes (les traces contenant au moins un détecteur nécessaire seront de toute façon couvertes).

Définition 6.7 (Ensemble des traces contenant uniquement des répétitifs)**.**

$$\mathcal{T}_R \equiv \forall t \in \mathcal{T} \mid Det(t) \cup \mathcal{D}_R$$

La table 6.2 présente la classification des détecteurs du programme $vp2c+TD$ pour une limite de fautes jusqu'à 4 [5]. Les détecteurs peuvent être nécessaires (N), répétitifs (R) ou inactifs (I).

4. La recherche de ces ensembles de détecteurs qui couvrent un ensemble de trace se rapproche de l'algorithme *bloc-opt* évoqué dans le chapitre 5.
5. Cet exemple n'est pas étudié avec des entrées symboliques mais avec des tableaux d'entrées fixés.

Détecteur	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}
1 faute	I	I	I	R	I	I	I	I	N	I	R
2 fautes	R	I	R	R	R	R	R	R	N	I	N
3 fautes	N	I	N	N	N	N	R	N	N	I	N
4 fautes	N	I	N	N	N	N	R	N	N	I	N

TABLE 6.2 – Classification des détecteurs pour $vp2c{+}TD$ en fonction du nombre d'inversion de test maximum

Les détecteurs D_1 et D_9 sont inactifs quel que soit le nombre de fautes, ce qui s'explique par le fait qu'ils protègent des branches qui n'ont pas d'impact sur l'objectif d'attaque (respectivement ligne 12 et 18 de la fonction `verify_pin` du listing 6.1). A l'inverse, le détecteur D_8 est nécessaire même en faute unique, étant donné que l'inversion de l'appel à la fonction `compare` permet de gagner en une faute.

6.2.4 Sélection des détecteurs

La seconde étape de la méthodologie consiste à chercher les ensembles de détecteurs répétitifs $\mathcal{D}_{Ri}$ qui, associés aux détecteurs nécessaires $\mathcal{D}_N$, couvrent l'ensemble des traces de $\mathcal{T}_R$. La recherche des $\mathcal{D}_{Ri}$ optimaux est ainsi un problème d'optimisation, dont l'implémentation sera précisée dans la section 6.4.

Dans le cas de l'exemple $vp2{+}TD$ en deux fautes, seules deux traces d'attaques détectées réussies contiennent uniquement des détecteurs répétitifs, notées a_1 et a_2. Les figures 6.3 et 6.4 présentent respectivement l'attaque a_1 en une faute et l'attaque a_2 en deux fautes, sans prendre en compte les transitions qui ne sont pas des fautes ou des détections. Deux ensembles de détecteurs répétitifs $\{D_3\}$ et $\{D_7\}$ permettent de couvrir $\mathcal{T}_R = \{a_1, a_2\}$, et doivent être associés aux détecteurs nécessaires $\mathcal{D}_N = \{D_8, D_{10}\}$ pour couvrir $\mathcal{T}$:

— $\mathcal{D}_1 = \{D_3, D_8, D_{10}\}$
— $\mathcal{D}_2 = \{D_7, D_8, D_{10}\}$

Trace a_1	...	Faute (bb2)	...	Dét 3	...	Dét 7	...

TABLE 6.3 – Trace de l'attaque réussie détectée a_1

Trace a_2	...	Faute (bb2)	...	Dét 3	...	Faute (bb3)	...	Dét 7	...

TABLE 6.4 – Trace de l'attaque réussie détectée a_2

En fonction de la pondération $\mathcal{W}$ choisie, $\mathcal{D}_1$ ou $\mathcal{D}_2$ seront sélectionnés. Pour une pondération égale pour chaque détecteur, $\mathcal{D}_1$ et $\mathcal{D}_2$ sont équivalents, mais le détecteur D_3 correspond à la duplication de la branche fausse du test `i < size` dans la boucle et pourrait par conséquent être considéré comme plus coûteux que le détecteur D_7 (duplication de D_{10}, hors de la boucle), privilégiant ainsi l'ensemble $\mathcal{D}_2$. Les listing 6.3 et 6.4 présentent les parties du code retirées (en rouge) et conservées (en vert) pour l'ensemble $\mathcal{D}_1$, pour respectivement les fonctions `verify_pin` et `compare`. Le corps de contre-mesure (ici chaque variable locale c_i associée à des détecteurs retirés) est également retiré.

6.2.5 Conclusion

La méthodologie de calcul de l'ensemble de détecteurs minimaux est ainsi découpée en deux étapes, la *classification* et la *sélection*, comme le montre la figure 6.3. Ainsi, une seule

exploration des exécutions est effectuée sur le programme P contenant tous les détecteurs. Cette méthodologie impose cependant que l'évaluation de la condition d'un détecteur n'ait pas d'effet de bord sur l'exécution de la suite du programme ainsi que sur l'objectif d'attaque étudié. La correction de cette approche, correspondant à la garantie qu'aucune nouvelle attaque n'est introduite par le retrait des détecteurs, est discutée en section 6.3.

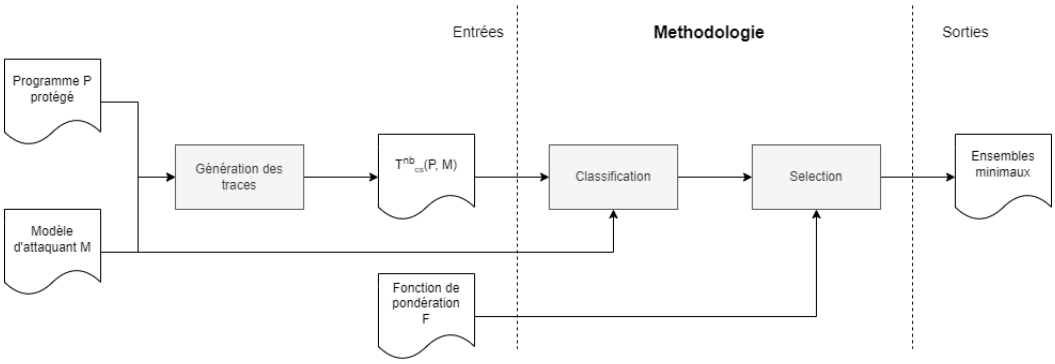

FIGURE 6.3 – Schéma de la méthodologie d'optimisation de détecteurs

La problématique de l'optimisation des contre-mesures a été abordée dans les chapitres précédents et la méthodologie présentée dans ce chapitre propose une solution dans le cas des contre-mesures à détecteurs, visant à répondre à la problématique 6.2. L'optimisation de détecteurs est une forme de comparaison de programmes protégés puisqu'il s'agit de comparer les différentes variations $\mathcal{V}(P)$. Elle peut aussi être utilisée en amont du processus de développement lors de l'application des contres-mesures pour aider au placement de ces contre-mesures.

La méthodologie présentée ici garantit que les détecteurs peuvent être retirés sans introduire de nouvelles attaques. Néanmoins, déterminer quelles portions du *corps de contre-mesure* peuvent être enlevées par rapport à l'ensemble de détecteurs calculé n'est pas trivial dans le cas général. La figure 6.4 présente les différents cas d'utilisation de la méthodologie en fonction de la méthode de placement utilisée pour le programme P : placement automatique (a) ou placement manuel (b). Le scénario (a) correspond au cas où le placement est effectué à l'aide d'un outil automatique, que ce soit par un outil dédié [Lalande 2014] ou un compilateur [Reis 2005, Proy 2017]. Il s'agit ici du scénario de l'exemple $vp2c + TD$ où la duplication de test peut être appliquée avec une granularité de l'ordre du point d'injection. Dans ce cas, le programme protégé P' est simplement régénéré à partir d'un ensemble de détecteurs calculé par l'algorithme d'optimisation :

1. Appliquer systématiquement les contre-mesures.
2. Calculer un ensemble optimal de détecteurs D_i.
3. Appliquer le placement avec les détecteurs de D_i uniquement.

Le scénario (b) décrit le cas où il n'est pas possible de ré-appliquer automatiquement la contre-mesure à l'aide de l'ensemble D_i. Cela implique qu'il faut calculer le corps de contre-mesure pouvant être retiré. Cette analyse n'est pas triviale puisqu'elle requiert de prendre en compte toutes les fautes possibles de manière à déterminer si une portion du corps de contre-mesure est nécessaire. Cette partie de la problématique n'est pas traitée dans ce manuscrit.

Un dernier cas d'utilisation peut être noté et correspond à un cas particulier du scénario a, où le placement automatique est un placement *optimal* (comme décrit dans le chapitre précédent). Dans ce cas, la méthodologie peut être utilisée pour vérifier l'optimalité du

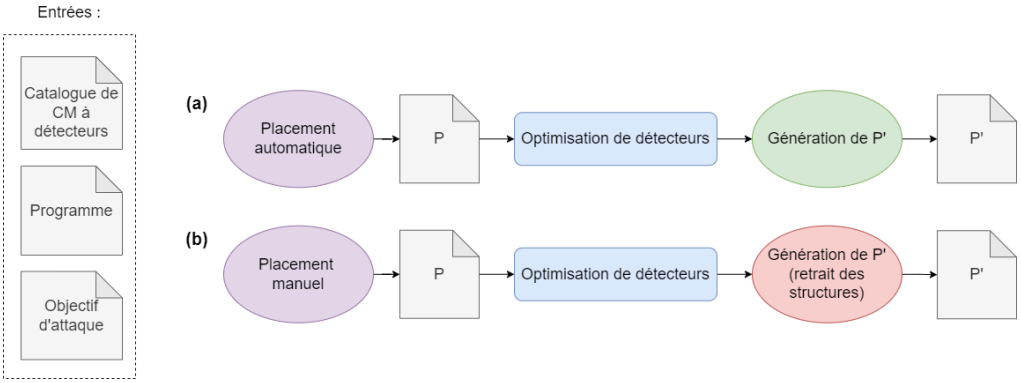

FIGURE 6.4 – Différentes applications de la méthodologie

placement, celui-ci dépendant de la complétude et correction de l'exploration des chemins d'attaques fautés [6].

6.3 Garanties de la méthode et protection des détecteurs

Cette section s'intéresse aux garanties de la méthodologie présentée dans la section précédente, en supposant l'exploration des chemins $T_{cs}^{nb}(P, M)$ complète et correcte. La section 6.3.1 revient sur les différentes classes d'exécution dans le cadre d'une exécution avec détecteurs non bloquants et la section 6.3.2 indique les relations autorisées entre ces classes pour préserver la robustesse du programme P'. La section 6.3.3 explique comment les traces d'attaques peuvent être mises en correspondance avec les traces d'attaques des différentes variations du programme P, permettant ainsi de vérifier que les attaques détectées dans P resteront détectées dans le programme P'. Enfin, la section 6.3.4 s'intéresse aux possibles faux positifs et présente une solution appelée « protection des détecteurs ».

6.3.1 Classification des exécutions pour $T^{nb}(P, M)$

Les exécutions d'un programme peuvent être caractérisées par :
— la validation ou non de l'objectif d'attaque ϕ.
— la terminaison (exécution terminée ou timeout/boucle infinie).
— la présence ou non de fautes.
— le déclenchement d'une contre-mesure (d'un détecteur).

Si on choisit d'abstraire la terminaison dans l'objectif d'attaque, ainsi que les cas d'erreurs (crash, division par zéro, accès mémoire invalide), cela permet de simplifier les catégories de traces en laissant l'utilisateur définir, dans l'objectif d'attaque, s'il considère un timeout ou une double désallocation avec `free` par exemple, comme une attaque réussie. C'est ce qui est fait dans Lazart, dans lequel l'utilisateur peut paramétrer les types de terminaison correspondant à une attaque réussie (voir section 3.4.2).

Classiquement, les exécutions ou les traces d'exécutions, pour un programme P et un modèle de faute M, sont partitionnées dans les ensembles suivants :
— $T_n(P, M)$: les exécutions nominales (sans fautes).
— $T_c(P, M)$: les attaques terminant en contre-mesure (détectées).

6. De la même manière, l'optimisation de détecteurs repose sur la complétude et la correction des chemins d'attaques détectées.

— $T_s(P, M)$: les attaques réussies (ϕ satisfait) et non détectées.
— $T_f(P, M)$: les attaques non réussies (ϕ non satisfait) et non détectées.

Dans le cas d'une analyse avec détecteurs non bloquants, l'exécution ne peut plus terminer en contre-mesure. Pour autant il est possible de différencier les traces contenant le déclenchement d'un détecteur et celles n'en contenant pas. De plus, il est possible d'évaluer la condition de succès de l'objectif d'attaque, l'exécution n'étant plus interrompue par les détecteurs. Pour une exécution non bloquante, les traces produites à partir de $T_c(P, M)$ peuvent se décomposer en :

— $T_{cs}(P, M)$: les attaques dans lesquelles au moins un détecteur a été déclenché et où l'objectif d'attaque ϕ est validé.
— $T_{cf}(P, M)$: les attaques dans lesquelles au moins un détecteur a été déclenché et où ϕ n'est pas satisfait.

Ensemble / Caractéristique	ϕ	fautes	détecteurs
$T_n(P, M)$	faux [7]	0	0 [8]
$T_s(P, M)$	vrai	1+	0
$T_{cs}(P, M)$	vrai	1+	1+
$T_{cf}(P, M)$	faux	1+	1+
$T_f(P, M)$	faux	1+	0

TABLE 6.5 – Caractéristiques des différents ensembles de traces d'exécution

La table 6.5 présente les caractéristiques (nombre de fautes, validation de l'objectif d'attaque et nombre de déclencheurs déclenchés) pour les cinq classes de traces d'exécution, ces ensembles de traces étant disjoints.

6.3.2 Préservation de la robustesse

Le paradoxe de dilution [Dureuil 2016a], énoncé dans le chapitre précédent, explique qu'il soit difficile de comparer les attaques de différentes versions protégées d'un programme. Les contre-mesures ajoutent des chemins d'exécution et de la surface d'attaque. S'il est possible d'associer chaque trace d'exécution d'un programme P à des traces d'un programme P' à l'aide d'une fonction δ, alors pour que P' soit au moins aussi robuste que P, il faut que chacune des traces qui ne sont pas des attaques réussies pour P restent inefficaces (attaques bloquées ou objectif d'attaque non réussi) pour P'. La table 6.6 présente les classes autorisées pour les traces de $T(P')$ en fonction des traces correspondantes (par une fonction δ) dans $T(P)$ pour une conservation de la robustesse.

Classe de $t \in T(P)$	Classes autorisées pour $t' \in \delta(t)$				
	$T_n(P', M)$	$T_s(P', M)$	$T_{cs}(P', M)$	$T_{cf}(P', M)$	$T_f(P', M)$
$T_n(P, M)$	✓	✗	✓	✓	✓
$T_s(P, M)$	✓	✓	✓	✓	✓
$T_{cs}(P, M)$	✓	✗	✓	✓	✓
$T_{cf}(P, M)$	✓	✗	✓	✓	✓
$T_f(P, M)$	✓	✗	✓	✓	✓

TABLE 6.6 – Classes autorisées pour les traces $t' \in f(t)$

7. On suppose que l'objectif d'attaque ne peut pas être validé dans une exécution nominale.
8. On suppose que les détecteurs ne se déclenchent pas dans une exécution nominale, la contre-mesure préservant le comportement observable du programme en l'absence de faute.

L'optimisation de détecteurs implique qu'un programme P' puisse avoir des attaques réussies (non détectées) nécessitant moins de fautes que leur attaque correspondante dans P, c'est-à-dire $|F(t')| < |F(t)|, t \in T_s(P, M), t' \in T_s(P', M), t' \in \delta(t)$ (avec $F(t)$ la liste ordonnée des fautes dans la trace t).

6.3.3 Correspondances entre les traces

Cette section vise à établir les correspondances entre les traces d'attaques des différentes variations de P et les traces d'exécution avec détecteurs non bloquants sur le programme P. Une trace d'exécution bloquante peut correspondre à plusieurs traces d'exécutions non bloquantes. Dans ce cas, la trace bloquante (définition 6.8) est un préfixe de chaque trace non bloquante correspondante (définition 6.9).

