Design de cryptographie white-box : et à la fin, c'est Kerckhoffs qui gagne

Joppe W. Bos[1], Charles Hubain[2⋆], Wil Michiels[1] et Philippe Teuwen[2⋆]
joppe.bos@nxp.com
chubain@quarkslab.com
wil.michiels@nxp.com
pteuwen@quarkslab.com

[1] NXP Semiconductors
[2] Quarkslab

Résumé. Bien que toutes les tentatives académiques actuelles pour créer des primitives cryptographiques standard en «white-box» aient été cassées, il y a encore un grand nombre d'entreprises qui vendent des solutions «sécurisées» de cryptographie white-box.
Afin d'évaluer le niveau de sécurité de solutions en boîte blanche, nous présentons une nouvelle approche qui ne nécessite ni connaissance des tables internes ni effort de rétro-ingénierie. Cette attaque par analyse différentielle de calcul (differential computation analysis - DCA) est la contrepartie logicielle de l'attaque différentielle de la consommation (DPA) bien connue de la communauté de cryptographie matérielle.
Nous avons développé des greffons pour des outils populaires d'instrumentation binaire dynamique afin de produire des traces d'exécution qui enregistrent les adresses mémoire et les données qui sont utilisées. Pour illustrer l'efficacité de cette attaque, nous montrons comment la DCA parvient à extraire les clés secrètes de nombreuses implémentations (non commerciales) publiquement disponibles d'algorithmes de chiffrement standard en «white-box» en analysant ces traces afin d'identifier les éventuelles corrélations. Cette approche permet d'extraire la clé d'une «white-box» nettement plus rapidement et sans connaissances spécifiques des détails de sa conception, et cela de manière automatisée : réduisant largement la connaissance et l'effort requis par rapport aux techniques mathématiques de cryptanalyse utilisées dans les attaques précédentes.

1 Introduction

L'usage de plus en plus répandu d'ordiphones ou terminaux mobiles permet désormais aux utilisateurs d'accéder à une large gamme de services

⋆ Les résultats présentés ont été obtenus lorsque ces auteurs étaient respectivement stagiaire et employé à l'Innovation Center Crypto & Security de NXP Semiconductors. Le présent article est la traduction par ces deux auteurs de l'article [14].

omniprésents dans la société moderne. Cela fait de telles plateformes des cibles de grande valeur (cf. [56] pour une étude sur la sécurité des appareils mobiles). Il existe de nombreuses techniques pour protéger les clés cryptographiques stockées dans ces appareils mobiles. Les solutions vont des implémentations logicielles sans aucune protection aux implémentations matérielles résistantes aux attaques physiques. Une des approches les plus populaires, connue sous le nom de *white-box*, essaye de cacher une clé cryptographique au sein même d'une implémentation logicielle.

Le modèle de sécurité traditionnellement utilisé pour modéliser l'interaction avec les primitives cryptographiques est dit «black-box». Dans ce modèle d'attaque «black-box», le design interne est supposé infaillible et seules les entrées et sorties du modèle sont considérées dans l'évaluation de sécurité. Comme cela a été montré par Kocher, Jaffe et Jun [37] dans la fin des années nonante, cette hypothèse est en fait fausse dans de nombreux scénarios. En effet, cette «black-box» peut laisser échapper des méta-informations via par exemple les temps de calcul ou la consommation électrique. Cette analyse des canaux auxiliaires a donné naissance au modèle d'attaque dit «gray-box». Depuis lors, l'usage et l'accès aux clés cryptographiques ont beaucoup changé, ce qui a forcé à réévaluer ce modèle de sécurité. Dans deux articles datés de 2002, Chow, Eisen, Johnson et van Oorschot ont introduit le modèle d'attaque «white-box» ainsi que des techniques pour tenter de réaliser une implémentation «white-box» d'algorithmes de chiffrement symétriques [20, 21].

L'idée principale du modèle d'attaque «white-box» est que l'adversaire est potentiellement le propriétaire du matériel sur lequel tourne l'implémentation logicielle. Il est donc supposé que l'adversaire a un contrôle complet de l'environnement d'exécution. Cela permet à l'adversaire d'effectuer une analyse statique du logiciel, mais aussi d'inspecter et de modifier la mémoire et donc de modifier des résultats intermédiaires (de manière similaire aux attaques matérielles par injection de faute). Ce modèle d'attaque «white-box» où l'adversaire possède des capacités avancées est réaliste dans le cas courant de plateformes mobiles contenant des clés de tiers. Une implémentation «white-box» peut être utilisée pour protéger l'installation d'applications sur un appareil mobile (venant par exemple d'un marché d'applications en ligne), pour chiffrer du contenu dans le cadre de la gestion numérique des droits (DRM), pour protéger une implémentation de *Host Card Emulation* (HCE) qui émule une carte bancaire ou encore sécuriser les clés d'authentification à un service du «cloud». Une implémentation «white-box» *parfaite* d'une primitive cryptographique signifie qu'un attaquant est incapable, même en ayant accès aux détails

internes de l'implémentation, de déduire la moindre information à propos de la clé secrète utilisée. Cette situation deviendrait alors équivalente à être face à un dispositif accessible uniquement en «black-box». Comme observé par [24], cela implique qu'une implémentation «white-box» devrait être capable de résister à toutes les attaques par canaux auxiliaires existantes et futures.

Comme expliqué dans [20], « *When the attacker has internal information about a cryptographic implementation, choice of implementation is the sole remaining line of defense.* »[3]. C'est exactement cette idée qui est exploitée dans une implémentation «white-box» : la clé secrète est incorporée à l'intérieur de l'implémentation des opérations cryptographiques de manière à rendre l'extraction de la clé difficile même en ayant le code source à disposition. Il est important de noter que cette approche est différente de contre-mesures contre la rétro-ingénierie telles que l'obscurcissement de code [8,41] et l'obscurcissement de structures de contrôle [30], bien que celles-ci soient régulièrement utilisées en complément à une implémentation «white-box» comme seconde ligne de défense. Bien qu'il soit conjecturé qu'aucune défense contre les attaques sur des implémentations «white-box» n'existe sur le long terme [20], il y a tout de même un nombre significatif de sociétés qui vendent des solutions d'implémentation «white-box» sécurisées. Il est important de noter qu'il n'existe quasi pas de résultat public sur l'implémentation «white-box» d'un algorithme cryptographique standard à clé publique hormis un brevet de Zhou et Chow proposé en 2002 [72]. Les autres techniques d'implémentation «white-box» se concentrent uniquement sur la cryptographie symétrique. De plus, toutes ces techniques ont été cassées théoriquement (cf. Section 2 pour un aperçu). Un désavantage des attaques publiées est qu'elles exigent une connaissance détaillée de l'architecture de l'implémentation «white-box» à attaquer. Par exemple l'emplacement exact des S-boxes ou encore des transitions entre tours ainsi que le format des encodages appliqués aux tables de correspondance (cf. Section 2 sur comment les implémentations «white-box» sont généralement construites). Les vendeurs d'implémentations «white-box» essayent généralement d'éviter ces attaques en ignorant le principe de Kerckhoffs et en gardant les détails de leur implémentation secrets (et en changeant de design si jamais celui-ci est cassé).

