BINSEC/REL : Exécution Symbolique Relationnelle Efficace pour l'Analyse de Binaire Constant-Time

Lesly-Ann Daniel[1], Sébastien Bardin[1], and Tamara Rezk[2]

[1]CEA List, Université Paris-Saclay
[2]INRIA Sophia-Antipolis, INDES Project
lesly-ann.daniel@cea.fr, sebastien.bardin@cea.fr, tamara.rezk@inria.fr

Résumé

Constant-time est une contre-mesure aux attaques temporelles qui interdit les branchements et les accès mémoires dépendants des secrets. Cette contre-mesure n'est généralement pas préservée par le compilateur et requiert donc de raisonner au niveau binaire. Or, les outils d'analyse dédiés à constant-time raisonnent actuellement à un plus haut niveau (C ou LLVM), approximent la sémantique du programme, ou ne passent pas à l'échelle. Nous concevons une technique d'analyse efficace au niveau binaire qui ne fait pas d'approximation sur la sémantique du programme, permettant à la fois de trouver des bugs ou de faire de la vérification bornée pour constant-time. Celle-ci s'appuie sur l'exécution symbolique relationnelle, à laquelle nous ajoutons des optimisations dédiées. Nous proposons un prototype, BINSEC/REL et réalisons des expériences sur un ensemble de 338 binaires cryptographiques, démontrant le passage à l'échelle de notre technique. De plus, en utilisant BINSEC/REL, nous avons automatisé et étendu une étude existante sur la préservation de constant-time par les compilateurs. Nous avons ainsi découvert des violations introduites par les compilateurs qui étaient hors de portée des outils d'analyse pour LLVM, soulignant l'importance de raisonner au niveau binaire.

Objet : AFADL 2020 résumé long. Travail original accepté à S&P 2020 [1].

Contexte : Constant-time contre les attaque temporelles.
Les attaques temporelles permettent à un attaquant capable de surveiller le temps d'exécution d'un système, d'extraire des informations sur les secrets manipulés par un programme (e.g. clés cryptographiques). Ces attaques exploitent des corrélations entre les secrets et le temps d'exécution du programme qui peuvent subvenir quand le *flot de contrôle* du programme dépend des secrets, mais aussi à travers ses *accès mémoire*, via les attaques par cache. La discipline de programmation *constant-time* est une contre-mesure qui permet de décorréler le temps d'exécution des secrets manipulés par un programme. Un programme est constant-time si

pour chaque traces d'exécution t et t' avec les même entrées publiques, t et t' ont le même flot de contrôle et les mêmes accès mémoire, peu importe la valeur des entrées secrètes. Cette contre-mesure est aujourd'hui largement utilisée pour sécuriser les implémentations cryptographiques (e.g. BearSSL, NaCL, HACL*, etc).

Problème : Un outil efficace pour vérifier constant-time au niveau binaire ?
Écrire une implémentation constant-time n'est pas trivial et requiert l'utilisation d'opérations bas niveau pour remplacer l'utilisation de branchements conditionnels. De plus, constant-time n'est généralement pas préservé par le compilateur [2]. Par exemple, le code `c=(x<y)-1` peut être, ou non, compilé vers un saut conditionnel selon la version du compilateur et les optimisations utilisées.

Plusieurs outils d'analyse de constant-time ciblent le code source [3] ou le bytecode LLVM [4], laissant la porte ouverte aux vulnérabilités introduites par le compilateur. Les outils d'analyse pour constant-time ciblant le binaire se basent soit sur des approches dynamiques [5], [6] qui peuvent trouver des bugs mais sont incomplètes ; soit sur des approches statiques [7] qui peuvent garantir qu'un programme est constant-time mais ne peuvent reporter de bugs précis.

Il manque donc un outil efficace d'analyse de constant-time au niveau binaire qui puisse à la fois trouver des bugs ou garantir leur absence.

Défis de la vérification de constant-time.