Définition 6.8. *Les traces d'attaque détectée avec détecteurs bloquants $T_c(P, M)$ sont de la forme $s_0...s_n d_i$, avec s_i les transitions nominales ou fautées et d_i le déclenchement d'un détecteur.*

Définition 6.9. *Les traces d'attaque détectée avec détecteurs non bloquants $T_c^{nb}(P, M)$ sont de la forme $s_0^1...s_{n_1}^1 d_i s_0^2...s_{n_2}^2 d_j...$ avec s_j^i les transitions nominales ou fautées et d_i les déclenchements de détecteurs.*

La figure 6.5 présente les correspondances entre les traces des différentes variations d'un programme P contenant deux détecteurs D_A et D_B, et les traces de l'exécution non bloquante $T^{nb}(P)$ (en haut). Pour chaque trace d'exécution, sa classe peut être attaque réussie (T_s) ou attaque détectée (T_c). Dans le cas de l'exécution non bloquante, les traces d'attaques réussies et détectées sont indiquées par T_{cs}. Dans cet exemple, les ensembles de détecteurs $\{D_B\}$ et $\{\}$ ne permettent pas de couvrir l'ensemble des attaques de T^{nb} et les programmes P_{D_B} et $P_{\{\}}$ introduisent de nouvelles attaques réussies qui ne sont pas détectées (respectivement les traces $B3$ et les traces 2 et 3).

L'exploration des exécutions non bloquantes du programme P contenant tous les détecteurs permet ainsi d'évaluer la robustesse de chaque variation de P. Si toutes les traces de $T_{cs}^{nb}P$ sont couvertes par (au moins) un détecteur d'un ensemble $\mathcal{D}_i$, alors les traces correspondante dans $P_{\mathcal{D}_i}$ seront détectées (dans $T_c(P_{\mathcal{D}_i})$). Cela implique cependant que les conditions des détecteurs non bloquants n'aient pas d'effet de bord sur le reste de l'exécution du programme (et l'objectif d'attaque).

6.3.4 Protection des détecteurs

L'analyse de l'ensemble $T_{cs}^{nb}(P, M)$ peut néanmoins générer des faux positifs, puisqu'il existe certaines exécutions qui peuvent fausser l'analyse, en classifiant un détecteur comme *nécessaire* de manière incorrecte. Il s'agit des fautes qui forcent le déclenchement d'un détecteur (alors que la condition de détection est fausse).

Le problème s'illustre facilement avec le modèle d'inversion de test, où la notion de *forcer le passage dans un détecteur* est plus claire. L'extrait de programme 6.5 contient un simple détecteur avec pour condition `c`. Si l'exploration des exécutions fautées est complète, et qu'il existe des exécutions où c est faux avec encore (au moins) une faute de disponible, alors il existe un ensemble d'exécutions dans lesquelles le test est inversé pour forcer l'entrée dans le détecteur.

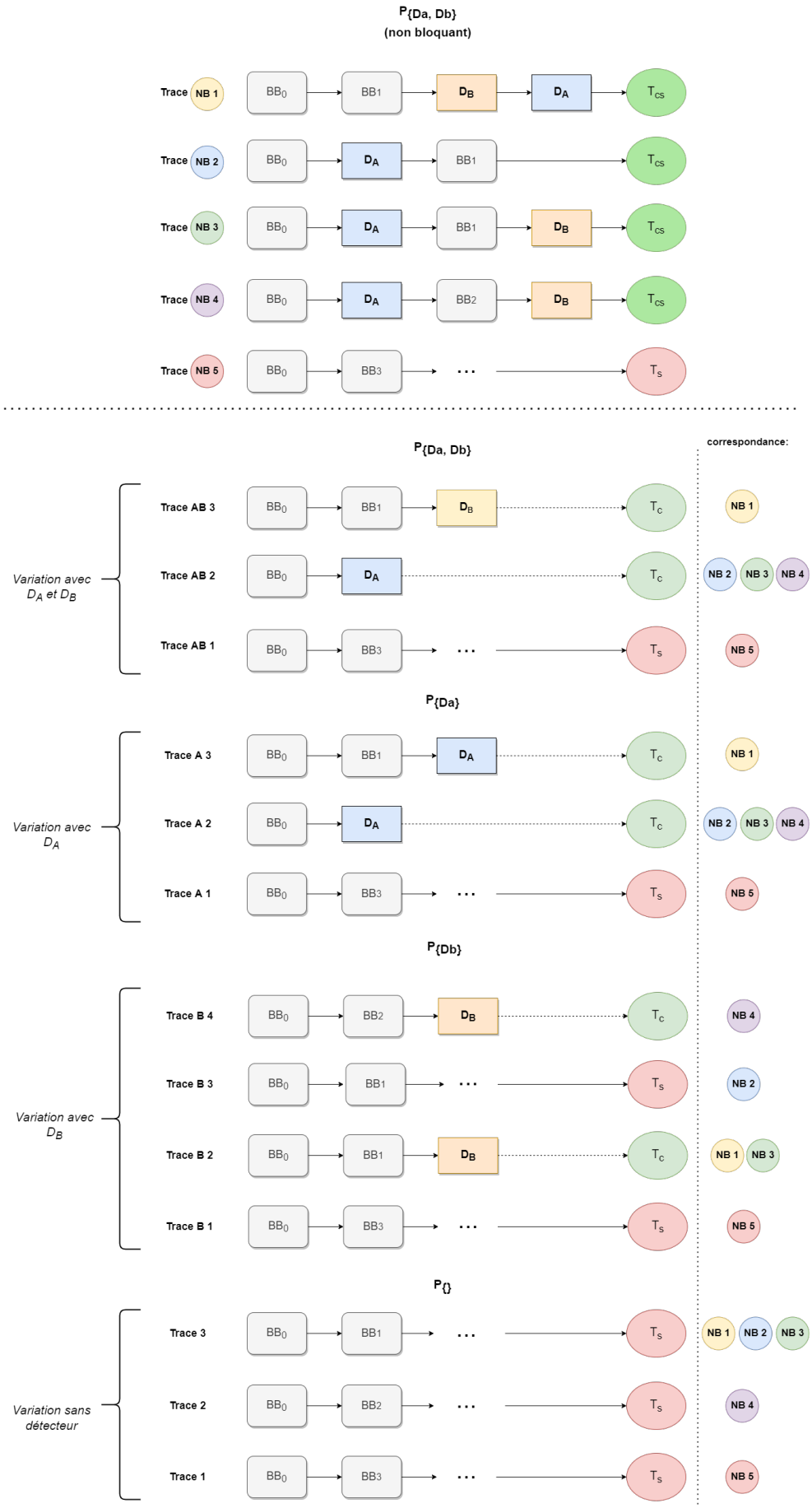

FIGURE 6.5 – Exemple de correspondance entre les traces des différentes variations $\mathcal{V}(P)$

```
...

if(c) {
countermeasure("X");
}

...
```

Listing 6.5 – Exemple de trace forcant le passage en contre-mesure

L'ensemble de ces traces donnant lieu à une sur-approximation de la méthodologie est nommé $T_b^{nb}(P, M)$ et est contenu dans $T_{cs}(P, M)$. La recherche des ensembles minimaux de détecteurs doit donc s'effectuer sur $T_{cs}^{nb}(P, M)/T_b^{nb}(P, M)$. Trouver les traces $T_b^{nb}(P, M)$ à partir de $T_{cs}^{nb}(P, M)$ n'est pas trivial, mais une solution consiste à interdire les fautes forçant le passage vers la branche *vraie* des détecteurs au niveau de la génération des traces. C'est cette approche qui est implémentée dans Lazart.

6.4 Implémentation et expérimentations

Cette section présente les résultats obtenus pour la méthodologie décrite dans ce chapitre et son implémentation au sein de l'outil Lazart.

6.4.1 Implémentation dans Lazart

Cette section présente l'implémentation de la méthodologie d'optimisation de détecteurs au sein de Lazart. La section 6.4.1.1 revient sur l'implémentation des détecteurs non bloquants. Les sections 6.4.1.2 et 6.4.1.3 détaillent respectivement les étapes de classification et de sélection. Enfin, la section 6.4.1.4 discute de la problématique de la création de *super-détecteur* lors de l'application automatique de certaines contre-mesures.

6.4.1.1 Détecteurs non bloquants

Usuellement, les outils d'analyses considèrent la terminaison d'une exécution en contre-mesure comme un type de terminaison à part entière. Pour l'analyse de contre-mesures présentée dans ce chapitre, il est nécessaire de supporter un mode d'exécution dans lequel les détecteurs ne terminent pas l'exécution du programme, tout en gardant en mémoire les détecteurs déclenchés. Le déclenchement d'un détecteur D_i est traduit par un évènement de type spécifique qui est affiché sur la sortie pour être récupérée lors du rejeu (voir section 4.1.1).

En pratique, la fonction associée au déclenchement d'un détecteur bloquant correspond au code C présenté dans le listing 6.6. Le paramètre `_det_id` correspond à l'identifiant unique du détecteur déclenché, en tant que chaîne de caractères. L'exécution est arrêtée avec un appel à `klee_assume(false);`.

```
void _LZ__trigger_detector_stop(const char* _det_id)
{
    _LZ__detector_alarm_ = true;
    if(klee_is_replay())
        printf("\n[CM] triggered %s \n", _det_id);  // See wiki for print syntax.
    klee_assume(false); // Cut exploration.
}
```

Listing 6.6 – Fonction de déclenchement de détecteur bloquant

Le listing 6.7 correspond à la fonction pour un détecteur non bloquant, qui ne diffère que par l'absence de l'appel à `klee_assume(false);`. La variable globale `_LZ__detector_alarm_` permet de vérifier dans l'objectif d'attaque si au moins un détecteur a été déclenché. La question de savoir si une attaque est détectée peut également se faire en vérifiant si $Det(t) = \emptyset$ dans l'API Python.

```
1  void _LZ__trigger_detector(const char* _det_id)
2  {
3      _LZ__detector_alarm_ = true;
4      if(klee_is_replay())
5          printf("\n[CM] triggered %s \n", _det_id); // See wiki for print syntax.
6  }
7
```

Listing 6.7 – Fonction de déclenchement d'un détecteur non bloquant

Dans le cas d'une exécution symbolique complète et correcte, une analyse d'attaque avec des détecteurs bloquants ou non bloquants n'a pas d'impact sur les chemins d'attaques réussies obtenus. En effet, seuls des chemins d'attaques détectées peuvent être générés à partir d'un détecteur non bloquant. Le mode bloquant a l'avantage d'arrêter plus tôt l'exploration des attaques détectées et donc de réduire le temps de l'exploration. C'est pourquoi il s'agit du mode par défaut pour les analyses d'attaques, les détecteurs non bloquants étant réservés à l'optimisation de détecteurs.

6.4.1.2 Classification

La première étape de classification est effectuée sur l'ensemble des traces d'attaques réussies et détectées, dans une exécution non bloquante. La classification des détecteurs repose sur le calcul du *niveau de répétition minimal* $L_m[D_i]$ de chaque détecteur D_i sur l'ensemble des traces. Pour chaque trace, on calcule le *niveau de répétition* qui correspond au nombre de détecteurs *différents* déclenchés dans cette trace (c'est-à-dire sans prendre en compte les répétition d'un même détecteur au sein de la trace). $L_m[D_i]$ qui est initialisé à ∞ et est mis à jour à chaque trace pour chaque détecteur présent. Chaque détecteur est finalement classé en fonction de ce niveau de répétition minimal :

— Inactif : si $L_m[D_i] = \infty$ *(ne s'est jamais déclenché)*
— Nécessaire : si $L_m[D_i] = 1$
— Répétitif : si $L_m[D_i] \in]1; \infty[$

L'implémentation utilise une classification en deux étapes, permettant d'obtenir les résultats pour chaque nombre de fautes jusqu'à n fautes. Dans un premier temps, la *classification locale* est calculée, correspondant au calcul de la classe de chaque détecteur pour un nombre de fautes donné. Dans un second temps, la classification globale est calculée, en fonction de la classification locale pour les nombres de fautes inférieurs (correspondant à une relation d'ordre $inactif < repetitif < necessaire$:

— si un détecteur est *répétitif* en n fautes, il est *répétitif* ou *nécessaire* en m fautes pour $m \geq n$.
— si un détecteur est *nécessaire* en n fautes, il est *nécessaire* en m fautes pour $m \geq n$.

6.4.1.3 Étape de sélection

Pour rappel, l'étape de sélection vise à trouver les ensembles minimaux (selon la fonction de pondération $\mathcal{W}$) de détecteurs répétitifs (dans $\mathcal{D}_R$) qui couvrent l'ensemble des traces $\mathcal{T}_R$. Une première solution consiste à essayer toutes les combinaisons de détecteurs possibles, et de conserver uniquement les ensembles qui couvrent $\mathcal{T}_R$.

L'algorithme présenté dans le listing 6.8 présente le pseudo code de l'algorithme de sélection utilisé dans Lazart. Celui-ci vise à explorer les ensembles de détecteurs contenant un seul élément (ajoutés à la pile, lignes 4 à 6) et explorer progressivement les ensembles contenant un élément de plus. L'ensemble des solutions minimales (`bests`) est initialisé avec l'ensemble $\mathcal{D}_r$. L'objectif est d'arrêter l'exploration lorsqu'un ensemble est d'ores et déjà supérieur aux minimums courants [9] (lignes 11 et 12). Si un ensemble ne couvre pas $\mathcal{T}_R$ et qu'il n'a pas un poids supérieur aux ensembles minimaux déjà trouvés, alors les ensembles avec un détecteur de plus sont ajoutés à la liste des ensembles à explorer (lignes 22 à 25).

```
1  def selection(traces, repetitives):
2  bests = [repetitives] # Current best sets covering traces.
3
4  set_stack = [] # Stack of set to be checked, by total weight (W)
5  for det in repetitives:
6  set_stack += [det] # Initialize with singletons.
7
8  while set_stack not empty:
9  curr = set_stack.pop()
10
11 if W(curr) > W(bests[0]):
12     continue # Stop exploration for super-sets
13
14 if Cover(curr, traces):
15     # Update best sets.
16     if W(curr) == W(bests[0]):
17         bests += curr
18     elif W(curr) < W(bests[0]):
19         bests = [curr]
20
21     continue # Stop exploration for super-sets
22 else: # Add super-sets to stack
23     for det in repetitives:
24         if det not in curr:
25             set_stacks.insert(curr + [det])
26
27 return bests
```

Listing 6.8 – Pseudo-code de l'algorithme de sélection

La complexité de cet algorithme est dépendante du nombre de détecteurs répétitifs $\mathcal{D}_R$ ainsi que du nombre de traces dans $\mathcal{T}_R$. Dans le pire cas, tous les détecteurs sont répétitifs et toutes les variations de P sont explorées. Néanmoins, les expérimentations tendent à montrer que cette étape est souvent négligeable par rapport au reste de l'analyse (voir section 6.4.2.5). Dans le pire cas, la seule solution est l'ensemble $\mathcal{D}_R$ (qui couvre toujours l'ensemble $\mathcal{T}_R$ par définition). Par ailleurs, l'algorithme de placement *bloc-opt* (voir section 5.3.4) visant à obtenir la couverture d'un IP protégé par trace d'attaques pourrait être implémenté avec cet algorithme (en cherchant des ensembles d'IPs plutôt que des ensembles de détecteurs).

6.4.1.4 Application automatique et super-détecteurs

L'application de la duplication de test systématique sur un détecteur déjà existant mène à la création de structures plus complexes comme le montre la figure 6.6, ce qui est le cas

9. Par exemple : si $\{D_1\}$ couvre l'ensemble de traces, alors $\{D_1, D_2\}$ le couvre aussi. Cependant l'ensemble $\{D_1, D_2\}$ ne peut pas être une solution minimale puisqu'il a un poids plus élevé et il n'est pas nécessaire d'explorer les ensembles plus grands. En revanche, il est nécessaire d'explorer $\{D_2\}$ qui pourrait faire partie des solutions minimales.

dans $vp2c + TD$ sur la vérification du compteur de boucle déjà présente dans la fonction *compare*.

```
1  ...
2
3  if(cond) {
4      detect("D1");
5  }
6
7  ...
8
9
10
11
12
```

```
1  ...
2
3  if(bool c_tmp = cond) {
4      if(!c_tmp) {
5          detect("DT");
6      }
7      detect("D1");
8  }
9  else if (c_tmp) {
10     detect("DF");
11 }
12 ...
```

FIGURE 6.6 – Détecteur (à gauche) et super-détecteur formé par TD (à droite)

La structure formée par le doublement de la branche du détecteur original (ligne 3 à 8) est appelée *super-détecteur*. Celle-ci pose problème, puisque D_1 n'a plus la structure *if(cond) detect() ;* après l'application de la duplication de test. Dans ce cas, la duplication est redondante par rapport au test original (c'est-à-dire que D_1 sera toujours déclenché si D_T est déclenché).