Notre contribution. Toutes les approches cryptanalytiques actuelles exigent une connaissance détaillée du design utilisé, par exemple l'em-

[3] *Lorsque l'attaquant a des informations internes sur une implémentation cryptographique, le choix de cette implémentation est la seule ligne de défense qui nous reste.*

placement et l'ordre des S-boxes appliquées ainsi qu'où et comment les encodages sont utilisés. L'effort d'analyse nécessaire pour obtenir ces informations est un aspect important de la valeur attribuée aux solutions de «white-box» commerciales. Les vendeurs sont au courant que leurs solutions n'offrent pas de défense sur le long terme, mais compensent ce problème par exemple en mettant à jour régulièrement leur logiciel. Notre contribution est une attaque qui fonctionne de manière automatique et qui remet donc en question le niveau de sécurité supposé de ces solutions commerciales.

Dans cet article nous utilisons de l'analyse binaire dynamique (DBA), une technique souvent utilisée pour inspecter et améliorer la qualité d'implémentations logicielles, pour accéder et contrôler l'état intermédiaire d'une implémentation «white-box». Une approche pour effectuer de la DBA est l'instrumentation binaire dynamique (DBI). L'idée est d'insérer du code d'analyse additionnel dans le code original du programme durant l'exécution dans le but de déboguer les accès mémoires, détecter les fuites mémoires ou profiler le code. Les outils de DBI les plus avancés tels que Valgrind [53] et Pin [42] permettent d'observer et de modifier des instructions dans un binaire en cours d'exécution. Ces outils ont déjà démontré leur potentiel pour l'analyse comportementale de code obscurci [61].

Nous avons développé des greffons pour Valgrind et Pin qui permettent d'obtenir des *traces d'exécution* qui contiennent les instructions exécutées ainsi que les accès mémoire en lecture et écriture[4]. Ces traces d'exécution sont utilisées pour déduire des informations à propos de la clé secrète incorporée dans une implémentation «white-box» en corrélant des hypothèses sur certaines valeurs intermédiaires avec les valeurs contenues dans les traces. Pour ce faire nous introduisons *l'analyse différentielle des calculs* (*Differential Computation Analysis* ou DCA) qui peut être vue comme la contrepartie logicielle de l'analyse différentielle de la consommation (DPA) [37] utilisée par la communauté de cryptographie matérielle. Il y a cependant d'importantes différences entre ces traces d'exécution et des traces de consommation comme expliqué dans la Section 4.

Dans cet article nous démontrons que la DCA peut être utilisée efficacement pour extraire les clés secrètes d'une implémentation «white-box». Nous appliquons ensuite la DCA aux challenges de «white-box» publics que nous avons pu trouver ; soit l'extraction des clés secrètes de quatre implémentations «white-box» d'AES et de DES. Par rapport aux attaques

[4] Ces outils ont été publiés sous licence libre à l'adresse suivante : `https://github.com/SideChannelMarvels`

cryptanalytiques existantes sur les implémentations «white-box», cette technique ne requiert aucune connaissance quant à la stratégie d'implémentation utilisée, elle peut être utilisée de manière automatique sans nécessiter de connaissance en cryptographie et elle extrait la clé plus rapidement. Outre les détails cryptanalytiques, nous discutons de techniques qui pourraient être utilisées comme contre-mesures contre la DCA (cf. Section 6).

Indépendamment et après que [14] soit apparu en ligne, Sanfelix, de Haas et Mune ont également présenté des attaques sur des implémentations «white-box» [58]. Ils ont d'une part confirmé nos résultats et d'autre part considéré les attaques par injection de faute.

Organisation de cet article. Dans la Section 2 nous rappelons les techniques utilisées pour transformer un algorithme de cryptographie symétrique en implémentation «white-box». Ensuite, nous résumons la littérature sur les tentatives de construction d'implémentations «white-box» et les attaques contre celles-ci. La Section 3 rappelle les idées à la base des attaques par analyse différentielle de la consommation électrique. Dans la Section 4 nous introduisons le concept de traces d'exécution, leur obtention, et l'exécution d'une attaque par analyse différentielle des calculs sur ces traces. La Section 5 résume comment nous avons utilisé avec succès les attaques par analyse différentielle des calculs sur plusieurs challenges publics de «white-box». Les premières étapes vers le développement de contre-mesures sont discutées dans la Section 6 et la Section 7 conclut cet article avec quelques idées à explorer pour une recherche future.

2 Aperçu de quelques techniques de cryptographie «white-box»

Comme discuté dans l'introduction, le modèle d'attaque «white-box» permet à l'adversaire de prendre un contrôle complet de l'implémentation cryptographique et de l'environnement d'exécution. Il n'est pas surprenant qu'avec un adversaire possédant de telles capacités, les auteurs de l'article original sur les implémentations «white-box» [20] conjecturassent qu'aucune défense sur le long terme n'existe contre les attaques sur des implémentations «white-box». Cette conjecture doit être comprise dans le contexte de l'obscurcissement de code, car cacher une clé cryptographique au sein d'une implémentation est une sorte d'obscurcissement de code. Il a été prouvé que l'obscurcissement complet de tout programme, quel qu'il soit, est impossible [5]. Cependant, il n'est pas prouvé que ce résultat s'applique à un sous-domaine spécifique de fonctionnalités «white-box».

De plus, ce résultat doit être analysé à la lumière des évolutions récentes où des techniques utilisant des *multilinear maps* et mises en œuvre pour réaliser l'obscurcissement peuvent fournir des garanties de sécurité significatives (cf. [4, 15, 27]). Pour se protéger à moyen ou long terme dans le contexte de ce modèle de sécurité, il est nécessaire de mettre à profit les avantages des solutions purement logicielles. L'idée est d'utiliser le concept de *vieillissement logiciel* [32] : celui-ci force, à intervalles réguliers, une mise à jour de l'implémentation «white-box». Si cet intervalle est suffisamment court, l'adversaire ne dispose pas du temps de calcul nécessaire pour extraire la clé secrète de l'implémentation «white-box». Cette approche n'a du sens que si les données sensibles n'ont un intérêt que sur le court terme, par exemple dans le cadre de la protection de gestion numérique des droits appliquée à un match de football diffusé en direct. Cependant, la difficulté pratique d'imposer ces mises à jour sur des appareils avec une connexion à l'internet intermittente doit être prise en compte.