- Les outils de vérification standards ne s'appliquent pas directement car constant-time est une propriété de 2-hypersûreté [8] (i.e. reliant deux exécutions). Ces propriétés peuvent être réduites à des propriétés de sûreté sur un programme transformé via *auto-composition* [9], mais cette réduction est inefficace [10].
- Par ailleurs, il est connu que l'adaptation des méthodes formelles à l'analyse de binaire est complexe, notamment à cause de la perte de structure au niveau des données et du flot de contrôle et de la représentation explicite de la mémoire sous forme d'un large tableau d'octets.

L'exécution symbolique pour la 2-hypersûreté.
L'*exécution symbolique* (SE) [11] est une technique d'analyse qui permet à la fois de trouver des bugs ou de faire de la vérification bornée et qui a fait ses preuves en analyse de binaire. En revanche, les tentatives d'adaptation de cette dernière à des propriétés de 2-hypersûreté ne passent pas à l'échelle [12].

Une idée émergente consiste à représenter *deux traces d'exécution dans la même instance d'exécution symbolique* en maximisant le partage entre ces deux exécutions. Cette idée fut introduite sous le terme *ShadowSE* [13] dans le contexte du test de version, puis reprise dans le contexte de la vérification de 2-hypersûreté sous le terme *exécution symbolique relationnelle* [14].

L'exécution symbolique relationnelle.
L'exécution symbolique relationnelle (RelSE) [14] permet de modéliser deux traces d'exécution dans la même instance d'exécution symbolique en associant chaque variable soit à une *paire d'expressions symboliques* $\langle \varphi \,|\, \varphi' \rangle$ quand celle-ci *peut dépendre* des secrets ou alors à une *expression simple* $\langle \varphi \rangle$ quand celle-ci *ne dépend*

pas des secrets. Pour analyser constant-time, une requête est envoyée au solveur à chaque branchement ou accès mémoire dépendant d'une paire d'expressions afin de s'assurer que celle-ci ne dépend pas des secrets. En revanche, dans le cas d'une expression simple, l'appel au solveur n'est pas nécessaire puisque, par définition, celle-ci ne dépend pas des secrets. RelSE permet ainsi de partager les expressions similaires dans les deux exécutions et de réduire les appels au solveur en traçant les dépendances aux secrets.

Malheureusement, l'adaptation directe de RelSE ne passe pas à l'échelle dans le contexte de l'analyse de binaire. La mémoire, représentée sous forme d'un large tableau d'octets, est dupliqué dès le début de l'exécution symbolique, empêchant le partage entre les deux exécutions et le traçage des dépendances aux secrets.

Proposition. Nous proposons une technique basée sur l'exécution symbolique relationnelle qui permet de trouver des bugs et de faire de la vérification bornée efficace de constant-time au niveau binaire.

Contributions. Nos contributions sont les suivantes :

- Nous concevons trois *optimisations* pour RelSE dédiées à l'analyse de constant-time au niveau binaire : (1) *FlyRow*, variante à la volée du *read-over-write* [15], permet d'éviter de recourir à la mémoire dupliquée en résolvant les lectures en avant du solveur. (2) *Untainting* permet de transformer les paires d'expressions jugées égales par le solveur en expressions simples, favorisant le partage entre les exécutions. (3) *Fault-packing* permet de regrouper les requêtes pour un même bloc de base afin de limiter le nombre d'appels au solveur.

  De plus, nous prouvons que notre analyse est correcte pour la recherche de bugs et la vérification bornée de constant-time. Plus précisément, si l'analyse *trouve un bug* alors il existe deux traces d'exécution du programme qui, avec les mêmes entrées publiques, produisent un flot de contrôle ou des accès mémoire différents. Par ailleurs, si l'analyse *ne trouve pas de bug* jusqu'à une certaine borne, alors le programme est constant-time jusqu'à cette borne. Dans le cadre de primitives cryptographiques, cela revient à effectuer la vérification pour une taille d'entrée bornée.

- Nous proposons BINSEC/REL, le premier outil d'analyse automatique efficace pour la recherche de bugs et la vérification bornée de constant-time. Il est agnostique quant au compilateur, peut être utilisé pour des architectures x86 et ARM et ne requiert pas le code source. BINSEC/REL peut analyser 23 millions d'instructions en 98 minutes (3860 instructions par seconde) [1] tout en étant correct et complet pour constant-time.