L'implémentation dans Lazart lève un avertissement lorsque l'application locale d'une contre-mesure est susceptible de générer un super-détecteur [10]. La généralisation de la méthodologie à des structures telles que les super-détecteurs fait partie des perspectives de ces travaux.

6.4.2 Expérimentations

Cette section présente les résultats obtenus sur un ensemble de couples de programmes et de contre-mesures. La section 6.4.2.1 présente les différents programmes utilisés dans ces expérimentations, ainsi que les modèles d'attaquants considérés. La section 6.4.2.2 détaille les différentes contre-mesures qui ont été étudiées. La section 6.4.2.3 présente le nombre de détecteurs retirés pour chaque exemple de test. La section 6.4.2.4 s'intéresse à l'impact de l'objectif d'attaque sur les résultats et la section 6.4.2.5 présente les métriques de performance obtenus.

6.4.2.1 Programme d'exemple

Les expérimentations présentées dans cette section utilisent cinq programmes :
- *vp2c* : la version 2c du programme *verify_pin*.
- *aes* et *aes rk* : une implémentation de l'algorithme de chiffrement Advanced Encryption Standard (AES) et une implémentation de l'étape `addRoundKey`.
- *gc1* : une implémentation d'une fonction de génération de nonce *get_challenge*.
- $fu1$: la version 1 du programme *firmware_updater*.

Chaque programme est analysé avec le modèle d'inversion de test (TI). Les programmes *vp2c* et $fu0$ ont déjà été présentés dans le chapitre précédent et on utilisera les mêmes objectifs d'attaque. Le programme *gc1* contient plusieurs contre-mesures intégrées (pile

10. Notez qu'il serait possible de dupliquer le test sans créer de super-détecteur, en dupliquant le test en dehors du détecteur (sans l'imbriquer). Cependant, il s'agit là d'une autre contre-mesure que la duplication de test de Lazart.

cachée, compteur de boucle) avec des détecteurs. Le programme *aes* est une implémentation de l'algorithme AES et *aes rk* ne contient que l'étape *addRoundKey*. L'objectif d'attaque est ici de ne pas générer le bon chiffré.

6.4.2.2 Les contre-mesures

En plus des détecteurs éventuellement présents dans les programmes originaux, trois contre-mesures ajoutées systématiquement seront considérées :

— la duplication de test (TD) déjà présentée précédemment.
— la contre-mesure SecSwift Control Flow [de Ferrière 2019] ($SSCF$).
— la contre-mesure de compteurs d'instructions niveau C [Lalande 2014] (CNT).

SecSwift ControlFlow. La contre-mesure SecSwift ControlFlow (SSCF) vise la protection du flot de contrôle et repose sur un système de signature par XOR. La figure 6.7 présente la protection d'un branchement à l'aide de la contre-mesure SSCF. Chaque bloc de base se voit assigné un identifiant unique et les variables `RTS` (Register Transfert Signature) et `GSR` (General Signature Register) sont utilisées pour vérifier que la bonne branche a été sélectionnée. `GSR` étant égal à l'identifiant du bloc courant dans une exécution nominale. `RTS` est préparée (premier bloc) en fonction de la condition en lui associant l'identifiant du bloc courant XOR le bloc de destination. Lors de l'arrivée dans le bloc de destination, la variable `GSR` est xorée avec `RTS`. Une assertion vérifie que `GSR` est bien égal à l'identifiant attendu, et lève une erreur dans le cas contraire. Ces assertions correspondent aux détecteurs pour SSCF.

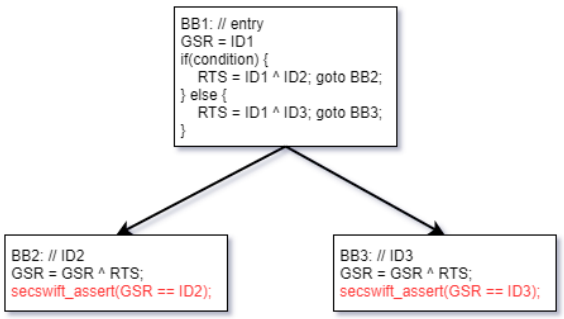

FIGURE 6.7 – Schéma de protection d'un branchement avec $SSCF$ [de Ferrière 2019]

Compteurs d'instructions. La contre-mesure CNT vise la protection contre le saut d'instruction niveau C en introduisant des compteurs incrémentés entre chaque instruction. Le listing 6.9 présente le début de la fonction *verify_pin* protégée avec CNT. Les macros `INCR` et `CHECK_INCR` effectuent l'incrémentation d'un compteur, la seconde ajoutant un détecteur pour vérifier que celui-ci vaut bien la valeur attendue. Diverses macros sont proposées dans [Lalande 2014] afin de protéger différentes structures (appel de fonction, boucle, condition...). La version étudiée ici est légèrement modifiée par rapport au papier original, certaines macros ajoutant un effet de bord dans la condition des détecteurs [11].

11. Par exemple, la macro `CHECK_INCR(cnt,val, det_id) cnt = (cnt == val ? cnt + 1 : detect(det_id));` dans sa version originale modifie le compteur dans le cas où une erreur est détectée. La vérification (détecteur) et la ré-initialisation du compteur ont donc été séparées.

```
#define INCR(cnt,val)  cnt = cnt + 1;
#define CHECK_INCR(cnt,val, cm_id) if(cnt != val) detect(cm_id); \
cnt = cnt + 1;
[...]


bool verify_pin(uint8_t* CNT_0_VP_1)
{
CHECK_INCR(*CNT_0_VP_1, CNT_INIT_VP + 0, 0LL)
g_authenticated = 0;
CHECK_INCR(*CNT_0_VP_1, CNT_INIT_VP + 1, 1LL)
DECL_INIT(CNT_0_byteArrayCompare_CALLNB_1, CNT_INIT_BAC)
CHECK_INCR(*CNT_0_VP_1, CNT_INIT_VP + 2, 2LL)
bool res = byteArrayCompare(user_pin, card_pin, PIN_SIZE, &CNT_0_compare_CALLNB_1);
[...]
```

Listing 6.9 – Pseudo-code de l'algorithme de sélection

6.4.2.3 Détecteurs retirés

La table 6.7 présente les résultats obtenus pour les différents exemples. La colonne "Programme" indique quel programme de base est utilisé et la colonne "Contre-mesure" indique la contre-mesure systématique ajoutée ("-" indiquant qu'aucune contre-mesure n'est ajoutée). La troisième colonne correspond au nombre global de détecteurs (après l'ajout de la contre-mesure). Les colonnes suivantes indiquent le pourcentage de détecteurs retirés après l'application de la méthodologie.

TABLE 6.7 – Pourcentage de détecteurs retirés pour différents programmes

Programme	Contre-mesure	$\mathcal{D}(P)$	1 faute	2 fautes	3 fautes
vp2c	TD	11	72%	63%	18%
vp2c	SSCF	13	92%	76%	23%
vp2c	CNT	31	93%	93%	32%
fu0	TD	14	0%	0%	0%
fu0	SSCF	24	12%	12%	8%
gc1	-	11	81%	72%	63%
gc1	TD	39	37%	34%	34%
gc1	SSCF	38	57%	28%	28%
aes rk	TD	2	50%	50%	0%
aes rk	SSCF	3	66%	33%	0%
aes c	TD	8	50%	50%	0%
aes c	SSCF	13	76%	61%	38%

On constate que le nombre de détecteurs retirés dépend fortement du programme et de la contre-mesure considérée. Naturellement, plus la limite de fautes est élevée, plus les détecteurs sont nécessaires. L'exemple $vp2c + CNT$ est un cas où la contre-mesure n'est pas prévue pour le modèle de faute considéré (CNT visant la protection contre les sauts d'instruction au niveau C), ce qui explique pourquoi autant de détecteurs peuvent être retirés (jusqu'à 2 fautes). Dans certains exemples (tels que $fu0 + TD$ ou bien $aes\ rk$ en 3 fautes), il n'est simplement pas possible de retirer de détecteurs.

6.4.2.4 Impact de l'objectif d'attaque

La table 6.8 présente les résultats obtenus sur l'exemple $vp2c$, en fonction de l'objectif d'attaque considéré : ϕ_{auth} (s'authentifier malgré un PIN incorrect), ϕ_{ptc} (ne pas décrémenter le compteur d'essais), leurs disjonction et conjonction logiques ($\phi_{auth \vee ptc}$ et $\phi_{auth \wedge ptc}$), ainsi que l'objectif d'attaque ϕ_{true} où toutes les traces sont considérées (hors cas d'erreurs).

TABLE 6.8 – Pourcentage de détecteurs retirés en fonction de l'objectif d'attaque ($vp + td$)

Objectif d'attaque	1 faute	2 fautes	3 fautes
ϕ_{auth}	83%	72%	18%
ϕ_{ptc}	72%	63%	9%
$\phi_{auth \vee ptc}$	83%	72%	18%
$\phi_{auth \wedge ptc}$	72%	63%	9%
ϕ_{true}	18%	9%	9%

Comme attendu, plus l'objectif d'attaque est général, moins il est possible de retirer de détecteurs. L'objectif d'attaque ϕ_{true} ne permet le retrait que de 18% des détecteurs en une faute, mais en trois fautes, les résultats sont équivalents à ceux de ϕ_{ptc} (et donc de $\phi_{auth \wedge ptc}$). Par ailleurs, on constate que ϕ_{auth} est contenu dans ϕ_{ptc} (ϕ_{auth} est équivalent à $\phi_{auth \vee ptc}$ et ϕ_{ptc} est équivalent à $\phi_{auth \wedge ptc}$).

6.4.2.5 Performances

La table 6.9 présente les métriques d'exécution et de performance des expérimentations précédentes. Les colonnes "Programme", "Contre-mesure" et "$\mathcal{D}(P)$" ont la même signification que pour la table 6.7. La colonne "Chemins" montre le nombre de chemins complétés et la colonne "Traces" correspond au nombres de traces (dans $T_{cs}^{nb}(P, M)$). La colonne "Temps (DSE)" indique le temps d'exécution de KLEE pour la génération des traces. La dernière colonne "Temps (Analyse)" indique le temps de l'analyse d'optimisation de détecteurs combinant les étapes de classification et de sélection.

On peut constater que l'exploration des chemins est toujours le facteur limitant pour tous les programmes de test. La durée de l'analyse est fortement liée au nombre de traces, la classification en faisant un parcours complet. L'étape de sélection est négligeable dans tous les exemples traités, dû au faible nombre de traces contenant uniquement des détecteurs répétitifs, même pour les cas avec un nombre de traces élevé.

6.5 Conclusion, limitations et perspectives

Les sections précédentes ont décrit une méthodologie d'analyse de contre-mesures à détecteur permettant de retirer les tests sans introduire de nouvelles attaques. L'exécution avec détecteurs non bloquants permet d'effectuer une seule exploration des traces d'exécutions, ce qui rend l'analyse réaliste même pour des programmes contenant un nombre important de détecteurs. Les expérimentations effectuées montrent que les résultats sont très dépendants des programmes et des contre-mesures considérés mais qu'il reste possible de retirer des détecteurs dans la grande majorité des exemples étudiés. La correction de cette approche est dépendante de celle de la méthode de génération des traces sous-jacente. L'implémentation dans Lazart avec l'exécution symbolique dépend donc de la complétude et de la correction des traces produites par KLEE.

Programme	Contre-mesure	$\mathcal{D}(P)$	Chemins	Traces	Temps (DSE)	Temps (Analyse)
vp	*TD*	11	7118	296	0:00:03	26ms
vp	*SSCF*	13	130 576	1005	0:01:54	89ms
vp	*CNT*	31	1 173 312	37 347	0:38:24	371ms
fu	*TD*	14	935 409	43 328	0:39:16	736ms
fu	*SSCF*	24	1 490 767	91 713	1:04:39	4s
*gc*1	-	11	4628	78	0:00:04	12ms
*gc*1	*TD*	39	102 169	10 281	0:01:35	1s
*gc*1	*SSCF*	38	1 048 354	58 367	0:31:45	2s
aesrk	*TD*	2	9 439	847	0:00:07	61ms
aesrk	*SSCF*	3	410 095	6 952	0:09:19	195ms
aes	*TD*	8	1 064 007	38 810	1:17:25	575ms
aes	*SSCF*	13	842 583	29 770	1:45:00	2s

TABLE 6.9 – Métriques de temps en 3 fautes

L'exécution avec détecteurs non bloquants impose certaines propriétés sur les détecteurs étudiés, ce qui rend l'analyse incomplète pour certaines contre-mesures (comme la contre-mesure *CNT* [Lalande 2014] qui a dû être modifiée en partie). La protection des détecteurs est également un problème qui est difficile à résoudre. Seule la protection simple pour l'inversion de test a été implémentée, une sur-approximation de la nécessité d'un détecteur est possible lorsqu'une trace forçant le passage en contre-mesure est détectée.

Retirer le corps de contre-mesure en plus des détecteurs n'est pas une question triviale. Dans le cas général, c'est-à-dire un programme P protégé contre n'importe quel type de contre-mesure à détecteurs, il est difficile de savoir quelles parties du corps des contre-mesures peuvent être retirées à partir de l'ensemble de détecteurs conservés (par exemple pour la contre-mesure SSCF où la variable GSR dépend des identifiants précédents). Des analyses statiques adaptées pour l'injection de fautes sont une piste d'exploration.

La méthodologie présentée dans ce chapitre autorise qu'une attaque $a' \in T(P')$ corresponde à une attaque $a \in T(P)$ mais avec un nombre de fautes nécessaires plus petit en raison des détecteurs qui ont été retirés. Il est possible d'adapter la méthodologie en imposant que certaines attaques ne soient pas simplifiée de la sorte, en considérant comme nécessaire tout détecteur correspondant aux points d'injection de ces attaques. Associer le détecteur correspondant à une faute est simple dans le cas de contre-mesures à granularité de l'ordre du point d'injection, comme TM ou SSCF par exemple, mais ne l'est pas dans le cas général.

Par ailleurs, les expérimentations présentées dans ce chapitre ont été réalisées sur une ancienne version de l'outil, l'optimisation de contre-mesures n'ayant pas été mise-à-jour sur la version 4 actuelle (présentée dans le chapitre 3 et 4). La mise à jour sur la version actuelle permettrait de profiter du modèle de faute sur les données (DL) et la combinaison de modèles. De la même manière cela permettrait d'étendre les expérimentations aux contre-mesures LM.

Chapitre 7

Conclusion et perspectives

Ce manuscrit a présenté trois contributions principales visant a répondre à différentes problématiques concernant l'analyse de robustesse d'un programme dans le contexte de l'injection de fautes et de l'analyse de contre-mesures. L'outil Lazart fournit un ensemble d'analyses pour la recherche d'attaques. Plusieurs algorithmes visant à aider au placement de contre-mesures logicielles ont été présentés. L'optimisation de détecteur propose une solution pour réduire l'ensemble des protections appliquées à un programme. Ces différentes contributions visent à prendre en compte les fautes multiples, et donc le fait que les protections ajoutées puissent être attaquées.

Cette conclusion revient sur les différentes contributions de ce manuscrit, ainsi que sur leurs limitations. Elle présente des perspectives de recherche permettant d'étendre ce travail et de répondre à certaines des problématiques soulevées.