Encodage externe. Outre l'objectif principal qui est de cacher la clé, une implémentation «white-box» peut apporter d'autres fonctionnalités comme le marquage d'une clé avec une empreinte permettant de tracer une éventuelle fuite ou encore la protection d'un logiciel contre des modifications [47]. Il y a cependant d'autres problèmes de sécurité à considérer outre l'extraction de la clé de l'implémentation «white-box». Si l'intégralité de l'implémentation «white-box» peut être extraite et copiée sur un autre appareil, alors l'intégralité des fonctionnalités «white-box» a elle aussi, été copiée car la clé secrète est contenue à l'intérieur du logiciel. Cette attaque est appelée *code lifting*. Une solution à ce problème est d'utiliser des encodages externes [20]. Si l'on suppose que la fonctionnalité cryptographique E_k fait partie d'un écosystème plus grand alors on peut implémenter $E'_k = G \circ E_k \circ F^{-1}$ à la place. Les encodages d'entrée (F) et de sortie (G) sont des bijections choisies au hasard de telle sorte que l'extraction de E'_k ne permette pas à un adversaire de calculer E_k. L'écosystème qui utilise E'_k doit s'assurer que les encodages d'entrée et de sortie sont annulés. En pratique, en fonction de l'application, les encodages d'entrée ou de sortie doivent être effectués localement par le logiciel appelant E'_k. Par exemple dans les applications de gestion numérique des droits, le serveur peut s'occuper de l'encodage d'entrée à distance, mais le client doit annuler lui-même l'encodage de sortie pour obtenir le clair final.

Dans cet article, nous attaquons avec succès des implémentations qui appliquent au plus un seul encodage externe à distance (soit sur l'entrée, soit sur la sortie). Quand à la fois l'entrée arrive avec un encodage externe appliqué à distance et que la sortie calculée est aussi encodée avec un

encodage externe alors l'implémentation n'est plus à proprement parler une implémentation «white-box» d'un algorithme standard, comme AES ou DES, mais une version modifiée de l'algorithme $G \circ \mathrm{AES} \circ F^{-1}$ ou $G \circ \mathrm{DES} \circ F^{-1}$.

Idée générale. L'approche générale pour implémenter une «white-box» est présentée dans [20]. L'idée est d'utiliser des tables de correspondance plutôt que des étapes de calcul individuelles pour implémenter l'algorithme et d'encoder ces tables avec des bijections choisies aléatoirement. L'utilisation d'une clé secrète particulière est incorporée dans ces tables. À cause de l'utilisation intensive de tables de correspondance, les implémentations «white-box» sont typiquement plus larges, voire plus lentes, de plusieurs ordres de magnitude qu'une implémentation standard (non «white-box») du même algorithme. Il est courant d'écrire un logiciel qui génère automatiquement une implémentation «white-box» aléatoire à partir d'un algorithme et d'une clé secrète fixée donnés en entrée. L'aléa se retrouve dans les bijections choisies au hasard pour cacher l'utilisation de la clé secrète au sein des différentes tables de correspondance.

Dans le reste de cette section, nous rappelons brièvement les principes de DES et AES, les deux algorithmes les plus couramment choisis pour les implémentations «white-box», avant de résumer la littérature scientifique liée aux techniques «white-box».

Data Encryption Standard (DES). DES est un algorithme de chiffrement symétrique publié en tant que «Federal Information Processing Standard» (FIPS) pour les États-Unis en 1979 [67]. Pour la suite de cet article, il est suffisant de savoir que DES est un algorithme de chiffrement itératif qui consiste en 16 tours identiques et un schéma entrecroisé connu sous le nom de réseau de Feistel. DES peut être implémenté en travaillant uniquement avec des valeurs encodées sur 8 bits (un seul octet) et en utilisant principalement des opérations simples comme la rotation, le ou-exclusif et des tables de correspondance (S-boxes). Les attaques par force brute sur DES étant préoccupantes, l'usage du triple DES, qui applique DES trois fois sur chaque bloc, a été ajouté aux versions suivantes du standard [67].

Advanced Encryption Standard (AES). Pour sélectionner un successeur à DES, le NIST a organisé une compétition publique ouverte appelant à de nouveaux designs. En 2000, après approximativement trois ans de compétition, l'algorithme de chiffrement Rijndael fut choisi pour devenir l'AES [1, 22] : un algorithme non classifié et publiquement disponible de chiffrement symétrique par bloc. Les opérations utilisées par AES sont, comme pour DES, relativement simples : ou-exclusif, multiplications

par des éléments d'un champ fini de 2^8 éléments et des tables de correspondance (S-boxes). Rijndael a été conçu pour être performant sur des architectures 8 bits et il est donc facile d'écrire une implémentation opérant par octets. L'AES est disponible en trois niveaux de sécurité. Par exemple l'AES-128 utilise une clé de 128 bits et 10 tours pour calculer le résultat chiffré à partir du clair.

2.1 Résultats d'implémentation white-box

White-Box Data Encryption Standard (WB-DES). La première publication tentant de construire une implémentation de WB-DES date de 2002 [21]. Une approche y est discutée pour construire des implémentations «white-box» pour des algorithmes de chiffrement basés sur les réseaux de Feistel. Une première attaque sur ce schéma, qui permet de défaire les mécanismes d'obscurcissement de la clé, fut publiée la même année et utilise l'injection de fautes [31] pour extraire la clé secrète en observant comment le logiciel échoue suite à l'injection d'une certaine erreur. En 2005 une version améliorée du design WB-DES résistante à cette injection de fautes fut présentée dans [40]. Cependant en 2007 deux attaques par cryptanalyse [9] furent publiées et sont capables d'extraire la clé secrète de ce type de «white-box» [28, 69]. Cette dernière méthode a une complexité en temps de seulement 2^{14}.

White-Box Advanced Encryption Standard (WB-AES). La première approche pour réaliser une implémentation de WB-AES fut proposée en 2002 [20]. En 2004 les auteurs de [11] présentèrent comment l'information sur l'encodage intégré aux tables de correspondance peut être révélée en analysant la composition des tables de correspondance. Cette approche est connue sous le nom d'attaque BGE et permet d'extraire la clé de ce type de WB-AES avec une complexité en temps de 2^{30}. Une version de ce design de WB-AES qui introduit des perturbations dans l'algorithme de chiffrement pour essayer de contrecarrer l'attaque précédente fut présentée dans [70] en 2009 mais fut cassée en 2012 avec une complexité en temps de 2^{32} [52].

