Session doctorant : Analyse automatisée de binaires à la recherche de vulnérabilités matérielles *

Theo De Castro Pinto[1,2]
[1] Serma SAFETY & SECURITY
[2] Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Talence, France

Résumé

Les cartes à puce et autres appareils embarqués mettent les développeurs face à un problème de taille : les attaques physiques. Parmi celles-ci, certaines visent à modifier le comportement de la puce (temporairement ou non) grâce à une perturbation physique (laser, glitch ou autre). Ces dernières, qui peuvent être multiples au cours d'une seule exécution, sont difficiles à détecter et même si des contre-mesures existent, elles peuvent être insuffisantes ou disparaître à la compilation. Pour ces raisons une analyse des binaires générés est nécessaire, afin de détecter la présence (ou non) de vulnérabilités face à ces attaques. Cette analyse est généralement effectuée à la main. Ce papier présente un outil d'aide à la recherche de vulnérabilités matérielles pour du code binaire. Il s'appuie sur l'outil Binsec. Il permet de détecter des vulnérabilités dans un binaire en fonction d'un schéma d'attaque et d'une cible à atteindre. Il s'agit de résultats préliminaires obtenus lors d'un stage de Master. Ce papier présente aussi quelques problématiques de recherche qui seront abordées au cours de la thèse.

1 Introduction

En informatique, une vulnérabilité est une faille de sécurité exploitable par un attaquant et pouvant mener à l'altération du comportement du logiciel ou à la divulgation d'informations secrètes. Il existe des vulnérabilités dites logicielles (buffer overflow, ROP, JOP [9], ...) et des vulnérabilités dites matérielles, ce sont ces dernières qui sont considérées dans ce papier. Ces vulnérabilités peuvent être exploitées via des attaques par injection de faute (en utilisant des LASER, des glitch de tension, ...). Ce travail se focalise sur les systèmes embarqués.

Une attaque matérielle peut avoir de nombreuses conséquences. Il est possible de classer ces effets en quatre catégories atomiques : les *bit set* (un bit du programme est mis à 1), les *bit reset* (un bit du programme est mis à 0), les *bit flip* (un bit du programme est inversé) et le *rejeu d'une instruction précédente*. Ces différents effets peuvent être combinés ce qui donne lieu à des attaques plus sophistiquées, par exemple, un branchement conditionnel inversé, une instruction transformée ou sautée. La conséquence de tout cela est que des comportements non-attendus peuvent être ajoutés via des attaques par injection de faute.

Il est donc nécessaire de limiter les vulnérabilités matérielles dans les programmes, notamment embarqués, cependant c'est une tâche ardue. Les développeurs doivent imaginer un maximum d'attaques possibles afin d'intégrer des contre-mesures [1] qui pourront limiter le nombre de vulnérabilités exploitables. Un exemple typique de contre-mesure est la redondance des opérations dans le code. Il existe des organismes (tels que des CESTIs) qui évaluent la sécurité des logiciels embarqués et tentent d'y trouver des vulnérabilités. Actuellement, les évaluations consistent en des revues de code (source en conjonction avec le code assembleur fourni par le client) à la main (faites par des experts) et ces dernières ont le désavantage d'être très longues et difficiles.

Il y a donc un besoin d'analyse automatisée dans ce milieu. L'objectif de ce projet est de développer un outil permettant d'assister les experts dans leur recherche de vulnérabilités matérielles et d'en automatiser une grande partie. Les experts ont accès au code source et assembleur, mais le code assembleur est difficile à analyser pour un humain (en raison de sa taille et de son intrication). Pourtant, la compilation peut

* Ce projet est soutenu par l'ANRT via le financement CIFRE numéro 2021/1673.

1. Des sécurités permettant de détecter des attaques en faute.

```
1 cmp  r0, r1
2 bne  4
3 call  critical_function
4 halt
```

FIGURE 1 – Code assembleur non sécurisé

```
1 cmp  r0, r1
2 bne  6
3 cmp  r0, r1
4 bne  6
5 call  critical_function
6 halt
```

FIGURE 2 – Code assembleur un peu sécurisé

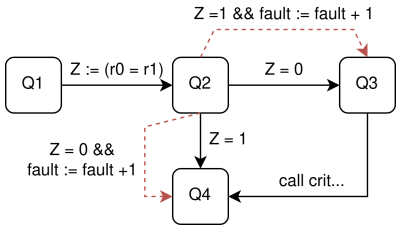

FIGURE 3 – Automate avec mutations correspondant au code en listing 1

souvent retirer des contre-mesures, donc l'analyse du code assembleur est inévitable et l'automatisation de ce procédé est une solution utile.

Différentes approches ont été proposées, par exemple certaines se basent sur de la vérification formelle de code par rapport à des propriétés (Frama-C [5] [10], BMC [3], ...), d'autres analysent [12] et ajoutent [8] automatiquement des contre-mesures. Une autre approche, qui va nous intéresser dans ce papier, consiste à détecter les vulnérabilités (via par exemple de l'exécution symbolique). Dans cette dernière catégorie, on peut citer FIVD [7] qui utilise de l'exécution concrète et des tests physiques ou bien Lazart [11] qui permet de détecter l'existence de vulnérabilités permettant d'atteindre un bout de code donné.