Lazart. L'outil d'analyse de robustesse dans le cadre des fautes multiples Lazart (chapitres 3 et 4) a été développé et étendu au cours de cette thèse. Les différentes analyses proposées visent à aider l'utilisateur pour la recherche de chemins d'attaques et l'analyse de contre-mesures. La redondance et l'équivalence permettent de simplifier les résultats fournis à l'utilisateur en ne sélectionnant que les attaques les plus pertinentes. L'analyse de points chauds peut être utilisée dans le contexte de la réduction de l'espace de recherche mais aussi pour évaluer les contre-mesures, en appliquant l'analyse sur les chemins d'attaques détectées afin de déterminer les points d'injection qui sont le plus souvent bloqués. Parmi les perspectives des travaux présentés dans ce manuscrit, un certain nombre sont liées à l'extension de Lazart.

Extension des modèles de fautes. L'extension de l'outil vis-à-vis du support de nouveaux modèles de fautes serait profitable pour la recherche de chemins d'attaques ainsi que pour l'étude de contre-mesures visant des modèles non supportés par Lazart. Les modèles actuellement implémentés par l'outil permettent de couvrir un large spectre de modèles de fautes communément considérés au niveau logiciel. La mutation de donnée sur les `load` (DL) permet de simuler les attaques sur le flot de données tandis que le modèle de l'inversion de test (TI) et du saut arbitraire (JMP) permettent de représenter les attaques au niveau du flot de contrôle. D'autres modèles tels que le saut de fonction par exemple peuvent être simulés à l'aide de l'inversion de test. Cependant, le support de modèles de fautes plus spécifiques peut avoir des avantages en termes de performance et d'ergonomie pour l'utilisateur. Par exemple, le modèle `then/else` consistant à exécuter successivement les deux branches d'un test conditionnel [1] peut être simulé à l'aide du modèle de saut arbitraire mais requiert l'utilisation de l'instrumentation du programme, ce qui ne serait pas le cas avec un support « natif » via Wolverine. Les modèles liés aux pointeurs sont difficiles à prendre en compte pour Lazart, notamment parce que KLEE ne supporte pas les pointeurs symboliques. Cela

1. Pouvant correspondre à plus bas niveau à un NOP de l'instruction de branchement terminant le bloc then par exemple.

étant, il est envisageable de considérer des modèles de faute sur les données sans manipuler directement des adresses, par exemple en mutant l'identité des variables dans le code LLVM, permettant ainsi de simuler l'injection de fautes sur des adresses.

Extension de la bibliothèque de programmes d'exemples. Un grand nombre des programmes d'exemples présentés dans ce manuscrit sont issus de la collection FISSC [Dureuil 2016b]. Ceux-ci ont été mis à jour au cours de cette thèse de manière à les rendre plus modulaires afin de pouvoir étudier différentes propriétés (différents objectifs d'attaque, l'impact de l'inlining des fonctions, et l'ajout de certaines protections locales dans les exemples). Les programmes *firmware updater* et *memcmps* ont été développés afin de pouvoir étudier des combinaisons de modèles sur les données et le flot de contrôle. La recherche de nouveaux programmes de test serait un plus afin de pouvoir expérimenter sur des exemples, que ce soit pour leur complexité afin de tester le passage à l'échelle, leur représentativité en tant que code destiné à des appareils sécurisés ou encore pour mettre en évidence des attaques ou des modèles de fautes. L'extension des contre-mesures automatiques de Lazart (multiplication de tests, multiplication de loads et SSCF) est aussi une solution pour étendre les programmes d'exemple, en permettant de générer des versions protégées.

Placement de contre-mesure. Plusieurs algorithmes de placement de contre-mesures ont été proposés (chapitre 5). Ces algorithmes visent des classes particulières de contre-mesures, impliquant les notions de localité, pondération, adéquation et complétude. Ces algorithmes se basent sur un catalogue de contre-mesures, et leurs garanties dépendent des propriétés de ce catalogue. La protégeabilité d'un modèle de faute permet de classifier les modèles en fonction du type de protection possible.

Extension des contre-mesures supportées. Une piste d'amélioration consiste à étendre les algorithmes de placement pour pouvoir prendre en compte un ensemble plus large de contre-mesures. Plus particulièrement, les contre-mesures à granularité de l'ordre du bloc de base, d'une fonction ou bien encore d'une boucle pourraient être prises en compte. Les propriétés des contre-mesures à considérer pour pouvoir estimer quelle granularité de contre-mesure est nécessaire en fonction du contexte restent encore à étudier. Cette approche pourrait nécessiter de préciser à l'algorithme si un point d'injection appartient à une structure particulière (fonction, bloc de base...) du programme.

Optimisation de détecteurs Concernant l'optimisation de détecteurs (chapitre 6), la problématique du retrait des corps de contre-mesures à partir de l'ensemble $\mathcal{D}_i$ de détecteurs à conserver est délicate. Le support de détecteurs à la forme moins contrainte, comme les super-détecteurs évoqués précédemment (section 6.4.1.4), fait aussi partie des pistes d'exploration.

Optimisation de détecteurs symbolique. Une solution qui n'a pas encore été abordée précédemment consiste en l'utilisation d'une variable symbolique booléenne pour faire une disjonction des cas lors de l'arrivée sur un détecteur entre le code avec détecteur et le code sans détecteur. Cela correspond à ce qui est fait avec Lazart pour l'injection des fautes, mais appliqué aux différents détecteurs. Cela permettrait de supporter des détecteurs d'une forme plus complexe. Il est aussi envisageable d'étendre cette disjonction aux corps de contre-mesure, ce qui permettrait de les retirer au même titre que les détecteurs. Néanmoins, le passage à l'échelle d'une telle approche risque d'être encore plus difficile,

la disjonction due aux structures de la contre-mesure se combinant à celle des fautes injectées. De plus, certaines contre-mesures dont les paramètres dépendent des protections présentes dans le reste du programme (comme les valeurs attendues de `GSR` à chaque bloc pour SSCF), nécessiteraient de maintenir en parallèle les portions de code présentes dans le chemin courant.

Approche par sur-approximation. Les implémentations présentées pour le placement et l'optimisation de contre-mesures reposent toutes sur l'exécution concolique pour la génération des traces d'exécution d'entrée. L'utilisation d'une méthode de génération de traces par sur-approximation aurait du sens, cependant les expérimentations présentées pour la réduction de l'espace des points d'injection à l'aide de l'analyse statique (voir section 3.5) tendent à montrer que les propriétés sont difficiles à prouver lorsqu'on prend en compte les fautes multiples. Le « déroulement » des exécutions par l'exécution concolique offre des avantages importants en termes de précision des résultats. Cela étant, il est possible d'imaginer des optimisations préalables, à la fois pour le placement et l'optimisation de détecteurs, de manière à écarter une partie des points d'injection ou des détecteurs à considérer. Par exemple, l'optimisation de détecteurs pourrait profiter d'une analyse statique visant a minima à détecter les détecteurs inactifs afin de soulager l'exploration concolique.

Support de représentation plus bas-niveau. Une autre piste d'amélioration pour Lazart concerne le support de niveaux de représentation différents. Les approches de placement et d'optimisations de contre-mesures présentées précédemment sont, sur le principe, indépendantes du niveau de représentation considéré puisqu'elles se basent sur un ensemble de traces d'exécution. L'application à des niveaux de représentation différent introduit cependant d'autres problématiques. Pour l'optimisation de détecteurs au niveau binaire par exemple, si l'objectif d'attaque dépend des adresses des instructions (qui dépendent alors de la présence ou non de certains détecteurs), il est plus délicat de garantir qu'un détecteur n'a pas d'impact sur la suite de l'exécution du programme (pour l'exécution non-bloquante).

Annexe A

Documentation sur Lazart

Cette annexe présente différentes tables sur les paramètres et options de Lazart.

A.1 Paramètres d'une analyse dans l'API Python

La table A.1 décrit les paramètres d'une analyse dans l'YAML Python de Lazart. La colonne "Paramètre" indique le nom du paramètre. La colonne "Arg." indique si l'argument est positionnel (`argX`) ou par mot-clef (`kwargs`) et la colonne "Type" indique son type. La colonne "Défaut" correspond à la valeur par défaut de l'argument et "YAML" indique si cet argument a une correspondance dans le fichier de mutation YAML ou s'il est réservé à l'API Python.

Paramètre	Arg.	Type	Défaut	YAML	Description
intput_files	arg0	`List[str]`	req.	non	Liste des fichier sources.
attack_model	arg1	`str | dict`	req.	oui	Modèle d'attaquant, fourni soit comme le chemin vers le fichier YAML, soit sous la forme d'un dictionnaire décrivant les modèles appliqués à chaque fonction.
path	kwarg	`str`	généré	non	Chemin du dossier dans lequel les fichiers de l'analyse seront générés.
name	kwarg	`str`	""	non	Nom de l'analyse.
flag	kwarg	`int`	`Atk`	non	Informations sur le type d'analyse, utilisé pour des fonctions automatiques comme textttexecute ou `report`.
max_order	kwarg	`int`	2	non	Limite de faute pour l'analyse.
rename_bb	kwarg	`List[str]`	`"__mut__"`	oui	Description[1] des fonctions où appliquer de renommage des blocs de base.
add_trace	kwarg	`List[str]`	`"__mut__"`	oui	Description des fonctions où appliquer la trace du chemin de l'exécution.
compiler	kwarg	`str`	`"clang"`	non	Nom de la commande du compilateur
linker	kwarg	`str`	`"clang"`	non	Nom de la commande de l'éditeur des liens
disassembler	kwarg	`str`	`"clang"`	non	Nom de la commande du désassembleur
compiler_args	kwarg	`str`	""	non	Arguments supplémentaires pour l'appel de l'éditeur des liens
linker_args	kwarg	`str`	""	non	Arguments supplémentaires pour l'appel du compilateur
dis_args	kwarg	`str`	""	non	Arguments supplémentaires pour l'appel du désassembleur
wolverine_args	kwarg	`str`	""	non	Arguments supplémentaires pour l'appel de Wolverine.
klee_args	kwarg	`str`	`"emit-all-errors"`	non	Arguments pour l'appel de Klee.
countermeasures	kwarg	`List[dict]`	[]	oui	Liste des contre-mesures automatiques à appliquer sur le programme à analyser.

Table A.1 – Paramètres d'une analyse dans l'API Python.

A.2 Arguments des analyses

La table A.2 décrit les arguments et options des différentes analyses (traitements) de Lazart. La colonne "Argument" indique le nom de l'argument et la colonne "Opt" indique si celui-ci est optionnel. "Type" correspond au type de l'argument et "Défaut" à sa valeur

par défaute. Les colonnes "AA", "AAR", "HS", "DO" et "PL" indiquent si l'argument est disponible pour respectivement les analyses d'attaque, de redondance-équivalence, de points chauds, d'optimisation de détecteur et de placement.

Argument	Opt	Type	Défaut	AA	AAR	HS	DO	PL
analysis	-	`(Trace) => bool`	`lambda t: t.satisfies()`	✓	✓	✓	✓	✓
s_fct	✓	`(Trace) => bool`	`lambda t: t.satisfies()`	✓	✓	✓	✓	-
quiet	✓	`bool`	`false`	✓	✓	✓	✓	✓
lazy	✓	`bool`	`false`	-	✓	-	-	-
eq_rule	✓	`(Trace, Trace) => bool`	Equivalence	-	✓	-	-	-
red_rule	✓	`(Trace, Trace) => bool`	Préfixe	-	✓	-	-	-
do_red	✓	`bool`	`true`	-	✓	-	-	-
do_eq	✓	`bool`	`true`	-	✓	-	-	-
ccp_list	✓	`[str]`	all	-	-	-	✓	-
weight_fct	✓	`([str]) => int`	`lambda dr: len(dr)`	-	-	-	✓	-

TABLE A.2 – Arguments des analyses

A.3 Defines macros de Lazart

La table A.3 donne la liste des macros définies de Lazart lors de la compilation du programme, qui dépendent du type d'analyse et des options sélectionnées. Les colonnes "Macro" et "Type" correspondent respectivement à l'identifiant et au type de la macro. La colonne "Définition" précise si la macro est définie par Lazart en fonction de l'analyse ou précisée par l'utilisateur.

Macro	Type	Définition	Description
`LAZART`	define	lazart	Macro définie dans toute analyse avec Lazart.
`_LZ__VERSION`	string	lazart	Détermine la version actuelle de Lazart core.

`_LZ__VERSION_MAJOR`
`_LZ__VERSION_MINOR`
`_LZ__VERSION_PATCH`
	int	lazart	Valeur entière correspondant au numéro de version (major.minor.patch) de l'outil.

`_LZ__ATTACKS`	define	lazart	Macro définie si l'analyse est une analyse d'attaques.
`_LZ__CPPO`	define	lazart	Macro définie si l'analyse est une analyse d'optimisation de contre-mesures.
`_LZ__PLACEMENT`	define	lazart	Macro définie si l'analyse est une analyse de placement de contre-mesures.
`_LZ__NO_STD`	define	utilisateur	Si cette macro est définie, l'API C de Lazart est implémentée sans utiliser la bibliothèque standard C.
`_LZ__MUT_VALUE`	define	utilisateur	Active la récupération des valeurs injectées pour le modèle de mutation de données.
`_LZ__CM_USE_EXIT`	define	utilisateur	Si cette macro est définie, les détecteurs stoppant sont implémentés avec `exit` plutôt que `klee_assume`.

TABLE A.3 – Macros définies dans Lazart

A.4 Fonctions d'instrumentation de Lazart

La table A.4 liste les différentes fonctions d'instrumentation supportées par Lazart. La colonne "Catégorie" précise le type de catégorie de la commande et "Commande" indique l'identifiant. La colonne "Args." correspond aux paramètres de la fonction ou de la macros et précise leurs types. La colonne "Étape" indique à quelle phase d'une analyse la fonction d'instrumentation est traitée par Lazart. Finalement la dernière colonne donne une description de la commande.

Catégorie	Commande	Args.	Étape	Description
Utilitaire	`_LZ__RENAME_BB`	(str)	Pre-traitement	Renomme le bloc de base courant avec la chaîne de caractère spécifiée.
	`_LZ__MAX_DEPTH`	(int, expr)	DSE	Effectue une action lorsqu'un compteur local atteint une certaine valeur. Généralement utilisé de manière à limiter l'exécution d'une boucle ou une récursion. Prend en argument la limite de profondeur l'expression à exécuter.
Contrôles des modèles	`_LZ__DISABLE_BB` `_LZ__DISABLE_FUNCTION`	()	Mutation	Désactive toute mutation dans le bloc de base ou la fonction.
	`_LZ__DISABLE_MODEL` `_LZ__ENABLE_MODEL`	(str)	Mutation	Désactive / réactive un modèle de faute nommé à partir de ce point de la mutation.
	`_LZ__DISABLE_MODELS` `_LZ__ENABLE_MODELS`	()	Mutation	Désactive / réactive tous les modèles de fautes à partir de ce point de l'exécution symbolique.
	`_LZ__RESET`	()	Mutation	Réinitialise tous les modèles activés et désactivés.
IP utilisateur	`_LZ__mut_ti`	(bool, str, str, str)	Mutation	Fonction de mutation pour l'inversion de test prenant en entrée la condition à fauter, l'identifiant du point d'injection et les noms des deux blocs de base cibles.
	`_LZ__mut_ti_true` `_LZ__mut_ti_false`	(bool, str, str)	Mutation	Fonction de mutation pour l'inversion de test où seule l'une des branche peut être fautée.
	`_LZ__mut_dl_fix_iN`	(intN, intN, str)	Mutation	Fonctions de mutation pour l'injection de donnée fixe prenant en entrée la valeur originale, la valeur fautée et l'identifiant du point d'injection.
	`_LZ__mut_dl_sym_iN`	(intN, str)	Mutation	Fonctions de mutation pour l'injection de donnée arbitraire non-contrainte prenant en entrée la valeur originale et l'identifiant du point d'injection.
	`_LZ__mut_dl_sym_pred_iN` `_LZ__mut_dl_sym_fct_iN`	(intN, str, (intN) -> bool) (intN, str, (intN) -> intN)	Mutation	Fonctions de mutation pour l'injection de donnée arbitraire contrainte prenant en entrée la valeur originale, l'identifiant du point d'injection ainsi que la fonction de contrainte ou de prédicat.
	`_LZ__mut_jump`	(label, str)	Mutation	Macro pour le saut paramétré par un label de saut et un identifiant de point d'injection.
Objectif d'attaque	`_LZ__ORACLE`	(bool)	DSE	Macro de définition de l'objectif d'attaque. `Correspond à klee_assume.`
	`_LZ__triggered`	() -> bool	DSE	Fonction permettant de déterminer si au moins une contre-mesure a été déclenchée sur l'exécution courante.
	`_LZ__SYM` `_LZ__SYM_N`	(ptr, size) (ptr, size, str)	DSE	Macros pour la définition de variables symbolique. Prend en paramètre la zone mémoire à rendre symbolique et eventuellement un nom (utilisé par Klee). `Correspond à klee_make_symbolic.`
	`_LZ__EVENT`	(str)	TP	Macro permettant d'ajouter un évènement utilisateur dans la trace, pouvant ensuite être trié dans le script Python. Prend en paramètre une chaîne de caractère de description de l'évènement.
Contre-mesures	`_LZ__CM`	(str)	TP	Macro de déclenchement d'un détecteur, utilise la version bloquante ou non-bloquante en fonction de l'analyse. Prend en paramètre l'identifiant du détecteur.
	`_LZ__trigger_detector` `_LZ__trigger_detector_stop`	(str)	TP	Fonctions de déclenchement de détecteur. Prend en paramètre l'identifiant du détecteur.