Une autre approche intéressante est l'utilisation d'une structure algébrique différente pour la même instance d'un chiffrement itératif par bloc (comme proposé à l'origine dans [10]). Cette approche [33] utilise deux algorithmes de chiffrement pour modifier la représentation de l'état et de la clé à chaque tour ainsi que deux des quatre opérations classiques d'AES. Cette approche fut démontrée comme équivalente à la première

implémentation de WB-AES [20] dans [38] en 2013. De plus, les auteurs de [38] utilisèrent un résultat de 2012 [66] qui améliore la phase la plus gourmande en ressources de l'attaque BGE. Cela réduit le coût de l'attaque BGE à une complexité en temps de 2^{22}. Une attaque indépendante, de la même complexité en temps, est aussi présentée dans [38].

Autres résultats sur les «white-boxes». Les résultats présentés ci-dessus s'intéressent uniquement à la construction et la cryptanalyse d'implémentations WB-DES et WB-AES. Les techniques «white-box» ont été également étudiées et utilisées dans un contexte plus étendu. En 2007 les auteurs de [48] présentèrent une technique «white-box» pour rendre un code résistant aux modifications. En 2008 les résultats de cryptanalyse sur WB-DES et WB-AES furent généralisés à tout chiffrement de type réseau Substitution - Transformation Linéaire (chiffrements SLT) [49]. Ce travail fut encore plus généralisé et un outil générique d'analyse capable d'extraire une clé secrète d'un chiffrement SLT générique est présenté dans [3]. Des notions formelles de sécurité pour les schémas de chiffrement symétrique «white-box» sont discutées dans [24, 59]. Dans [12] il est démontré qu'il est possible d'utiliser une construction ASASA avec des S-boxes injectives (où ASA représente une construction Affine-Substitution-Affine [55]) pour implémenter des primitives cryptographiques «white-box». Un tutoriel concernant la WB-AES est disponible dans [51].

2.2 Prérequis des attaques existantes

Pour mettre nos résultats en perspective, il est utile de spécifier exactement les prérequis nécessaires pour appliquer les attaques «white-box» de la littérature scientifique. Ces approches nécessitent au moins une connaissance sommaire du schéma qui est implémenté en «white-box». Plus précisément, l'adversaire doit découvrir :

- quel type d'encodage est appliqué sur les *résultats intermédiaires.*
- quelle *opération cryptographique* est implémentée par quel *réseau de tables de correspondance.*

Le problème de ces prérequis est que les vendeurs d'implémentation «white-box» sont généralement peu enclins à partager les informations relatives à leur schéma de «white-box» (la bien nommée «sécurité par obscurité»). Si cette information n'est pas directement accessible et si seul un exécutable ou une bibliothèque binaire est disponible, il est nécessaire

d'investir une quantité significative de temps dans la rétro-ingénierie manuelle du binaire. Défaire ces couches d'obscurcissement avant de pouvoir récupérer l'information sur l'implémentation, information nécessaire pour appliquer avec succès ce type d'attaque, peut prendre une quantité considérable de temps. Cet effort additionnel qui demande un niveau élevé d'expertise et d'expérience est illustré par les méthodes sophistiquées utilisées dans les solutions des challenges publics de «white-box» comme cela est détaillé dans la Section 5. L'approche de l'analyse différentielle des calculs que nous détaillons dans la Section 4 ne requiert pas de défaire les couches d'obscurcissement ou de rétro-concevoir un exécutable binaire.

3 Analyse différentielle de la consommation

Depuis la fin du XX^{e} siècle, il est publiquement connu que l'analyse statistique d'une trace de consommation électrique obtenue lors de l'exécution d'une primitive cryptographique peut corréler avec la clé secrète utilisée [37], et donc révéler des informations sur cette clé. De manière générale, on suppose avoir accès à l'implémentation matérielle d'un algorithme cryptographique connu. $I(p_i, k)$ dénote un état intermédiaire ciblé de l'algorithme ayant en entrée p_i et où seule une faible portion de la clé secrète, notée k, est utilisée dans le calcul. Si l'on assume que la consommation électrique de l'appareil à l'état $I(p_i, k)$ est la somme d'une composante dépendante des données et de bruit aléatoire, c.-à-d. $\mathcal{L}(I(p_i, k)) + \delta$ où la fonction $\mathcal{L}(s)$ retourne la consommation électrique de l'appareil durant l'état s et δ dénote un bruit de courant de fuite. Il est courant d'assumer (cf. par exemple [44]) que ce bruit est aléatoire, indépendant de l'état intermédiaire et suit une distribution normale avec une moyenne nulle. Puisque l'adversaire a accès à l'implémentation matérielle, il peut obtenir des triplets de valeurs (t_i, p_i, c_i). Ici p_i est l'entrée en clair choisie au hasard par l'adversaire, c_i est la sortie chiffrée calculée par l'implémentation matérielle utilisant une clé fixe inconnue et t_i indique la consommation électrique durant le calcul réalisé par l'implémentation matérielle. La consommation électrique $\mathcal{L}(I(p_i, k)) + \delta$ est juste une petite fraction de la trace de consommation complète t_i.

Le but de l'attaquant est d'obtenir une partie de la clé k en comparant la consommation électrique t_i réelle de l'appareil avec une estimation de la consommation électrique sous toutes les hypothèses possibles pour k. L'idée derrière une attaque par analyse différentielle de la consommation [37] (cf. [36] pour une introduction sur le sujet) est de diviser les traces mesurées en deux ensembles distincts selon une propriété définie. Par exemple cette

propriété peut être la valeur d'un des bits de l'état intermédiaire $I(p_i, k)$. On peut supposer — et ce fait est confirmé en pratique par les mesures sur des implémentations matérielles non protégées — que la distribution de la consommation électrique pour ces deux ensembles est différente (c.-à-d. qu'ils ont une moyenne et une déviation standard différentes).

Pour obtenir une information à propos d'une partie de la clé secrète k, pour chaque trace t_i et entrée p_i, il est nécessaire d'énumérer toutes les valeurs possibles de k (typiquement $2^8 = 256$ lorsque l'on attaque un octet de la clé), de calculer les valeurs intermédiaires $g_i = I(p_i, k)$ pour chaque hypothèse sur la clé et de séparer les traces t_i en deux ensembles en fonction de cette propriété mesurée sur g_i. Si l'hypothèse sur k est correcte alors la différence entre les moyennes des deux ensembles va converger vers la différence de la moyenne des distributions. Cependant, si l'hypothèse sur la clé est fausse alors les données dans chaque ensemble peuvent être vues comme le résultat d'un tri aléatoire des mesures et la différence des moyennes devrait converger vers zéro. Cela permet de déterminer quelle hypothèse est correcte, si suffisamment de traces sont disponibles. Le nombre de traces nécessaires dépend, entre autres, du bruit de mesure et de la moyenne des distributions (ce qui dépend de la plateforme considérée).