Notre méthodologie est une extension de celle proposée par Lazart. Ce dernier est un outil qui analyse du code LLVM (compilé à partir du code source) afin d'y détecter des vulnérabilités par injection de faute. Nous avons donc développé un outil qui applique et étend cette méthodologie aux binaires, notamment via l'utilisation de Binsec [6]. Nous avons développé un prototype dont les premiers résultats tendent à montrer qu'il permet de détecter plus de vulnérabilités que Lazart (la comparaison a été faite via des tests sur Wookey [1]).

2 Contributions et résultats préliminaires

Afin de clarifier quels sont les enjeux et comment fonctionne l'outil, des exemples de codes sont donnés en figures 1 et 2. Ces codes sont fonctionnellement équivalent. Dans l'idée `r0` est une valeur secrète (un code pin par exemple) et `r1` une entrée utilisateur. Un attaquant ne connaît donc pas le pin mais en exploitant des vulnérabilités matérielles, il est possible d'atteindre la fonction critique. En effet, une attaque en faute peut par exemple transformer l'instruction `bne 4` en `nop`, et dans ce cas la comparaison précédente est inutile et toute valeur donnée en entrée mènera à l'appel de la fonction. Le code 2 est donc plus sécurisé face à une attaque en faute que le code 1.

Dans le code en figure 2, cette attaque n'est pas suffisante grâce à la contre-mesure ajoutée (redondance du test), mais bien sûr, il est possible qu'un attaquant fasse deux attaques en faute dans le même binaire, dans ce cas il lui est possible de transformer les deux instruction `bne` en `nop`, ce qui aura les mêmes conséquences que l'attaque présentée dans le premier code. Enfin, il est intéressant de noter qu'en réalité, un compilateur peut supprimer ce type de redondance, notamment si les optimisations sont activées. Il est donc intéressant de faire une analyse au niveau binaire, car c'est plus représentatif du programme réellement exécuté. Cela permet de détecter, en particulier, des vulnérabilités qui n'existent pas au niveau du code source, et qui sont dues à des optimisations trop agressives.

Dans ces deux exemples, il est facile de trouver des vulnérabilités mais les experts qui analysent des binaires font face à des codes sources de plusieurs milliers de lignes et des codes assembleurs encore plus longs. Ce n'est donc pas envisageable de trouver toutes les vulnérabilités, ni même toutes les vulnérabilités exploitables dans des conditions réelles (celles qui sont vraiment problématiques).

L'approche que nous proposons est la suivante :

1. Transformer le binaire en un langage intermédiaire (DBA [2]) qui correspond à un automate (voir figure 3).

2. Ajouter des mutations dans cet automate, ce qui correspond à ajouter des transitions, voir par exemple la figure 3, ce sont les transitions en pointillés. Dans ce cas, le saut conditionnel `bne` a la possibilité d'être inversé lors de l'exploration de l'automate. La variable `fault` permet de compter le nombre de fautes injectées.
3. Explorer symboliquement l'automate (grâce à Binsec [6]) afin d'atteindre une cible (typiquement une instruction assembleur) pré-définie avec comme pré-condition que les valeurs secrètes ne sont pas connues.
4. Analyser les chemins atteignant la cible et en réduire le nombre car beaucoup de chemins générés sont en réalité équivalents. Un humain doit ensuite choisir quels chemins semblent correspondre à une vulnérabilité exploitable, il est donc nécessaire d'en avoir le moins possible.

L'étape 1 se fait grâce à l'outil Binsec (un outil d'analyse de binaires), le DBA étant le langage intermédiaire utilisé par Binsec. Cet outil a été choisi car Binsec est activement utilisé dans la recherche de vulnérabilités dans du code binaire. Il s'agit de plus d'un outil open source ce qui nous a permis de le modifier afin d'obtenir des informations supplémentaires lors de l'exécution symbolique.

L'étape 2 est plus complexe, car il est nécessaire de limiter le nombre de transitions rajoutées dans l'automate. Bien sûr il est possible de créer un automate très complexe tel que toutes les attaques imaginables sont possibles à tout moment mais l'exécution symbolique ne pourrait jamais explorer efficacement un tel automate.

Afin de limiter le nombre de transitions, il est nécessaire d'une part de définir un modèle de faute, c'est-à-dire quelles attaques sont autorisées dans la recherche, et d'autre part de déterminer à quels nœuds de l'automate il est utile ou non d'ajouter des transitions (certains nœuds n'ont absolument aucun effet sur la cible à atteindre). Le modèle de faute choisi est l'inversion des tests : tout branchement peut être transformé en son complément et le choix des nœuds découle de la méthode proposée par Lazart [11].