  Nous l'évaluons sur en échantillon de 338 binaires cryptographiques et montrons qu'il peut trouver des bugs ou vérifier une implémentation 700 fois

1. Évaluation réalisée sur un ordinateur portable avec un processeur Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz et 32GB de RAM, avec Linux Mint 18.3 Sylvia

plus vite que RelSE, rendant possible l'analyse de vraies primitives cryptographiques.

- Parmi notre échantillon de 338 binaires cryptographiques, 296 respectent constant-time et les 42 autres comportent des bugs (connus).

  Nous donnons de nouvelles preuves bornées de constant-time au niveau binaire pour les 296 programmes constant-time, précédemment vérifiés à plus haut-niveau (e.g. HACL*, BearSSL, NaCL). Plus précisément, pour une taille d'entrée donnée, BINSEC/REL est capable d'explorer exhaustivement ces programmes sans trouver de violations.

  Pour les 42 autres programmes, BINSEC/REL est capable de retrouver les bugs connus (e.g. Lucky13) et de fournir des contre-exemples.

- Nous étendons une étude existante sur la préservation de constant-time par le compilateur `clang` [2] pour des binarires x86. Nous passons d'une analyse manuelle à une analyse automatique. Nous l'étendons pour prendre en compte 29 nouvelles fonctions, le compilateur `gcc`, une nouvelle versions de `clang` ainsi que des binaires ARM.

  Nous avons découvert que `gcc -O0` et les passes finales de `clang` peuvent introduire des vulnérabilités hors de portée des outils de vérification pour LLVM comme `ct-verif` [4]. Ce dernier point souligne l'importance de développer des outils de vérification pour constant-time au niveau binaire.

Références

[1] L.-A. DANIEL, S. BARDIN et T. REZK, "Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level", in *2020 IEEE Symposium on Security and Privacy (SP)*.

[2] L. SIMON, D. CHISNALL et R. J. ANDERSON, "What You Get is What You C: Controlling Side Effects in Mainstream C Compilers", in *EuroS&P*, 2018.

[3] S. BLAZY, D. PICHARDIE et A. TRIEU, "Verifying Constant-Time Implementations by Abstract Interpretation", in *ESORICS*, 2017.

[4] J. B. ALMEIDA, M. BARBOSA, G. BARTHE, F. DUPRESSOIR et M. EMMI, "Verifying Constant-Time Implementations.", in *USENIX*, 2016.

[5] S. WANG, P. WANG, X. LIU, D. ZHANG et D. WU, "CacheD: Identifying Cache-Based Timing Channels in Production Software", in *USENIX*, 2017.

[6] J. WICHELMANN, A. MOGHIMI, T. EISENBARTH et B. SUNAR, "MicroWalk: A Framework for Finding Side Channels in Binaries", in *ACSAC*, 2018.

[7] G. DOYCHEV, B. KÖPF, L. MAUBORGNE et J. REINEKE, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels", *ACM Transactions on Information and System Security*, t. 18, n° 1, 2015.

[8] M. R. CLARKSON et F. B. SCHNEIDER, “Hyperproperties”, *Journal of Computer Security*, t. 18, n° 6, 2010.

[9] G. BARTHE, P. R. D’ARGENIO et T. REZK, “Secure Information Flow by Self-Composition”, in *CSFW*, 2004.

[10] T. TERAUCHI et A. AIKEN, “Secure information flow as a safety problem”, in *SAS*, 2005.

[11] J. C. KING, “Symbolic execution and program testing”, *Commun. ACM*, t. 19, n° 7, 1976.

[12] D. MILUSHEV, W. BECK et D. CLARKE, “Noninterference via symbolic execution”, in *Formal Techniques for Distributed Systems*, 2012.

[13] H. PALIKAREVA, T. KUCHTA et C. CADAR, “Shadow of a doubt: testing for divergences between software versions”, in *ICSE*, 2016.

[14] G. P. FARINA, S. CHONG et M. GABOARDI, “Relational Symbolic Execution”, in *PPDP*, 2019.

[15] B. FARINIER, R. DAVID, S. BARDIN et M. LEMERRE, “Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing”, in *LPAR*, 2018.