Bien que disposer de la sortie chiffrée soit utile pour valider la clé extraite, cela n'est pas strictement nécessaire. Inversement, il est possible d'attaquer une implémentation où seule la sortie chiffrée est disponible en ciblant les valeurs de l'état intermédiaire du dernier tour. Les mêmes attaques s'appliquent de manière évidente à l'opération de déchiffrement.

La même technique peut aussi être appliquée sur des traces contenant des informations d'autres types de canaux auxiliaires comme le rayonnement électromagnétique de l'appareil. Bien que nous nous concentrions sur l'analyse différentielle de la consommation dans cet article, il est important de noter qu'il existe d'autres attaques plus avancées et plus efficaces. Cela inclut entre autres les attaques d'ordre algébrique plus élevé [46], l'analyse par corrélation de la consommation [16] et les attaques par profilage [19].

4 Traces d'exécution logicielle

Pour auditer la sécurité d'un exécutable binaire implémentant une primitive cryptographique conçue pour être résistante selon le modèle d'attaque «white-box», il est possible d'exécuter le binaire sur un processeur de l'architecture correspondante et d'observer sa consommation électrique pour effectuer une analyse différentielle de la consommation (cf. Section 3).

Cependant, selon le modèle «white-box», il y a moyen d'être plus efficace, car le modèle implique qu'il est possible d'observer toutes les opérations, et ce, sans bruit de mesure. En pratique, une telle capacité d'observation peut être obtenue en instrumentant le binaire ou en instrumentant un émulateur en charge de l'exécution du binaire. Nous avons choisi la première approche en utilisant des suites logicielles d'instrumentation binaire dynamique (DBI). En résumé, la DBI considère le binaire exécutable à analyser comme le «bytecode» d'une machine virtuelle et utilise une technique appelée la compilation «just-in-time». Cette recompilation du code machine permet d'effectuer des transformations sur le code tout en préservant les effets originaux du code. Ces transformations sont pratiquées au niveau de chaque «basic block»[5] et sont stockées dans un cache pour accélérer l'exécution. Ce mécanisme est par exemple utilisé par QEMU (un hyperviseur libre qui effectue de la virtualisation matérielle) pour exécuter du code machine venant d'une architecture sur une autre architecture. Dans ce cas-là, la transformation effectuée est la traduction entre les deux architectures [7]. Les suites logicielles de DBI, telles que Pin [42] et Valgrind [53], effectuent d'autres types de transformations : elles permettent d'ajouter des appels vers des fonctions personnalisées entre les instructions machine grâce à l'écriture de greffons qui s'interposent dans le processus de recompilation. Ces fonctions de rappel peuvent être utilisées pour observer l'exécution du logiciel et suivre des événements spécifiques. La différence principale entre Pin et Valgrind est que Valgrind utilise une représentation intermédiaire (IR) des instructions, appelée VEX, qui est indépendante de l'architecture ce qui permet d'écrire des greffons compatibles avec toutes les architectures supportées par l'IR.

Nous avons développé des greffons pour les deux suites logicielles capables de tracer l'exécution de binaires sous les architectures x86, x86-64, ARM et ARM64 et d'enregistrer les informations désirées : c.-à-d. les adresses des accès mémoire effectués (en lecture, écriture et exécution) ainsi que leur contenu. Il est aussi possible d'enregistrer le contenu des registres du processeur, mais cela ralentirait considérablement l'acquisition des traces et augmenterait la taille des traces de manière significative. De plus nous avons réussi à extraire les clés secrètes des implémentations «white-box» sans cette information supplémentaire. Ceci n'est pas surprenant, car les implémentations «white-box» basées sur des tables

[5] Un «basic block» est une portion de code avec un point d'entrée et un point de sortie uniques. Cependant, pour des considérations techniques, la définition d'un «basic block» telle qu'utilisée par Pin et par Valgrind varie légèrement et peut inclure plusieurs points d'entrée ou de sortie.

de correspondance sont principalement constituées d'accès mémoire et utilisent très peu d'instructions arithmétiques (cf. Section 2 pour les fondamentaux du design de la plupart des implémentations «white-box»). Dans certaines configurations plus complexes, par exemple lorsque l'implémentation «white-box» est enfouie à l'intérieur d'un exécutable beaucoup plus grand, il peut être nécessaire de changer le comportement de l'exécutable en appelant directement les fonctionnalités de chiffrement ou en injectant le clair à l'intérieur d'une interface de programmation applicative (API) interne. Ceci est trivial à réaliser en utilisant un outil de DBI mais pour les cas étudiés en Section 5 nous n'avons pas eu besoin de recourir à de telles méthodes.

Première étape. Traçons une seule exécution du binaire «white-box» avec un message d'entrée arbitraire et enregistrons toutes les adresses accédées et les données correspondantes durant l'exécution. Bien que le traceur soit capable de suivre l'exécution partout, y compris dans les bibliothèques externes et systèmes, nous limitons les zones mémoires tracées à l'exécutable principal, ou à une bibliothèque si les opérations cryptographiques y sont effectuées. Une technique de sécurité informatique courante souvent activée par défaut dans les systèmes d'exploitation modernes est la distribution aléatoire de l'espace d'adressage (ASLR) qui arrange aléatoirement l'emplacement mémoire des exécutables, de leurs données, de leur pile, de leur tas ainsi que d'autres éléments comme les bibliothèques dynamiques. Pour rendre cette acquisition entièrement reproductible, nous désactivons simplement l'ASLR puisque le modèle «white-box» nous donne le contrôle sur l'environnement d'exécution. Et même si l'ASLR ne pouvait être désactivé, cela aurait seulement l'inconvénient de nous obliger à réaligner les traces.