Un graphe de flot de contrôle est extrait du binaire, où les nœuds sont des blocs d'instructions se terminant par un branchement (et n'en contenant pas d'autre). L'approche naïve consiste à ajouter des transitions fautées à la fin de chacun de ces blocs mais cela est très coûteux lors de l'exécution symbolique et inutile dans certains cas. Nous utilisons une heuristique qui ajoute des mutations seulement lorsque cela est utile. En effet, s'il n'existe pas de chemin entre un nœud donné et la cible, il n'est pas muté, de la même manière si tous les chemins issus d'un nœud donné atteignent la cible à un moment, il n'est pas muté (car ces mutations n'ont aucune incidence sur l'atteignabilité de la cible.). Tous les autres nœuds le sont, donc des transitions sont rajoutées dans l'automate aux points correspondant (bien sûr lors de l'exécution ces transitions ne sont pas nécessairement empruntées à chaque fois).

Avec cette stratégie, il est possible d'obtenir des résultats sensiblement meilleurs que ceux avancés par Lazart : sur un exemple classique de l'état de l'art, Wookey [1], Lazart détecte deux vulnérabilités, et l'outil que nous avons développé en détecte trois, dont une non documentée par Lazart (cependant cette vulnérabilité n'est peut être pas exploitable, cela n'a pas pu être vérifié). Néanmoins, les performances temporelles de l'outil sont plus mitigées, ce qui est notamment dû au problème de l'explosion combinatoire liée à l'exécution symbolique. Enfin, lors du développement de notre outil, Lazart utilisait uniquement un modèle de faute se basant sur l'inversion de branchements. L'outil développé quant à lui peut travailler avec n'importe quel modèle de faute (mais pas nécessairement de manière efficace).

3 Discussions et perspectives

Malgré les résultats encourageants, l'outil développé reste peu utilisable dans des conditions réelles. Le problème majeur reste le temps d'exécution et le faible nombre d'études réalisées sur des cas concrets. Au cours de la thèse, l'objectif va donc être le passage à l'échelle. Il y a différents points centraux qui seront explorés afin de rendre l'exécution symbolique plus rapide. La première piste est d'utiliser de meilleures heuristiques d'exploration. En effet, il existe de très nombreux chemins d'exécution à explorer dans un binaire de taille réelle (il n'est jamais possible de l'explorer entièrement), il est utile de trouver une mesure qui permet d'explorer les chemins les plus "intéressants" en premier.

Le deuxième objectif est d'utiliser l'interprétation abstraite [4]. Son utilisation peut être double, d'une part elle peut être utilisée afin d'obtenir des résumés de fonction (celles qui ne contiennent aucune mutation) afin de ne plus les exécuter symboliquement. D'autre part, elle peut aussi être utilisée durant l'exécution symbolique afin d'accélérer les calculs (au risque d'augmenter le nombre de faux positifs).

Dans les cas pratiques, l'architecture utilisée est souvent ARMv7-M, qui correspond à du THUMB-2. Cependant, la traduction vers du DBA est lourde à cause de certaines instructions (notamment l'instruction `IT`), il existe cependant d'autres représentations intermédiaires qui pourraient alléger la traduction (par exemple : VEX [13]). Une troisième idée repose donc sur l'utilisation d'une représentation intermédiaire un peu modifiée (mélange de DBA et de techniques issues de différentes représentations intermédiaires).

Enfin, il est crucial de formaliser différents aspects du domaine, notamment les binaires et les modèles de faute. Il s'agira donc de définir formellement la syntaxe et la sémantique des différents objets manipulés. De plus, la cible à atteindre et les chemins "intéressants" ne sont actuellement pas des concepts formels ce qui rend la recherche parfois hasardeuse.

Références

[1] ANSSI. Wookey. `https://wookey-project.github.io/publi.html`. Accessed : 24-05-2022.

[2] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The bincoa framework for binary code analysis. In *International Conference on Computer Aided Verification*, pages 165–170. Springer, 2011.

[3] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. 2003.

[4] Patrick Cousot. Interprétation abstraite. *Technique et science informatique*, 19(1-2) :3, 2000.

[5] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International conference on software engineering and formal methods*, pages 233–247. Springer, 2012.

[6] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se : A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656. IEEE, 2016.

[7] Nisrine Jafri. *Formal fault injection vulnerability detection in binaries : a software process and hardware validation*. PhD thesis, Université Rennes 1, 2019.

[8] Pantea Kiaei, Cees-Bart Breunesse, Mohsen Ahmadi, Patrick Schaumont, and Jasper Van Woudenberg. Rewrite to reinforce : Rewriting the binary to apply countermeasures against fault injection. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 319–324. IEEE, 2021.

[9] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. Integration of rop/jop monitoring ips in an arm-based soc. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 331–336. IEEE, 2016.

[10] Thibault Martin, Nikolai Kosmatov, and Virgile Prevosto. Verifying redundant-check based countermeasures : A case study. 2022.

[11] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart : A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 213–222. IEEE, 2014.

[12] Pablo Rauzy and Sylvain Guilley. A formal proof of countermeasures against fault injection attacks on crt-rsa. *Journal of Cryptographic Engineering*, 4(3) :173–185, 2014.

[13] ANGR team. Intermediate representation - angr. `https://docs.angr.io/advanced-topics/ir`. Accessed : 24-05-2022.