TABLE A.4 – Fonctions d'instrumentation dans Lazart

Annexe B

Programmes d'expérimentation

Cette annexe présente plusieurs programmes utilisés dans différentes expérimentation de ce manuscrit. Il s'agit de donner le code des programmes ainsi que les objectifs d'attaque qui sont étudiés. Le code des programmes a été légèrement modifié de manière à rendre les codes auto-suffisant et certaines macros, permettant la compilation conditionnelle de différentes versions protégées du programme, ont été retirée par soucis de simplicité.

La section B.1 présente le programme `verify_pin_2b`. La section B.2 décrit le programme `memcmps3`. La section B.3 présente le programme `firmware_updater_2`.

B.1 Programme `verify_pin_2b`

Le programme *verify_pin_2b* (listing B.1) est une version du programme `verify_pin` comprenant les booléens endurcis et la boucle en temps constant. Contrairement à la version `verify_pin_2` de la collection FISSC, il n'y a pas de contre-mesure visant à vérifier le compteur de boucle.

Ce programme a notamment été utilisé dans le chapitre 5 en tant que programme d'exemples pour les expérimentations (voir section 5.4). Il est analysé avec le modèles TI (ou BI).

```
1  #include "lazart.h"
2
3  typedef int sec_bool_;
4  typedef enum { sec_true = 0xAA,
5      sec_false = 0x55 } sec_bool_t;
6
7  #define BOOL sec_bool_t
8  #define TRUE sec_true
9  #define FALSE sec_false
10
11 #define PIN_SIZE 4
12 #define TRY_COUNT 3
13
14 uint8_t try_counter;
15 uint8_t card_pin[PIN_SIZE];
16
17 BOOL compare(uint8_t* a1, uint8_t* a2, size_t size)
18 {
19     int i;
20     bool diff = FALSE;
21     for (i = 0; i < size; i++) { // OT
22         if (a1[i] != a2[i]) { // 1T
23             diff = TRUE;
24         }
25         // 1F
26     } // OF
27     return !diff;
28 }
29
30
```

```
31 BOOL verify_pin(uint8_t* user_pin)
32 {
33     if (try_counter > 0) { // 2T
34         if (compare(card_pin, user_pin, PIN_SIZE) == TRUE) { // 3F
35             try_counter = TRY_COUNT;
36             return TRUE;
37         } else { // 3F
38             try_counter--;
39             return FALSE;
40         }
41     }
42     // 2F
43 
44     return FALSE;
45 }
46 
47 
48 void init(uint8_t* user_pin, uint8_t* card_pin)
49 {
50     try_counter = TRY_COUNT;
51     klee_make_symbolic(user_pin, PIN_SIZE, "user_pin");
52     klee_make_symbolic(card_pin, PIN_SIZE, "card_pin");
53 
54     int equal = (card_pin[0] == user_pin[0]) & (card_pin[1] == user_pin[1]) & (card_pin[2]
       == user_pin[2]) & (card_pin[3] == user_pin[3]);
55 
56     klee_assume(!equal);
57 }
58 
59 bool oracle(BOOL ret) {
60     return ret == TRUE & !_LZ__triggered();
61 }
62 
63 int main()
64 {
65     uint8_t user_pin[PIN_SIZE];
66 
67     init(user_pin, card_pin);
68 
69     bool ret = verify_pin(user_pin);
70     _LZ__ORACLE(oracle(ret));
71 
72     return 0;
73 }
```

Listing B.1 – Programme d'analyse pour `verify_pin_2b`

B.2 Programme `memcmps3`

Le programme `memcmps3` (listing B.2) correspond à une versions protégée de la fonction standard `memcmp`, utilisant des masques. La version 3 contient 4 appels à la fonction standard, contrairement à la version 2 (voir section 3.3.1) qui n'en contient que 3.

L'objectif d'attaque étudié correspond à retourner `TRUE`, avec des tableaux d'entrées différents. Le programme peut être analysé avec le modèle TI ou en mutation de donnée sur les variables `len` et `result`.

```
1 #include "lazart.h"
2 #include <stdio.h>
3 
4 #define TRUE 0x1234
```

```
#define FALSE 0x5678
#define MASK 0xABCD
#define MASK2 0xF4F4

int memcmps3(char* a, char* b, int len)
{
    int result = FALSE;
    if (!memcmp(a, b, len)) {
        result ^= FALSE ^ MASK; // result = MASK
        if (!memcmp(a, b, len)) {
            result ^= MASK2; // result = MASK ^ MASK2
            if (!memcmp(a, b, len)) {
                result ^= TRUE ^ MASK; // result = MASK2 ^ TRUE
                if (!memcmp(a, b, len))
                    result ^= MASK2; // result = TRUE
            }
        }
    }

    return result;
}

void initialize_sym(char s1[SIZE], char s2[SIZE])
{
    klee_make_symbolic(s1, SIZE * sizeof(char), "s1");
    klee_make_symbolic(s2, SIZE * sizeof(char), "s2");
    int equal = 0;
    for (unsigned int i = 0; i < SIZE; i++) {
        equal += s1[i] == s2[i];
    }
    klee_assume(equal != SIZE);
}

int main()
{
    char s1[SIZE], s2[SIZE];
    initialize_sym(s1, s2);

    int res;
    res = memcmps3(s1, s2, SIZE);

    _LZ__ORACLE(oracle(res));
    return 0;
}
```

Listing B.2 – Programme d'analyse pour `memcmps3`

B.3 Programme `firmware_updater` 2

Les programmes `firmware_updater` simulent le processus de chargement d'un micro-programme depuis le réseau et la mise à jour du code local. Le listing B.3 correspond à la version 2 de ce programme, qui inclut une checksum permettant de vérifier l'intégrité à différents points du programme.

L'objectif d'attaque étudié consiste à réussir à corrompre le micro-programme, que ce soit en écrivant à la mauvaise adresse, ou en chargeant des données corrompues. L'intégrité du micro-programme effectivement chargé en mémoire est vérifiée dans l'objectif d'attaque.

```
#include "lazart.h"
#include "stdbool.h"
#include "stdint.h"
```

```c
#include <stdio.h>
#include <stdlib.h>


#define PAGE_NUMBER 4 // number of pages in a firmware
#define SIZEP 3 // number of bytes in a page
#define LOAD_ADDRESS 0x0F // default load address value for the new firmware
#define BAD_ADDRESS 0x0 // represents any corrupted load address ...

#define NW_CANARY() 3
#define NW_SIZE_BASE() (PAGE_NUMBER * SIZEP)
#ifndef GLOBAL_CHECKSUM
#define NW_SIZE_GCS() 0
#else
#define NW_SIZE_GCS() 1
#endif
#ifndef NO_CHECKSUM
#define NW_SIZE_CS() (PAGE_NUMBER)
#else
#define NW_SIZE_CS() 0
#endif
#define NW_SIZE() (NW_SIZE_BASE() + NW_SIZE_CS() + NW_SIZE_GCS() + NW_CANARY())

#include <stdio.h>
#include <stdint.h>

typedef uint8_t byte_t;

typedef struct {
    byte_t t[SIZEP];
    byte_t checksum;
} page_t;


typedef struct {
    page_t pages[PAGE_NUMBER];
} firmware_t;

byte_t memory[NW_SIZE() * 2] = {0};

byte_t network[NW_SIZE()] = {
    0xDE, 0xAD, 0xAA,
#ifndef NO_CHECKSUM
    (0xDE ^ 0xAD ^ 0xAA),
#endif
    0xAD, 0xDA, 0xAD,
#ifndef NO_CHECKSUM
    (0xAD ^ 0xDA ^ 0xAD),
#endif
    0xDE, 0xAD, 0xAA,
#ifndef NO_CHECKSUM
    (0xDE ^ 0xAD ^ 0xAA),
#endif
    0xAD, 0xDA, 0xAD,
#ifndef NO_CHECKSUM
    (0xAD ^ 0xDA ^ 0xAD),
#endif
#ifdef GLOBAL_CHECKSUM
#error "Not implemented"
#endif
    0xF5, 0xF5, 0xF5
};

```

```
67  byte_t receiveData() {
68      static unsigned nwstate = 0;
69
70      if(nwstate >= (NW_SIZE()))
71          exit(1);
72      return network[nwstate++];
73  }
74
75  bool save_firmware(uint8_t* memory, firmware_t* firmware, unsigned adress)
76  {
77      for(unsigned i = 0; i < PAGE_NUMBER; ++i) {
78          page_t* page = &firmware->pages[i];
79          uint8_t cs = computeChecksum(page);
80          if(cs != page->checksum) {
81              _LZ__CM("fcs");
82          }
83
84          unsigned offset = adress + (i * SIZEP);
85          for(int k = 0; k < SIZEP; ++k) {
86              memory[offset + k] = page->t[k];
87          }
88      }
89
90      return true;
91  }
92
93  void init_firmware(firmware_t* firmware) {
94      for(int i = 0; i < PAGE_NUMBER; ++i) {
95          firmware->pages[i].checksum = 0;
96          for(int k = 0; k < SIZEP; ++k)
97              firmware->pages[i].t[k] = 0xFE;
98      }
99
100 }
101
102 unsigned oracle_oob = 0;
103 unsigned oracle_diff = 0;
104
105 void verify_oracles(uint8_t* memory) {
106     for(unsigned i = 0; i < PAGE_NUMBER; ++i) {
107         unsigned i_fw = LOAD_ADDRESS + (i * SIZEP); // [B B B B B B]
108         unsigned i_nw = (i * (SIZEP + 1)); // [B B B CS B B B CS]
109
110         for(int k = 0; k < SIZEP; ++k) {
111             if (memory[i_fw + k] != network[i_nw + k]) {
112                 oracle_diff++;
113             }
114         }
115     }
116 }
117
118 uint8_t computeChecksum(page_t* page_buffer) {
119     // Fixed size.
120     return page_buffer->t[0] ^ page_buffer->t[1] ^ page_buffer->t[2];
121 }
122
123 void firmware_updater() {
124     firmware_t firmware;
125     unsigned page_counter = 0;
126     init_firmware(&firmware);
127
128     // Lecture
129     int byte_counter = 0;
```

```
130     while(true) {
131         byte_t received = receiveData();
132         page_t* page_buffer = &firmware.pages[page_counter];
133         page_buffer->t[byte_counter] = received;
134         byte_counter = byte_counter + 1;
135 
136         if(byte_counter >= SIZEP) { // End of page
137             byte_t cs = receiveData(); // return expected checksum
138             // Compute checksum
139             page_buffer->checksum = computeChecksum(page_buffer);
140             if(cs != page_buffer->checksum) {
141                 _LZ__CM("ics");
142             }
143             page_counter++;
144             byte_counter = 0;
145 
146             if(page_counter >= PAGE_NUMBER) {
147                 break;
148             }
149             continue; // next page
150         }
151     }
152 
153 
154     // check that all pages have been (properly) transfered)
155     if (page_counter != PAGE_NUMBER) {
156         _LZ__CM("pc");
157     }
158     save_firmware(memory, &firmware, LOAD_ADDRESS);
159 }
160 
161 
162 bool oracle() {
163     return (oracle_diff > 0) | (oracle_oob > 0);
164 }
165 
166 
167 
168 bool valid() {
169     for(unsigned i = 0; i < PAGE_NUMBER; ++i) {
170         unsigned i_fw = LOAD_ADDRESS + (i * SIZEP); // [B B B B B B]
171 #ifndef NO_CHECKSUM
172         unsigned i_nw = (i * (SIZEP + 1)); // [B B B CS B B B CS]
173 #else
174         unsigned i_nw = i * SIZEP;
175 #endif
176         for(int k = 0; k < SIZEP; ++k) {
177             if (memory[i_fw + k] != network[i_nw + k]) {
178                 return false;
179             }
180         }
181     }
182     return true;
183 }
184 
185 bool oob() {
186     for(unsigned i = 0; i < LOAD_ADDRESS; ++i) {
187         if(memory[i] != 0) {
188             return true;
189         }
190     }
191     for(unsigned i = LOAD_ADDRESS + (SIZEP * PAGE_NUMBER); i < (NW_SIZE() * 2); ++i) {
192         if(memory[i] != 0)
```

```
193             return true;
194     }
195     return false;
196 }
197
198
199 int main()
200 {
201     firmware_updater();
202
203 #ifdef LAZART
204     _LZ__ORACLE(!valid() | oob());
205 #endif
206 }
```

Listing B.3 – Programme d'analyse pour firmware_updater_2

Annexe C

Annexes supplémentaires

C.1 Contraintes ILP pour le programme `memcmps3`

L'implémentation du placement optimal dans Lazart génère les contraintes d'un programme ILP dans le format de l'outil externe utilisé (voir section 5.4). Le listing C.1 correspond aux contraintes pour le modèle de faute DL en une faute et le listing C.2 corresponds aux contraintes générées pour le modèle de faute $TI + DL$ pour 3 fautes, le fichier de contraintes en quatre fautes comporte 136 lignes).

```
# Ips
var ip7 >= 1;
var ip6 >= 1;
var ip14 >= 1;

minimize z: ip7 + ip6 + ip14;
subject to c0: ip14 >= 3;
subject to c1: ip6 + ip7 >= 3;
subject to c2: ip6 + ip14 >= 3;
subject to c3: ip6 + ip14 >= 3;
```

Listing C.1 – Contraintes ILP pour le programme `memcmps3` (DL, 1 faute)

```
# Ips
var ip12F >= 1;
var ip20 >= 1;
var ip13 >= 1;
var ip8 >= 1;
var ip9F >= 1;
var ip11 >= 1;
var ip10 >= 1;
var ip9T >= 1;

minimize z: ip12F + ip20 + ip13 + ip8 + ip9F + ip11 + ip10 + ip9T;
subject to c0: ip20 >= 4;
subject to c1: ip9F + ip20 >= 4;
subject to c2: ip8 + ip20 >= 4;
subject to c3: ip8 + ip10 >= 4;
subject to c4: ip9F + ip10 >= 4;
subject to c5: ip8 + ip20 >= 4;
subject to c6: ip8 + ip11 + ip20 >= 4;
subject to c7: ip9F + ip11 + ip20 >= 4;
subject to c8: ip9F + ip11 + ip20 >= 4;
subject to c9: ip8 + ip10 + ip20 >= 4;
subject to c10: ip8 + ip10 + ip11 >= 4;
subject to c11: ip9F + ip10 + ip11 >= 4;
subject to c12: ip8 + ip9T + ip20 >= 4;
subject to c13: ip8 + ip10 + ip11 >= 4;
subject to c14: ip8 + ip12F + ip20 >= 4;
subject to c15: ip8 + ip9F + ip10 >= 4;
subject to c16: ip8 + ip11 + ip20 >= 4;
subject to c17: ip8 + ip9F + ip20 >= 4;
subject to c18: ip9F + ip10 + ip12F >= 4;
```

```
31 subject to c19: ip9F + ip10 + ip11 >= 4;
32 subject to c20: ip9F + ip12F + ip20 >= 4;
33 subject to c21: ip8 + ip10 + ip12F >= 4;
34 subject to c22: ip8 + ip12F + ip13 >= 4;
35 subject to c23: ip9F + ip12F + ip13 >= 4;
36 subject to c24: ip9F + ip10 + ip20 >= 4;
37 subject to c25: ip9F + ip11 + ip13 >= 4;
38 subject to c26: ip8 + ip11 + ip13 >= 4;
```

Listing C.2 – Contraintes ILP pour le programme `memcmps3` ($TI + DL$, 3 fautes)

Bibliographie

[Abadi 2009] Martín Abadi, Mihai Budiu, Úlfar Erlingsson et Jay Ligatti. *Control-flow integrity principles, implementations, and applications*. ACM Transactions on Information and System Security (TISSEC), vol. 13, no. 1, pages 1–40, 2009. (Cité en page 21.)