Seconde étape. Ensuite, nous visualisons la trace pour comprendre où l'algorithme de chiffrement par blocs est utilisé et, en comptant le nombre de motifs répétés, nous déterminons quelle primitive cryptographique (standard) est implémentée : par exemple un AES-128 à 10 tours, un AES-256 à 14 tours, ou un DES à 16 tours. Pour visualiser la trace, nous avons décidé de la représenter sous forme graphique selon l'approche présentée dans [50]. La Figure 1 illustre cette approche : l'espace d'adressage virtuel est représenté sur l'axe horizontal sur lequel typiquement pour les plateformes modernes on retrouvera le segment *text* (qui contient les instructions), le segment *data*, le segment *bss* des données non initialisées, le tas et pour finir la pile, respectivement. L'espace d'adressage est extrêmement clairsemé, nous choisissons donc de ne montrer que les bandes de mémoire où il y a quelque chose à voir. L'axe vertical est un axe temporel

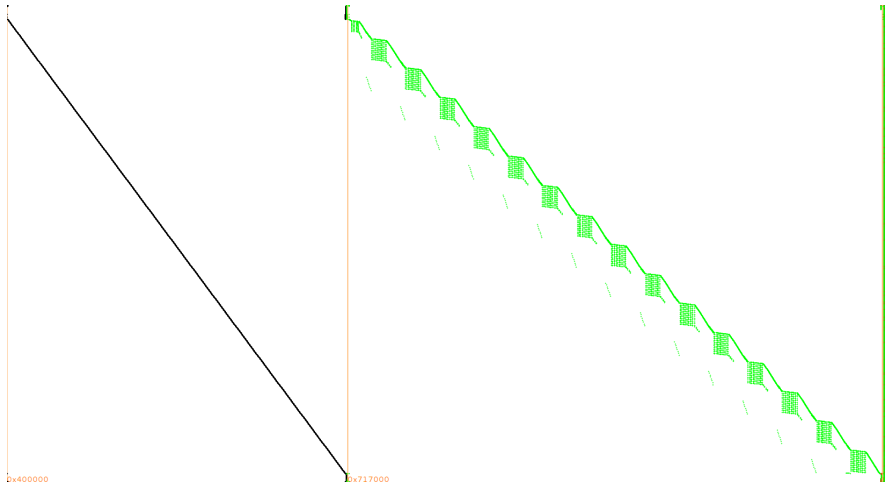

Fig. 1. Visualisation d'une trace d'exécution logicielle d'une implémentation «white-box» de DES.

allant de haut en bas. En noir on retrouve les instructions exécutées, en vert les adresses des données lues en mémoire et en rouge celles écrites en mémoire. Dans la Figure 1 on déduit que le code (en noir à gauche) a été complètement déroulé en un seul long bloc d'instructions, de nombreuses données sont lues depuis des tables en mémoire (en vert à droite) et la pile est comparativement beaucoup plus petite, de sorte que les lectures et écritures sur la pile sont à peine visibles à cette échelle, tout à droite.

Troisième étape. Une fois que nous avons déterminé quel algorithme cibler, nous gardons l'ASLR désactivé et enregistrons plusieurs traces avec des messages d'entrée choisis aléatoirement, éventuellement en utilisant des critères de filtre comme la limitation de l'espace d'adresses des instructions pour laquelle on souhaite capturer l'activité. C'est surtout utile dans le cas de binaires larges qui effectuent d'autres types d'opérations qui ne nous intéressent pas (par exemple lorsque la «white-box» est intégrée à une application plus conséquente). Si les opérations «white-box» elles-mêmes prennent un temps non négligeable, on peut limiter la portée de l'acquisition à l'enregistrement de l'activité aux alentours du premier ou du dernier tour, selon qu'on monte une attaque sur l'entrée ou la sortie de l'algorithme de chiffrement. Se concentrer sur le premier ou le dernier tour est typique des attaques du type DPA puisque cela limite la portion de clé à attaquer à un seul octet à la fois, comme expliqué dans la Section 3. Dans l'exemple donné Fig. 1, le motif des accès en lecture visible à droite rend triviale l'identification des tours du DES et l'observation des instructions en correspondance sur la gauche permet de définir un filtre sur l'espace d'adresses adapté. Tandis que nous enregistrions toute

l'information relative aux accès mémoire dans la trace initiale (lors de la première étape), nous n'enregistrons à cette étape qu'un seul type d'information (éventuellement dans un espace des adresses limité). Les exemples typiques sont notamment l'enregistrement des octets lus en mémoire, ou des octets écrits sur la pile, ou encore l'octet de poids faible des adresses des données accédées en mémoire.

Cette approche générique nous donne un bon compromis pour mener l'attaque rapidement tout en minimisant le stockage des traces d'exécution. Si l'espace de stockage n'est pas un souci, on peut sauter directement à cette troisième étape et enregistrer des traces de l'exécution complète, ce qui est parfaitement possible pour les exécutables légers, comme cela sera illustré dans les quelques exemples de la Section 5. Cette approche naïve mène la voie à la création d'un dispositif complètement automatisé d'acquisition et de récupération des clés.

Quatrième étape. À la troisième étape, nous avons obtenu une série de traces d'exécution qui consistent en des adresses (partielles) ou des valeurs qui ont été enregistrées à chaque fois qu'une instruction accédait aux données en mémoire. Pour les transformer en une représentation adéquate aux outils habituels de DPA qui sont conçus pour traiter des traces de consommation électrique, nous sérialisons ces valeurs (habituellement des octets) en vecteurs d'un et de zéro. Cette étape est essentielle pour exploiter toute l'information que nous avons capturée. Pour comprendre cela, comparons l'attaque à un dispositif matériel de DPA classique qui ciblerait le même type d'information : les transferts de et vers la mémoire physique.

Lors d'une DPA, une plateforme habituelle est par exemple un CPU avec un bus de 8 bits de large vers la mémoire et les huit lignes de ce bus basculent entre des tensions hautes ou basses pour transmettre les données. Si une fuite est observable dans les variations de la consommation électrique, ce sera une valeur analogique proportionnelle à la somme des bits égaux à 1 dans l'octet en cours de transfert sur le bus mémoire. Dans ce type de scénario, le modèle de fuite le plus élémentaire est le poids de Hamming des octets transférés entre le CPU et la mémoire. Cependant, dans notre cadre logiciel, nous connaissons la valeur exacte de chacun des huit bits et pour exploiter cette information au mieux, nous voulons attaquer chacun de ces bits individuellement, et non pas leur somme (comme dans le modèle de Hamming). C'est pourquoi l'étape de sérialisation que nous appliquons (la conversion des valeurs observées en vecteurs de un et de zéro) peut être vue dans le modèle matériel comme si chaque ligne du bus fuyait individuellement l'une après l'autre.

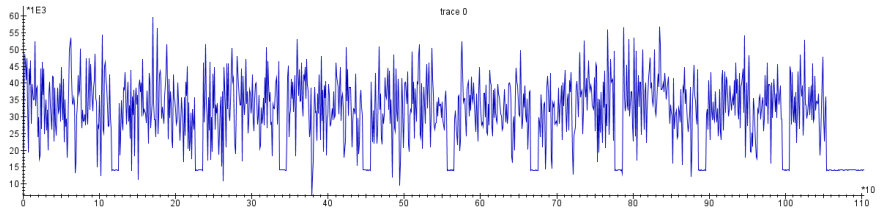

Fig. 2. Exemple typique d'une trace de consommation électrique d'une implémentation non protégée d'AES-128 (on observe facilement les dix tours).

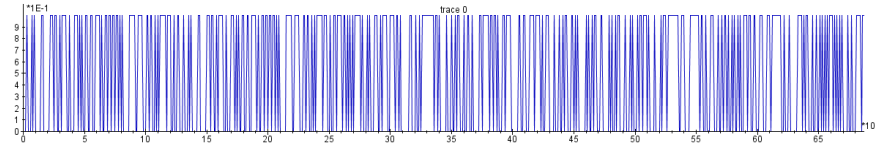

Fig. 3. Exemple typique d'une portion de trace sérialisée d'écritures sur la pile dans un WB-AES, constituée uniquement de uns et de zéros.

Lors d'une attaque par DPA, une trace de consommation électrique typique est constituée de mesures faites de valeurs analogiques échantillonnées. Dans notre cadre logiciel nous manipulons des fuites *parfaites* (c.-à-d. sans bruit de mesure) des bits individuels qui ne prennent donc que deux valeurs possibles : 0 ou 1. En conséquence, notre trace d'exécution logicielle peut être vue du point de vue matériel comme si nous étions en train de mesurer chaque piste individuelle du bus avec une aiguille, une opération requérant une préparation lourde des échantillons telle que le décapsulage des puces, le perçage et le détournement de pistes par sonde ionique focalisée (*Focused Ion Beam*, FIB) pour creuser à travers les couches métalliques et atteindre enfin les pistes du bus sans affecter le fonctionnement de la puce. C'est une situation bien plus avantageuse et invasive qu'une acquisition externe sur canaux auxiliaires.

Lorsqu'on utilise des traces d'exécution, il y a une autre différence importante par rapport aux traces de consommation habituelles sur l'axe du temps. Dans une trace de canaux auxiliaires physiques, les valeurs analogiques sont échantillonnées à des intervalles réguliers, souvent décorrélés du signal d'horloge interne au matériel attaqué, et l'axe temporel représente le temps linéairement. Avec les traces d'exécution logicielle, nous n'enregistrons l'information que lorsque cela est pertinent, par exemple chaque fois qu'un octet est écrit sur la pile, si c'est la propriété à laquelle nous nous intéressons. De plus les bits sont sérialisés comme s'ils avaient été écrits séquentiellement. On remarque qu'avec cette sérialisation et cet échantillonnage à la demande, notre axe temporel n'en est plus vraiment un. Mais une attaque par DPA ne nécessite pas de véritable axe temporel.

Tout au plus, il faut prendre garde que lorsqu'on compare deux traces, les événements qui se sont produits au même moment lors de ces deux exécutions soient bien comparés les uns aux autres. Les Figures 2 et 3 illustrent ces différences entre traces obtenues respectivement par DPA et par DCA.

Cinquième étape. Une fois les traces d'exécution acquises et mises en forme, nous pouvons utiliser les outils habituels de DPA pour extraire la clé. Nous montrons dans la section suivante à quoi ressemble le résultat d'un outil de DPA, outre la récupération de la clé.

Étape optionnelle. Si nécessaire, on peut identifier les points exacts de l'exécution où l'information utile fuite. Avec l'aide d'une analyse de *corrélation par clé connue*, il est possible de localiser précisément l'instruction «fautive» et la ligne correspondante dans le code source, si l'information de débogage est disponible. Cela peut être une aide appréciable pour le concepteur de la «white-box».

Pour conclure cette section, voici un résumé des prérequis de notre analyse différentielle des calculs, à mettre en correspondance avec les prérequis des attaques «white-box» précédentes qui ont été détaillées dans la Section 2.2 :

- Être capable d'exécuter de manière répétée (de quelques dizaines à quelques milliers de fois) le binaire dans un environnement contrôlé.
- Avoir connaissance des messages en clair (avant un éventuel encodage) ou des messages chiffrés (après un éventuel décodage).

5 Étude des implémentations «white-box» publiquement disponibles

5.1 Le challenge Wyseur

Pour autant que nous sachions, le premier challenge public de «white-box» a été créé par Brecht Wyseur en 2007. Sur son site[6] un exécutable binaire est disponible, contenant une opération de chiffrement DES en «white-box» avec une clé fixe embarquée. Selon l'auteur, cette approche WB-DES implémente les idées de [21,40] (cf. Section 2.1) avec «quelques améliorations personnelles». L'interaction avec le programme est immédiate : il prend un message en entrée et retourne le résultat chiffré sur la console. Le challenge a été résolu publiquement après cinq ans (en 2012) de manière

[6] `http://whiteboxcrypto.com/challenges.php`

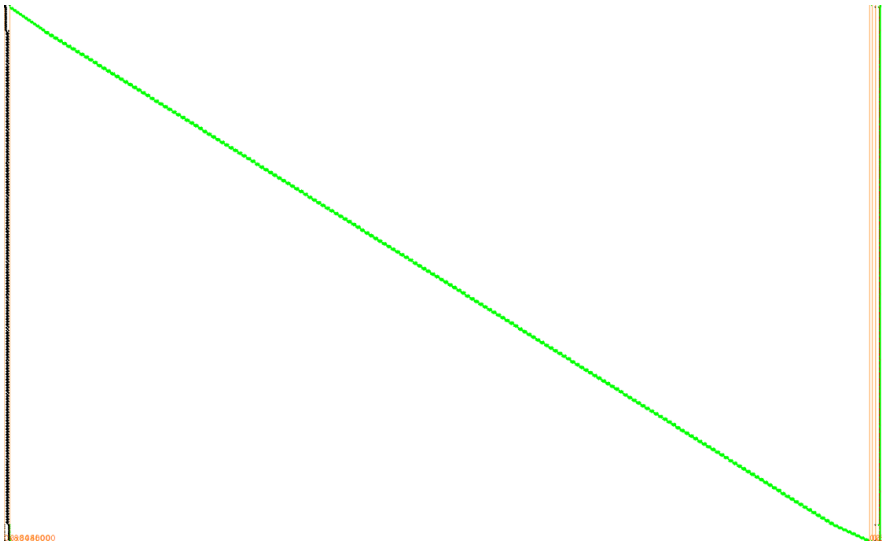

Fig. 4. Visualisation d'une trace d'exécution du binaire du challenge «white-box» de Wyseur, montrant l'ensemble des adresses mémoire accédées.

indépendante par James Muir et «SysK». Ce dernier a fourni une description détaillée [63] et a utilisé la cryptanalyse différentielle (semblable à [28, 69]) pour extraire la clé secrète embarquée.