[Agoyan 2010] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson et Assia Tria. *When clocks fail : On critical paths and clock faults*. In International conference on smart card research and advanced applications, pages 182–193. Springer, 2010. (Cité en pages iii, iv et 14.)

[Ahrendt 2005] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager*et al.* *The key tool*. Software & Systems Modeling, vol. 4, no. 1, pages 32–54, 2005. (Cité en page 35.)

[Amy Brown 2011] Greg Wilson Amy Brown. *The Architecture of Open-Source Application*, 2011. (Cité en pages vii et 41.)

[ANSSI 2018] ANSSI. `https://github.com/wookey-project`, 2018. accessed august 2022. (Cité en page 75.)

[ANSSI 2020] ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales et Trusted Labs. *Inter-CESTI : Methodological and Technical Feedbacks on Hardware Devices Evaluations*. In SSTIC 2020, Symposium sur la sécurité des technologies de l'information et des communications, 2020. (Cité en pages 11, 40 et 107.)

[Arlat 1990] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, J-C Fabre, J-C Laprie, Eliane Martins et David Powell. *Fault injection for dependability validation : A methodology and some applications*. IEEE Transactions on software engineering, vol. 16, no. 2, pages 166–182, 1990. (Cité en page 29.)

[ARM 2010] ARM. *ARM® v7-M Architecture Reference Manual*, 2010. (Cité en page 19.)

[Asimov 1954] Isaac Asimov. The caves of steel, volume 2. Asimov, 1954. (Cité en page 13.)

[Asonov 2004] Dmitri Asonov et Rakesh Agrawal. *Keyboard acoustic emanations*. In IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004, pages 3–11. IEEE, 2004. (Cité en page 14.)

[Aumüller 2002] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter et J-P Seifert. *Fault attacks on RSA with CRT : Concrete results and practical countermeasures*. In International Workshop on Cryptographic Hardware and Embedded Systems, pages 260–275. Springer, 2002. (Cité en pages 1 et 71.)

[Balasch 2011] J. Balasch, B. Gierlichs et I. Verbauwhede. *An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs*. In 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 105–114, 2011. (Cité en pages 17 et 19.)

[Baldoni 2018] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu et Irene Finocchi. *A survey of symbolic execution techniques*. ACM Computing Surveys (CSUR), vol. 51, no. 3, pages 1–39, 2018. (Cité en pages 8 et 40.)

[Bar-El 2006] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall et C. Whelan. *The Sorcerer's Apprentice Guide to Fault Attacks*. Proceedings of the IEEE, vol. 94, no. 2, pages 370–382, F ?rier 2006. (Cité en pages iii, iv, 14, 17 et 24.)

[Barenghi 2012] Alessandro Barenghi, Luca Breveglieri, Israel Koren et David Naccache. *Fault injection attacks on cryptographic devices : Theory, practice, and countermeasures.* Proceedings of the IEEE, vol. 100, no. 11, pages 3056–3076, 2012. (Cité en pages 24 et 26.)

[Barrett 2011] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds et Cesare Tinelli. *CVC4.* In Ganesh Gopalakrishnan et Shaz Qadeer, editeurs, Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. (Cité en page 47.)

[Barrett 2016] Clark Barrett, Pascal Fontaine et Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB).* `www.SMT-LIB.org`, 2016. (Cité en page 45.)

[Barrett 2018] Clark Barrett et Cesare Tinelli. *Satisfiability Modulo Theories.* In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith et Roderick Bloem, editeurs, Handbook of Model Checking, pages 305–343. Springer International Publishing, Cham, 2018. (Cité en page 44.)

[Barry 2016] Thierno Barry, Damien Couroussé et Bruno Robisson. *Compilation of a Countermeasure Against Instruction-Skip Fault Attacks.* In Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC, Prague, Czech Republic, January 20, 2016, pages 1–6, 2016. (Cité en pages 22 et 25.)

[Barton 1990] James H. Barton, Edward W. Czeck, Zary Z Segall et Daniel P. Siewiorek. *Fault injection experiments using FIAT.* IEEE Transactions on Computers, vol. 39, no. 4, pages 575–582, 1990. (Cité en page 30.)

[Basak 2019] Chowdhury Animesh Basak et Raveendra Kumar Medicherla. *VeriFuzz : Program Aware Fuzzing : (Competition Contribution).* In Tools and Algorithms for the Construction and Analysis of Systems : 25 Years of TACAS : TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III 25, pages 244–249. Springer, 2019. (Cité en page 47.)

[Benso 1998] Alfredo Benso, Paolo Prinetto, Maurizio Rebaudengo et M Sonza Reorda. *EXFI : a low-cost fault injection system for embedded microprocessor-based boards.* ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 3, no. 4, pages 626–634, 1998. (Cité en pages 23, 31, 32 et 34.)

[Berthomé 2012] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky et J. Lalande. *High Level Model of Control Flow Attacks for Smart Card Functional Security.* In ARES 2012, pages 224–229. IEEE, 2012. (Cité en pages 18, 20, 22 et 23.)

[Biham 1997] Eli Biham et Adi Shamir. *Differential fault analysis of secret key cryptosystems.* In Annual international cryptology conference, pages 513–525. Springer, 1997. (Cité en pages 1, 14 et 15.)

[Binder 1975] Daniel Binder, Edward C Smith et AB Holman. *Satellite anomalies from galactic cosmic rays.* IEEE Transactions on Nuclear Science, vol. 22, no. 6, pages 2675–2680, 1975. (Cité en page 13.)

[Blömer 2003] Johannes Blömer, Martin Otto et Jean-Pierre Seifert. *A New CRT-RSA Algorithm Secure against Bellcore Attacks.* In 10th ACM conference on Computer and communications security, CCS '03, page 311–320, New York, NY, USA, 2003. Association for Computing Machinery. (Cité en page 20.)

[Boespflug 2020] Etienne Boespflug, Cristian Ene, Marie-Laure Potet et Laurent Mounier. *Countermeasures Optimization in Multiple Fault-Injection Context.* In 2020 Workshop on Fault Detection and Tolerance in Cryptography (FDTC), pages 26–34. FDTC, 2020. (Cité en pages 71 et 133.)

[Boneh 1997] Dan Boneh, Richard A DeMillo et Richard J Lipton. *On the importance of checking cryptographic protocols for faults.* In International conference on the theory and applications of cryptographic techniques, pages 37–51. Springer, 1997. (Cité en pages 1 et 14.)

[Boneh 2001a] Dan Boneh, Richard A DeMillo et Richard J Lipton. *On the importance of eliminating errors in cryptographic computations.* Journal of cryptology, vol. 14, no. 2, pages 101–119, 2001. (Cité en page 15.)

[Boneh 2001b] Dan Boneh, Richard A DeMillo et Richard J Lipton. *On the importance of eliminating errors in cryptographic computations.* Journal of cryptology, vol. 14, no. 2, pages 101–119, 2001. (Cité en page 71.)

[Bouffard 2011] Guillaume Bouffard, Julien Iguchi-Cartigny et Jean-Louis Lanet. *Combined Software and Hardware Attacks on the Java Card Control Flow.* In Emmanuel Prouff, editeur, Smart Card Research and Advanced Applications, pages 283–296, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. (Cité en page 22.)

[Boyer 1975] Robert S Boyer, Bernard Elspas et Karl N Levitt. *SELECT—a formal system for testing and debugging programs by symbolic execution.* ACM SigPlan Notices, vol. 10, no. 6, pages 234–245, 1975. (Cité en page 42.)

[Bradley 2006] Aaron R Bradley, Zohar Manna et Henny B Sipma. *What's decidable about arrays ?* In International Workshop on Verification, Model Checking, and Abstract Interpretation, pages 427–442. Springer, 2006. (Cité en page 6.)

[Breier 2019] Jakub Breier, Xiaolu Hou et Yang Liu. *On evaluating fault resilient encoding schemes in software.* IEEE Transactions on Dependable and Secure Computing, 2019. (Cité en pages 11, 22 et 23.)

[Bréjon 2020] Jean-Baptiste Bréjon. *Quantification de la securite des applications en presence d'attaques physiques et detection de chemins d'attaques.* PhD thesis, Sorbonne Université, Paris, France, 2020. (Cité en page 18.)

[Brummayer 2009] Robert Brummayer et Armin Biere. *Boolector : An efficient SMT solver for bit-vectors and arrays.* In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 174–177. Springer, 2009. (Cité en page 47.)

[Bucur 2011] Stefan Bucur, Vlad Ureche, Cristian Zamfir et George Candea. *Parallel symbolic execution for automated real-world software testing.* In Proceedings of the sixth conference on Computer systems, pages 183–198, 2011. (Cité en page 47.)

[Cadar 2005] Cristian Cadar et Dawson Engler. *Execution generated test cases : How to make systems code crash itself.* In International SPIN Workshop on Model Checking of Software, pages 2–23. Springer, 2005. (Cité en page 45.)

[Cadar 2008a] Cristian Cadar, Daniel Dunbar et Dawson R. Engler. *KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.* In Richard Draves et Robbert van Renesse, editeurs, 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, USA, Proceedings, pages 209–224. USENIX Association, 2008. (Cité en pages 8, 35, 45, 46, 47 et 74.)

[Cadar 2008b] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill et Dawson R Engler. *EXE : Automatically generating inputs of death.* ACM Transactions on

Information and System Security (TISSEC), vol. 12, no. 2, pages 1–38, 2008. (Cité en page 45.)

[Cadar 2013] Cristian Cadar et Koushik Sen. *Symbolic execution for software testing : three decades later.* Commun. ACM, vol. 56, no. 2, pages 82–90, F ?rier 2013. (Cité en pages 45 et 46.)

[Cadar 2020] Cristian Cadar et Martin Nowack. *KLEE symbolic execution engine in 2019.* International Journal on Software Tools for Technology Transfer, pages 1–4, 2020. (Cité en pages 46 et 47.)

[Carreira 1998] João Carreira, Henrique Madeira et João Gabriel Silva. *Xception : A technique for the experimental evaluation of dependability in modern computers.* Software Engineering, IEEE Transactions on, vol. 24, no. 2, pages 125–136, 1998. (Cité en page 32.)

[CEA 2008a] CEA. *Frama-C EVA webpage*, 2008. (Cité en page 8.)

[CEA 2008b] CEA. *Frama-C WP webpage*, 2008. (Cité en page 8.)

[Charyyev 2019] Batyr Charyyev, Ahmed Alhussen, Hemanta Sapkota, Eric Pouyoul, Mehmet Hadi Gunes et Engin Arslan. *Towards securing data transfers against silent data corruption.* In 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pages 262–271. IEEE, 2019. (Cité en page 14.)

[Chipounov 2012] Vitaly Chipounov, Volodymyr Kuznetsov et George Candea. *The S2E platform : Design, implementation, and applications.* ACM Transactions on Computer Systems (TOCS), vol. 30, no. 1, pages 1–49, 2012. (Cité en pages 46 et 47.)

[Choi 1992] Gwan S. Choi et Ravishankar K. Iyer. *FOCUS : An experimental environment for fault sensitivity analysis.* IEEE Transactions on Computers, vol. 41, no. 12, pages 1515–1526, 1992. (Cité en page 33.)

[Christofi 2013] Maria Christofi. *Preuves de sécurité outillées d'implémentation cryptographiques.* PhD thesis, Laboratoire PRiSM, Université de Versailles Saint Quentin-en-Yvelines, France, 2013. (Cité en page 27.)

[Clark 1995] Jeffrey A Clark et Dhiraj K Pradhan. *Fault injection : A method for validating computer-system dependability.* Computer, vol. 28, no. 6, pages 47–56, 1995. (Cité en page 14.)

[Clarke 1976] Lori A. Clarke. *A system to generate test data and symbolically execute programs.* IEEE Transactions on software engineering, no. 3, pages 215–222, 1976. (Cité en page 42.)

[Clavel 1996] Manuel Clavel, Steven Eker, Patrick Lincoln et José Meseguer. *Principles of maude.* Electronic Notes in Theoretical Computer Science, vol. 4, pages 65–89, 1996. (Cité en page 35.)

[Clavel 2002] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer et José F Quesada. *Maude : Specification and programming in rewriting logic.* Theoretical Computer Science, vol. 285, no. 2, pages 187–243, 2002. (Cité en page 35.)

[Colombier 2019] Brice Colombier, Alexandre Menu, Jean-Max DUTERTRE, Pierre-Alain Moëllic, Jean-Baptiste Rigaud et Jean-Luc Danger. *Laser-induced Single-bit Faults in Flash Memory : Instructions Corruption on a 32-bit Microcontroller.* In 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pages 1–10, McLean, United States, Mai 2019. IEEE. (Cité en pages iii, iv, 14 et 19.)

[Cousot 1977] Patrick Cousot et Radhia Cousot. *Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252, 1977. (Cité en page 8.)

[Cousot 2014] Patrick Cousot et Radhia Cousot. *Abstract interpretation : past, present and future.* In Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pages 1–10, 2014. (Cité en page 8.)

[de Ferrière 2019] François de Ferrière. *A compiler approach to Cyber-Security.* 2019 European LLVM developers' meeting, 2019. (Cité en pages viii, 25, 104, 105, 112, 134 et 150.)

[De Moura 2011] Leonardo De Moura et Nikolaj Bjørner. *Satisfiability modulo theories : introduction and applications.* Communications of the ACM, vol. 54, no. 9, pages 69–77, 2011. (Cité en pages 44 et 47.)

[Di 2016] Sheng Di et Franck Cappello. *Adaptive impact-driven detection of silent data corruption for HPC applications.* IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 10, pages 2809–2823, 2016. (Cité en page 14.)

[Diffie 1976] Whitfield Diffie et Martin Hellman. *New directions in cryptography.* IEEE transactions on Information Theory, vol. 22, no. 6, pages 644–654, 1976. (Cité en page 14.)

[Djoudi 2015] Adel Djoudi et Sébastien Bardin. *Binsec : Binary code analysis with low-level regions.* In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 212–217. Springer, 2015. (Cité en pages 45 et 46.)

[Dumont 2019] Mathieu Dumont, Mathieu Lisart et Philippe Maurine. *Electromagnetic fault injection : how faults occur.* In 2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pages 9–16. IEEE, 2019. (Cité en page 19.)

[Dureuil 2015] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas et Jessy Clédière. *From code review to fault injection attacks : Filling the gap using fault model inference.* In International conference on smart card research and advanced applications, pages 107–124. Springer, 2015. (Cité en pages 17, 23 et 25.)

[Dureuil 2016a] Louis Dureuil. *Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute.* Theses, Université Grenoble Alpes, Octobre 2016. (Cité en pages ix, 2, 27, 33, 106 et 143.)

[Dureuil 2016b] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen et Philippe de Choudens. *FISSC : A Fault Injection and Simulation Secure Collection.* In Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings, pages 3–11, 2016. (Cité en pages 22, 23, 70, 106 et 156.)

[Dutertre 2014a] Bruno Dutertre. *Yices 2.2.* In International Conference on Computer Aided Verification, pages 737–744. Springer, 2014. (Cité en page 47.)

[Dutertre 2014b] Jean-Max Dutertre, Stephan De Castro, Alexandre Sarafianos, Noémie Boher, Bruno Rouzeyre, Mathieu Lisart, Joel Damiens, Philippe Candelier, Marie-Lise Flottes et Giorgio Di Natale. *Laser attacks on integrated circuits : from CMOS to FD-SOI.* In 2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS), pages 1–6. IEEE, 2014. (Cité en page 19.)

[Faurax 2006] Olivier Faurax, Laurent Freund, Assia Tria, Traian Muntean et Frédéric Bancel. *A generic method for fault injection in circuits.* In 2006 6th International Workshop on System on Chip for Real Time Applications, pages 211–214. IEEE, 2006. (Cité en page 33.)

[Faurax 2009] Olivier Faurax. *Évaluation par simulation de la sécurité des circuits face aux attaques par faute.* PhD thesis, Université de la Méditerranée, 2009. (Cité en pages 7, 29 et 33.)

[Fiat 1986] Amos Fiat et Adi Shamir. *How to prove yourself : Practical solutions to identification and signature problems.* In Conference on the theory and application of cryptographic techniques, pages 186–194. Springer, 1986. (Cité en page 14.)

[Gandolfi 2001] Karine Gandolfi, Christophe Mourtel et Francis Olivier. *Electromagnetic analysis : Concrete results.* In International workshop on cryptographic hardware and embedded systems, pages 251–261. Springer, 2001. (Cité en page 14.)

[Ganesh 2002] Vijay Ganesh et David L Dill. *A decision procedure for bit-vectors and arrays.* In International conference on computer aided verification, pages 519–531. Springer, 2002. (Cité en page 47.)

[Gangolli 2022] Aakash Gangolli, Qusay H Mahmoud et Akramul Azim. *A Systematic Review of Fault Injection Attacks on IoT Systems.* Electronics, vol. 11, no. 13, page 2023, 2022. (Cité en page 27.)

[Georgakoudis 2017] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S Nikolopoulos et Martin Schulz. *Refine : Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed.* In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–14, 2017. (Cité en pages 21, 23, 30, 31, 32 et 95.)

[Given-Wilson 2017] Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet et Axel Legay. *An automated formal process for detecting fault injection vulnerabilities in binaries and case study on PRESENT.* In 2017 IEEE Trustcom/BigDataSE/ICESS, pages 293–300. IEEE, 2017. (Cité en pages 27 et 128.)

[Godefroid 2005] Patrice Godefroid, Nils Klarlund et Koushik Sen. *DART : Directed automated random testing.* In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 213–223, 2005. (Cité en pages 8 et 45.)

[Godefroid 2008] Patrice Godefroid, Michael Y Levin, David A Molnar*et al. Automated whitebox fuzz testing.* In NDSS, volume 8, pages 151–166, 2008. (Cité en page 46.)

[Godefroid 2011] Patrice Godefroid. *Higher-order test generation.* In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, pages 258–269, 2011. (Cité en pages 44 et 46.)

[Goswami 1997] Kumar K Goswami. *DEPEND : A simulation-based environment for system level dependability analysis.* IEEE Transactions on Computers, vol. 46, no. 1, pages 60–74, 1997. (Cité en pages 32 et 33.)

[Goubet 2015] Lucien Goubet, Karine Heydemann, Emmanuelle Encrenaz et Ronald De Keulenaer. *Efficient Design and Evaluation of Countermeasures against Fault Attack with Formal Verification.* In 14th Smart Card Research and Advanced Application Conference, CARDIS 2015, Novembre 2015. (Cité en page 107.)

[Gruss 2016] Daniel Gruss, Clémentine Maurice et Stefan Mangard. *Rowhammer. js : A remote software-induced fault attack in javascript.* In International conference on detection of intrusions and malware, and vulnerability assessment, pages 300–321. Springer, 2016. (Cité en page 6.)

[Gupta 2018] Haritabh Gupta, Shamik Sural, Vijayalakshmi Atluri et Jaideep Vaidya. *A side-channel attack on smartphones : Deciphering key taps using built-in microphones*. Journal of Computer Security, vol. 26, no. 2, pages 255–281, 2018. (Cité en page 14.)

[Haedicke 2011] Finn Haedicke, Stefan Frehse, Görschwin Fey, Daniel Große et Rolf Drechsler. *metaSMT : Focus on Your Application not on Solver Integration*. In DIFTS@ FMCAD, 2011. (Cité en page 47.)

[Heydemann 2017] Karine Heydemann. *Sécurité et performance des applications : analyses et optimisations multi-niveaux*. PhD thesis, 2017. (Cité en pages 27 et 29.)

[Heydemann 2019] Karine Heydemann, Jean-François Lalande et Pascal Berthomé. *Formally verified software countermeasures for control-flow integrity of smart card C code*. Computers & Security, vol. 85, pages 202–224, 2019. (Cité en pages 105, 106 et 134.)

[Hutter 2013] Michael Hutter et Jörn-Marc Schmidt. *The temperature side channel and heating fault attacks*. In International Conference on Smart Card Research and Advanced Applications, pages 219–235. Springer, 2013. (Cité en page 19.)

[INRIA 1989] INRIA. *The Coq Proof Assistant*, 1989. (Cité en page 8.)

[ISO 2017] ISO. *Common Criteria for Information Technology Security Evaluation*. https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R5.pdf, 2017. [Online ; accessed 02-May-21]. (Cité en page 2.)

[Jamrozik 2013] Konrad Jamrozik, Gordon Fraser, Nikolai Tillman et Jonathan de Halleux. *Generating test suites with augmented dynamic symbolic execution*. In International Conference on Tests and Proofs, pages 152–167. Springer, 2013. (Cité en page 46.)

[Jia 2015] Xiangyang Jia, Carlo Ghezzi et Shi Ying. *Enhancing reuse of constraint solutions to improve symbolic execution*. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, pages 177–187, 2015. (Cité en page 46.)

[JIL 2020] JIL. *Joint Interpretation Library - Application of Attack Potential to Smartcards and Similar Devices*. https://www.sogis.eu/documents/cc/domains/sc/JIL-Application-of-Attack-Potential-to-Smartcards-v3-1.pdf, 2020. [Online ; accessed 25-April-22]. (Cité en page 6.)

[Juniper 2018] Juniper. *Internet of Things connected devices to almost triple to over 38 billion units by 2020*, 2018. (Cité en page 1.)

[Kanawati 1992] Ghani A Kanawati, Nasser A Kanawati et Jacob A Abraham. *FERRARI : A tool for the validation of system dependability properties*. In Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on, pages 336–344. IEEE, 1992. (Cité en pages 14, 30, 31 et 32.)

[Kao 1993] W-I Kao, Ravishankar K. Iyer et Dong Tang. *FINE : A fault injection and monitoring environment for tracing the UNIX system behavior under faults*. IEEE Transactions on Software Engineering, vol. 19, no. 11, pages 1105–1118, 1993. (Cité en page 30.)

[Kao 1996] Wei-Lun Kao et Ravishankar K Iyer. *DEFINE : A distributed fault injection and monitoring environment*. In Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pages 252–259. IEEE, 1996. (Cité en pages 30 et 32.)

[Karlsson 1994] Johan Karlsson, Peter Liden, Peter Dahlgren, Rolf Johansson et Ulf Gunneflo. *Using heavy-ion radiation to validate fault-handling mechanisms*. IEEE micro, vol. 14, no. 1, pages 8–23, 1994. (Cité en page 14.)

[Kelemen 2007] Péter Kelemen. *Silent Corruptions*. LCSC 2007, 2007. (Cité en page 21.)

[Kelly 2017] M. S. Kelly, K. Mayes et J. F. Walker. *Characterising a CPU fault attack model via run-time data analysis*. In 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pages 79–84, 2017. (Cité en page 19.)

[Kim 2007] Chong Hee Kim et Jean-Jacques Quisquater. *Fault attacks for CRT based RSA : New attacks, new results, and new countermeasures*. In IFIP International Workshop on Information Security Theory and Practices, pages 215–228. Springer, 2007. (Cité en page 26.)

[Kim 2014] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai et Onur Mutlu. *Flipping bits in memory without accessing them : An experimental study of DRAM disturbance errors*. ACM SIGARCH Computer Architecture News, vol. 42, no. 3, pages 361–372, 2014. (Cité en pages 3, 6 et 23.)

[King 1976] James C. King. *Symbolic Execution and Program Testing*. Commun. ACM, vol. 19, no. 7, pages 385–394, Juillet 1976. (Cité en pages 8 et 42.)

[KLEE 2008a] KLEE. *Publications and Systems Using KLEE*, 2008. (Cité en page 47.)

[Klee 2008b] Project Klee. *Klee Github page*, 2008. (Cité en page 47.)

[Kocher 1996] Paul C Kocher. *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*. In Annual International Cryptology Conference, pages 104–113. Springer, 1996. (Cité en page 14.)

[Kocher 1999] Paul Kocher, Joshua Jaffe et Benjamin Jun. *Differential power analysis*. In Annual international cryptology conference, pages 388–397. Springer, 1999. (Cité en page 14.)

[Kocher 2011] Paul Kocher, Joshua Jaffe, Benjamin Jun et Pankaj Rohatgi. *Introduction to differential power analysis*. Journal of Cryptographic Engineering, vol. 1, no. 1, pages 5–27, 2011. (Cité en page 14.)

[Kocher 2019] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher *et al. Spectre attacks : Exploiting speculative execution*. In 2019 IEEE Symposium on Security and Privacy (SP), pages 1–19. IEEE, 2019. (Cité en page 3.)

[Kooli 2014] Maha Kooli et Giorgio Di Natale. *A survey on simulation-based fault injection tools for complex systems*. In 2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS), pages 1–6. IEEE, 2014. (Cité en pages 27 et 29.)

[Lacombe 2021] Guilhem Lacombe, David Feliot, Etienne Boespflug et Marie-Laure Potet. *Combining Static Analysis and Dynamic Symbolic Execution in a Toolchain to detect Fault Injection Vulnerabilities*. In PROOFS WORKSHOP (SECURITY PROOFS FOR EMBEDDED SYSTEMS), 2021. (Cité en pages 23, 75 et 78.)

[Lacombe 2023] Guilhem Lacombe, David Féliot, Etienne Boespflug et Marie-Laure Potet. *Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities*. Journal of Cryptographic Engineering, pages 1–18, 2023. (Cité en pages vii, 75, 76, 78 et 79.)

[Lalande 2014] Jean-François Lalande, Karine Heydemann et Pascal Berthomé. *Software countermeasures for control flow integrity of smart card C codes*. In Miroslaw Kutylowski et Jaideep Vaidya, editeurs, ESORICS - 19th European Symposium on Research in Computer Security, volume 8713 of *Lecture Notes in Computer Science*,

pages 200–218, Wroclaw, Poland, Septembre 2014. Springer International Publishing. (Cité en pages 1, 22, 25, 105, 106, 131, 134, 141, 150 et 153.)

[Larsson 2007] Daniel Larsson et REINER Hahnle. *Symbolic Fault-Injection.* In International Verification Workshop (VERIFY), volume 259, pages 85–103, 2007. (Cité en pages 35, 36 et 37.)

[Lattner 2004] Chris Lattner et Vikram Adve. *LLVM : A compilation framework for lifelong program analysis & transformation.* In International Symposium on Code Generation and Optimization, 2004. CGO 2004., pages 75–86. IEEE, 2004. (Cité en pages 40 et 41.)

[Laurent 2018] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula et Athanasios Papadimitriou. *On the Importance of Analysing Microarchitecture for Accurate Software Fault Models.* In 2018 21st Euromicro Conference on Digital System Design (DSD), pages 561–564. IEEE, 2018. (Cité en page 21.)

[Laurent 2019] J. Laurent, V. Beroulle, C. Deleuze et F. Pebay-Peyroula. *Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures.* In 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pages 252–255, 2019. (Cité en pages 21 et 100.)

[Le 2018a] Hoang M Le, Vladimir Herdt, Daniel Große et Rolf Drechsler. *Resilience evaluation via symbolic fault injection on intermediate code.* In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 845–850. IEEE, 2018. (Cité en page 21.)

[Le 2018b] Hoang M Le, Vladimir Herdt, Daniel Große et Rolf Drechsler. *Resilience evaluation via symbolic fault injection on intermediate code.* In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 845–850. IEEE, 2018. (Cité en pages 35 et 36.)

[Li 2011] Guodong Li, Indradeep Ghosh et Sreeranga P Rajan. *KLOVER : A symbolic execution and automatic test generation tool for C++ programs.* In International Conference on Computer Aided Verification, pages 609–615. Springer, 2011. (Cité en page 47.)

[List 2015] CEA List. *Binsec Github page*, 2015. (Cité en page 45.)

[LLC 1999] MultiMedia LLC. *Coreutils*, 1999. (Cité en page 47.)

[LLVM 2011] LLVM. *LLVM Project - Github*, 2011. (Cité en page 42.)

[LLVM 2021] LLVM. *LLVM Project - Publications*, 2021. (Cité en page 42.)

[Lu 2015a] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas et Karthik Pattabiraman. *Llfi : An intermediate code-level fault injection tool for hardware faults.* In 2015 IEEE International Conference on Software Quality, Reliability and Security, pages 11–16. IEEE, 2015. (Cité en page 21.)

[Lu 2015b] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas et Karthik Pattabiraman. *Llfi : An intermediate code-level fault injection tool for hardware faults.* In 2015 IEEE International Conference on Software Quality, Reliability and Security, pages 11–16. IEEE, 2015. (Cité en pages 31 et 32.)

[Machemie 2011] J-B. Machemie, C. Mazin, J-L. Lanet et J. Cartigny. *SmartCM a smart card fault injection simulator.* In IEEE International Workshop on Information Forensics and Security. IEEE, 2011. (Cité en page 23.)

[Madeira 1994] Henrique Madeira, Mário Rela, Francisco Moreira et João Gabriel Silva. *RIFLE : A general purpose pin-level fault injector.* In European Dependable Computing Conference, pages 197–216. Springer, 1994. (Cité en page 29.)

[Maistri 2014] Paolo Maistri, Regis Leveugle, Lilian Bossuet, Alain Aubert, Viktor Fischer, Bruno Robisson, Nicolas Moro, Philippe Maurine, J-M Dutertre et Mathieu Lisart. *Electromagnetic analysis and fault injection onto secure circuits*. In 2014 22nd International Conference on Very Large Scale Integration (VLSI-SoC), pages 1–6. IEEE, 2014. (Cité en page 19.)

[Majumdar 2007] Rupak Majumdar et Koushik Sen. *Hybrid concolic testing*. In 29th International Conference on Software Engineering (ICSE'07), pages 416–426. IEEE, 2007. (Cité en page 46.)

[Manès 2019] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz et Maverick Woo. *The art, science, and engineering of fuzzing : A survey*. IEEE Transactions on Software Engineering, 2019. (Cité en page 8.)

[Martin 2022] Thibault Martin, Nikolai Kosmatov et Virgile Prevosto. *Verifying Redundant-Check Based Countermeasures : A Case Study*. pages 1849–1852, 2022. (Cité en page 107.)

[May 1978] Timothy C May et Murray H Woods. *A new physical mechanism for soft errors in dynamic memories*. In 16th International Reliability Physics Symposium, pages 33–40. IEEE, 1978. (Cité en page 13.)

[Moro 2013] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson et E. Encrenaz. *Electromagnetic Fault Injection : Towards a Fault Model on a 32-bit Microcontroller*. In 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 77–88, 2013. (Cité en pages 19, 22 et 23.)

[Moro 2014a] N. Moro, K. Heydemann, B. Robisson, E. Encrenaz et B. Robisson. *Formal verification of a software countermeasure against instruction skip attacks*. In Journal of Cryptographic Engineering, pages 145–156, 2014. (Cité en page 11.)

[Moro 2014b] Nicolas Moro. *Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués*. PhD thesis, Université Pierre et Marie Curie, Paris, 2014. 2014PA066616. (Cité en pages 24, 25 et 105.)

[Moro 2014c] Nicolas Moro, Karine Heydemann, Amine Dehbaoui, Bruno Robisson et Emmanuelle Encrenaz. *Experimental evaluation of two software countermeasures against fault attacks*. In 2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pages 112–117. IEEE, 2014. (Cité en page 106.)

[Morris 1979] Morris. *Morris Worm*. `https://en.wikipedia.org/wiki/Morris_worm`, 1979. (Cité en page 8.)

[Nashimoto 2017] Shoei Nashimoto, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji et Takafumi Aoki. *Buffer overflow attack with multiple fault injection and a proven countermeasure*. Journal of Cryptographic Engineering, vol. 7, no. 1, pages 35–46, 2017. (Cité en page 15.)

[Nashimoto 2022] Shoei Nashimoto, Daisuke Suzuki, Rei Ueno et Naofumi Homma. *Bypassing Isolated Execution on RISC-V using Side-Channel-Assisted Fault-Injection and Its Countermeasure*. IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 28–68, 2022. (Cité en page 15.)

[Natella 2016] Roberto Natella, Domenico Cotroneo et Henrique S. Madeira. *Assessing Dependability with Software Fault Injection : A Survey*. ACM Computing Surveys, vol. 48, no. 3, pages 1–55, F ?rier 2016. (Cité en page 26.)

[Nicolescu 2004] Bogdan Nicolescu, Yvon Savaria et Raoul Velazco. *Software detection mechanisms providing full coverage against single bit-flip faults*. IEEE Transactions on Nuclear science, vol. 51, no. 6, pages 3510–3518, 2004. (Cité en page 105.)

[Nipkow 2002] Tobias Nipkow, Lawrence C Paulson et Markus Wenzel. Isabelle/hol : a proof assistant for higher-order logic, volume 2283. Springer Science & Business Media, 2002. (Cité en page 8.)

[Oh 2002] Nahmsuk Oh, Philip P Shirvani et Edward J McCluskey. *Control-flow checking by software signatures.* IEEE transactions on Reliability, vol. 51, no. 1, pages 111–122, 2002. (Cité en pages 25, 105 et 131.)

[Park 2014] Kyungbae Park, Sanghyeon Baeg, ShiJie Wen et Richard Wong. *Active-precharge hammering on a row induced failure in ddr3 sdrams under 3× nm technology.* In 2014 IEEE International Integrated Reliability Workshop Final Report (IIRW), pages 82–85. IEEE, 2014. (Cité en pages iii, iv, 3 et 15.)

[Pattabiraman 2008] Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk et Ravishankar Iyer. *SymPLFIED : Symbolic program-level fault injection and error detection framework.* In Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on, pages 472–481. IEEE, 2008. (Cité en pages 35, 36 et 106.)

[Pattabiraman 2012] Karthik Pattabiraman, Nithin M Nakka, Zbigniew T Kalbarczyk et Ravishankar K Iyer. *Symplfied : Symbolic program-level fault injection and error detection framework.* IEEE Transactions on Computers, vol. 62, no. 11, pages 2292–2307, 2012. (Cité en pages 35 et 106.)

[Potet 2014] Marie-Laure Potet, Laurent Mounier, Maxime Puys et Louis Dureuil. *Lazart : A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections.* In Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, pages 213–222. IEEE, 2014. (Cité en pages 22, 35, 36, 39, 91, 99, 100 et 106.)

[Poucheret 2011] François Poucheret, Karim Tobich, Mathieu Lisarty, L Chusseauz, B Robissonx et Philippe Maurine. *Local and direct em injection of power into cmos integrated circuits.* In 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 100–104. IEEE, 2011. (Cité en page 14.)

[Proy 2017] Julien Proy, Karine Heydemann, Alexandre Berzati et Albert Cohen. *Compiler-assisted loop hardening against fault attacks.* ACM Transactions on Architecture and Code Optimization (TACO), vol. 14, no. 4, pages 1–25, 2017. (Cité en pages 25 et 141.)

[Puys 2014] Maxime Puys, Lionel Riviere, Julien Bringer et Thanh-ha Le. *High-level simulation for multiple fault injection evaluation.* In Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance, pages 293–308. Springer, 2014. (Cité en page 71.)

[Rauzy 2014] Pablo Rauzy et Sylvain Guilley. *A Formal Proof of Countermeasures Against Fault Injection Attacks on CRT-RSA.* Journal of Cryptographic Engineering, vol. 4, no. 3, pages 173–185, 2014. (Cité en page 11.)

[Reis 2005] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan et D. I. August. *SWIFT : software implemented fault tolerance.* In International Symposium on Code Generation and Optimization, pages 243–254, 2005. (Cité en pages 1, 25, 105 et 141.)

[Rivest 1978] Ronald L Rivest, Adi Shamir et Leonard Adleman. *A method for obtaining digital signatures and public-key cryptosystems.* Communications of the ACM, vol. 21, no. 2, pages 120–126, 1978. (Cité en page 14.)

[Rivière 2014] Lionel Rivière, Marie-Laure Potet, Thanh-Ha Le, Julien Bringer, Hervé Chabanne et Maxime Puys. *Combining High-Level and Low-Level Approaches to Evaluate Software Implementations Robustness Against Multiple Fault Injection Attacks.*

In Foundations and Practice of Security, Montreal, Canada, Novembre 2014. (Cité en page 21.)

[Rivière 2015] Lionel Rivière, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer et Laurent Sauvage. *High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures.* In IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015, pages 62–67. IEEE, 2015. (Cité en pages 21 et 33.)

[Roemer 2012] Ryan Roemer, Erik Buchanan, Hovav Shacham et Stefan Savage. *Return-oriented programming : Systems, languages, and applications.* ACM Transactions on Information and System Security (TISSEC), vol. 15, no. 1, pages 1–34, 2012. (Cité en page 9.)

[Roscian 2013] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre et Assia Tria. *Fault model analysis of laser-induced faults in sram memory cells.* In 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 89–98. IEEE, 2013. (Cité en pages iii, iv et 14.)

[Rosen 1988] Barry K Rosen, Mark N Wegman et F Kenneth Zadeck. *Global value numbers and redundant computations.* In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 12–27, 1988. (Cité en page 42.)

[Salwan 2020] Jonathan Salwan. *L'usage de l'exécution symbolique pour la déobfuscation binaire en milieu industriel.* PhD thesis, Université Grenoble Alpes, 2 2020. (Cité en page 46.)

[Sasnauskas 2010] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski et Klaus Wehrle. *KleeNet : discovering insidious interaction bugs in wireless sensor networks before deployment.* In Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, pages 186–196, 2010. (Cité en page 47.)

[Sayeed 2019] Sarwar Sayeed, Hector Marco-Gisbert, Ismael Ripoll et Miriam Birch. *Control-flow integrity : attacks and protections.* Applied Sciences, vol. 9, no. 20, page 4229, 2019. (Cité en page 21.)

[Schmidt 2007] Jörn-Marc Schmidt et Michael Hutter. Optical and em fault-attacks on crt-based rsa : Concrete results. na, 2007. (Cité en page 34.)

[Schnorr 1991] Claus-Peter Schnorr. *Efficient signature generation by smart cards.* Journal of cryptology, vol. 4, no. 3, pages 161–174, 1991. (Cité en page 14.)

[Schwartz 2010] Edward J Schwartz, Thanassis Avgerinos et David Brumley. *All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask).* In 2010 IEEE symposium on Security and privacy, pages 317–331. IEEE, 2010. (Cité en page 46.)

[Seaborn 2015] Mark Seaborn et Thomas Dullien. *Exploiting the DRAM rowhammer bug to gain kernel privileges.* Black Hat, vol. 15, page 71, 2015. (Cité en page 6.)

[Segall 1988] Zary Segall, D Vrsalovic, D Siewiorek, D Ysskin, J Kownacki, J Barton, R Dancey, A Robinson et T Lin. *FIAT-fault injection based automated testing environment.* In Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on, pages 102–107, June 1988. (Cité en pages 14, 30 et 32.)

[Sen 2005] Koushik Sen, Darko Marinov et Gul Agha. *CUTE : A concolic unit testing engine for C.* ACM SIGSOFT Software Engineering Notes, vol. 30, no. 5, pages 263–272, 2005. (Cité en page 45.)

[Séré 2011] A. Séré, J-L. Lanet et J. Iguchi-Cartigny. *Evaluation of Countermeasures Against Fault Attacks on Smart Cards.* International Journal of Security and Its Applications, vol. 5, no. 2, 2011. (Cité en page 23.)

[Shacham 2004] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu et Dan Boneh. *On the effectiveness of address-space randomization.* In Proceedings of the 11th ACM conference on Computer and communications security, pages 298–307, 2004. (Cité en page 1.)

[Shacham 2007] Hovav Shacham. *The geometry of innocent flesh on the bone : Return-into-libc without function calls (on the x86).* In Proceedings of the 14th ACM conference on Computer and communications security, pages 552–561, 2007. (Cité en page 20.)

[Shamir 1999] Adi Shamir. *Method and apparatus for protecting public key schemes from timing and fault attacks*, Novembre 23 1999. US Patent 5,991,415. (Cité en page 71.)

[Sharma 2013] Vishal Chandra Sharma, Arvind Haran, Zvonimir Rakamaric et Ganesh Gopalakrishnan. *Towards formal approaches to system resilience.* In 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing, pages 41–50. IEEE, 2013. (Cité en page 21.)

[Shoshitaishvili 2016] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel et Giovanni Vigna. *SoK : (State of) The Art of War : Offensive Techniques in Binary Analysis.* In IEEE Symposium on Security and Privacy, 2016. (Cité en pages 8 et 46.)

[Skorobogatov 2002] Sergei P Skorobogatov et Ross J Anderson. *Optical fault induction attacks.* In International workshop on cryptographic hardware and embedded systems, pages 2–12. Springer, 2002. (Cité en page 14.)

[SSTIC 2020] SSTIC. *Inter-CESTI : Methodological and Technical Feedbacks on Hardware Devices Evaluations.* https://www.sstic.org/media/SSTIC2020/SSTIC-actes/inter-cesti_methodological_and_technical_feedbacks/SSTIC2020-Article-inter-cesti_methodological_and_technical_feedbacks_on_hardware_devices_evaluations-benadjila.pdf, 2020. (Cité en pages 3, 15 et 26.)

[Stephens 2016] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel et Giovanni Vigna. *Driller : Augmenting fuzzing through selective symbolic execution.* In NDSS, volume 16, pages 1–16, 2016. (Cité en page 46.)

[Theissing 2013] Nikolaus Theissing, Dominik Merli, Michael Smola, Frederic Stumpf et Georg Sigl. *Comprehensive analysis of software countermeasures against fault attacks.* In 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 404–409. IEEE, 2013. (Cité en page 106.)

[Thomas 2013] Anna Thomas et Karthik Pattabiraman. *LLFI : An intermediate code level fault injector for soft computing applications.* In Workshop on Silicon Errors in Logic System Effects (SELSE), 2013. (Cité en pages 31 et 32.)

[Timmers 2016] Niek Timmers, Albert Spruyt et Marc Witteman. *Controlling PC on ARM using fault injection.* In 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pages 25–35. IEEE, 2016. (Cité en pages 1 et 15.)

[Tiri 2007] Kris Tiri. *Side-channel attack pitfalls.* In 2007 44th ACM/IEEE Design Automation Conference, pages 15–20. IEEE, 2007. (Cité en page 14.)

[Turing 1937] Alan Mathison Turing. *On computable numbers, with an application to the Entscheidungsproblem.* Proceedings of the London mathematical society, vol. 2, no. 1, pages 230–265, 1937. (Cité en page 6.)

[Van Der Kouwe 2014] Erik Van Der Kouwe, Cristiano Giuffrida et Andrew S Tanenbaum. *Evaluating distortion in fault injection experiments.* In 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering, pages 25–32. IEEE, 2014. (Cité en page 21.)

[Van Woudenberg 2011] Jasper GJ Van Woudenberg, Marc F Witteman et Federico Menarini. *Practical optical fault injection on secure microcontrollers.* In 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 91–99. IEEE, 2011. (Cité en page 19.)

[Verbauwhede 2011] I. Verbauwhede, D. Karaklajic et J. Schmidt. *The Fault Attack Jungle - A Classification Model to Guide You.* In 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, pages 3–8, 2011. (Cité en page 20.)

[Veyrat-Charvillon 2012] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof et François-Xavier Standaert. *Shuffling against side-channel attacks : A comprehensive study with cautionary note.* In International Conference on the Theory and Application of Cryptology and Information Security, pages 740–757. Springer, 2012. (Cité en page 10.)

[Visser 2012] Willem Visser, Jaco Geldenhuys et Matthew B Dwyer. *Green : reducing, reusing and recycling constraints in program analysis.* In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pages 1–11, 2012. (Cité en page 46.)

[Werner 2015] Mario Werner, Erich Wenger et Stefan Mangard. *Protecting the control flow of embedded processors against fault attacks.* In International Conference on Smart Card Research and Advanced Applications, pages 161–176. Springer, 2015. (Cité en page 23.)

[Werner 2020] Vincent Werner, Laurent Maingault et Marie-Laure Potet. *An end-to-end approach for multi-fault attack vulnerability assessment.* In 2020 Workshop on Fault Detection and Tolerance in Cryptography (FDTC), pages 10–17. IEEE, 2020. (Cité en pages 17 et 23.)

[Werner 2022] Vincent Werner. *Optimiser l'identification et l'exploitation de vulnérabilité à l'injection de faute sur microcontröleurs.* PhD thesis, Université Grenoble Alpes, 2022. (Cité en pages 33 et 106.)

[Witteman 2008] Marc Witteman et Martijn Oostdijk. *Secure application programming in the presence of side channel attacks.* In RSA conference, volume 2008, 2008. (Cité en pages 10 et 24.)

[Wu 2017] Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang et Zizhong Chen. *Silent data corruption resilient two-sided matrix factorizations.* In Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 415–427, 2017. (Cité en page 14.)

[Xiao 2011] Xusheng Xiao, Tao Xie, Nikolai Tillmann et Jonathan De Halleux. *Precise identification of problems for structural test generation.* In Proceedings of the 33rd International Conference on Software Engineering, pages 611–620, 2011. (Cité en page 47.)

[Yang 2012] Guowei Yang, Corina S Păsăreanu et Sarfraz Khurshid. *Memoized symbolic execution.* In Proceedings of the 2012 International Symposium on Software Testing and Analysis, pages 144–154, 2012. (Cité en page 46.)

[Yuce 2016] Bilgiday Yuce, Nahid F Ghalaty, Chinmay Deshpande, Conor Patrick, Leyla Nazhandali et Patrick Schaumont. *FAME : Fault-attack aware microprocessor extensions for hardware fault detection and software fault response.* In Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, pages 1–8. HASSP, 2016. (Cité en page 25.)

[Yuce 2018] Bilgiday Yuce, Patrick Schaumont et Marc Wittenman. *Fault Attacks on Secure Embedded Software : Threats, Design, and Evaluation.* In Journal of Hardware and Systems Security, pages 111–130, 2018. (Cité en pages iii, iv, 14 et 24.)

[Yun 2018] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang et Taesoo Kim. {*QSYM*} *: A practical concolic execution engine tailored for hybrid fuzzing.* In 27th USENIX Security Symposium (USENIX Security 18), pages 745–761, 2018. (Cité en page 46.)

[Zarandi 2003] Hamid R Zarandi, Seyed Ghassem Miremadi et Alireza Ejlali. *Dependability analysis using a fault injection tool based on synthesizability of HDL models.* In Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems, pages 485–492. IEEE, 2003. (Cité en page 33.)

[Ziegler 1979] James F Ziegler et William A Lanford. *Effect of cosmic rays on computer memories.* Science, vol. 206, no. 4420, pages 776–788, 1979. (Cité en page 14.)

[Zussa 2014] L. Zussa, A. Dehbaoui, K. Tobich, J. Dutertre, P. Maurine, L. Guillaume-Sage, J. Clediere et A. Tria. *Efficiency of a glitch detector against electromagnetic fault injection.* In 2014 Design, Automation Test in Europe Conference Exhibition (DATE), pages 1–6, 2014. (Cité en page 24.)