La Figure 4 montre une trace d'exécution complète de ce challenge WB-DES. Tout à gauche on voit le chargement des instructions. Vu que les instructions sont chargées de manière répétée depuis les mêmes adresses, cela implique que des boucles exécutent la même séquence d'instructions de nombreuses fois. Des données différentes sont lues à des adresses à peu près linéaires, mais avec de légères perturbations locales, comme l'indique la grande diagonale verte des accès en lecture. Même pour un œil exercé, la trace de la Figure 4 ne ressemble spécialement pas à une opération DES. Cependant, si on prend la peine d'observer de plus près l'espace des adresses de la pile (tout à droite), alors les 16 tours de DES deviennent clairement visibles. Cet agrandissement est présenté en Figure 5 où l'axe vertical est inchangé (par rapport à la Figure 4) mais où l'espace des adresses (l'axe horizontal) est redimensionné pour ne montrer que les lectures et les écritures sur la pile.

À cause des boucles dans l'exécution du programme, on ne peut pas limiter la trace à certaines instructions pour cibler un tour spécifique. La trace sur l'exécution complète ne prenant qu'une fraction de seconde, nous décidons de tracer l'intégralité du programme sans le moindre filtre. Les traces sont exploitables facilement par DCA : par exemple, si nous traçons les octets écrits sur la pile au cours de l'exécution et que nous calculons

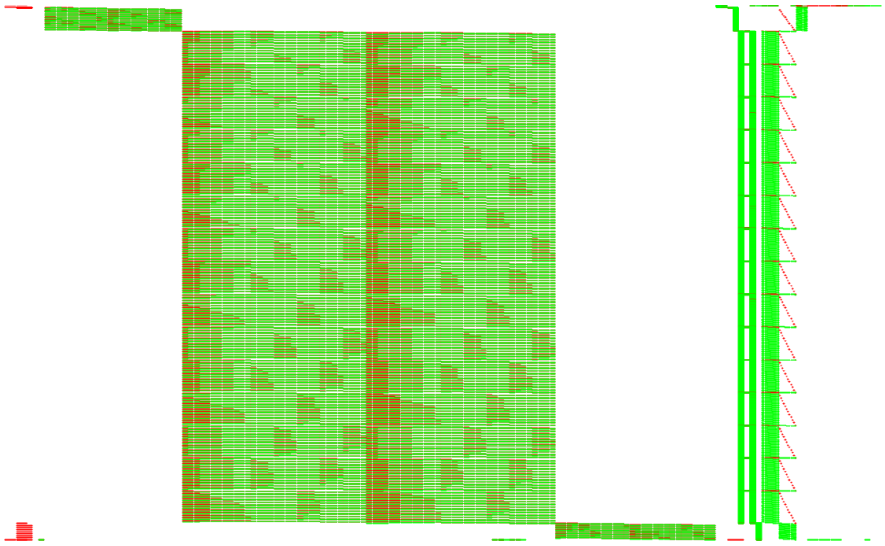

Fig. 5. Un agrandissement sur l'espace d'adresses de la pile. Les 16 tours de l'algorithme DES sont clairement visibles.

une DPA sur l'ensemble de la trace, sans même essayer de la limiter au premier tour, la clé est retrouvée complètement avec seulement 65 traces, en utilisant la sortie du premier tour comme valeur intermédiaire cible.

La mise en œuvre de l'attaque, depuis le téléchargement du binaire jusqu'à la récupération de la clé complète, y compris l'acquisition et l'analyse des traces, prit moins d'une heure vu que son interface textuelle basique rend le couplage aux outils d'attaque très aisée. Extraire des clés d'autres implémentations «white-box» basées sur le même design prend maintenant quelques secondes une fois l'intégralité du processus automatisée comme souligné dans la Section 4.

5.2 Le challenge Hack.lu 2009

Pour la conférence Hack.lu 2009, Jean-Baptiste Bédrune avait réalisé un challenge [6] qui consistait en un fichier *crackme.exe* : un exécutable pour la plateforme Microsoft Windows. Lorsqu'il est lancé, il ouvre une fenêtre graphique qui demande une entrée, l'envoie à travers une «white-box» et compare la sortie avec une référence interne. Il a été résolu indépendamment par Eloi Vanderbéken [68], qui a renversé la fonctionnalité de la «white-box» d'un chiffrement en un déchiffrement, et par SysK [63] qui est parvenu à extraire la clé secrète de l'implémentation.

Nos greffons pour les outils de DBI n'ont pas été portés sous Windows et s'exécutent pour l'instant uniquement sous GNU/Linux et Android. Pour utiliser nos outils tels quels, nous avons décidé de tracer le binaire

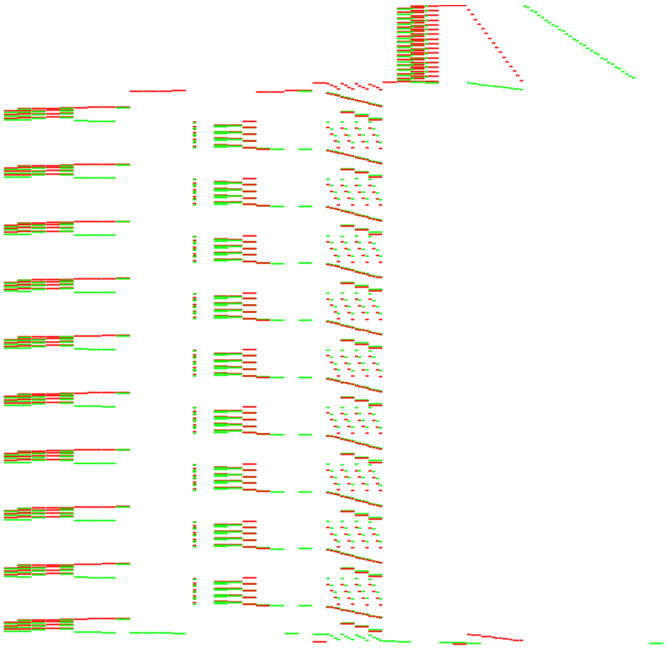

Fig. 6. Visualisation des écritures et lectures sur la pile dans une trace d'exécution du challenge Hack.lu 2009.

avec notre variante pour Valgrind via Wine [2], un outil libre qui offre une couche de compatibilité pour exécuter des applications prévues pour Windows sous GNU/Linux. Nous avons automatisé les interactions avec la fenêtre graphique, le clavier et la souris grâce à l'outil xdotool[7]. Par la configuration de ce challenge, nous avons le contrôle de l'entrée de la «white-box». La sortie n'étant pas nécessaire à l'attaque, nous n'avons pas eu besoin d'entreprendre la moindre rétro-ingénierie du binaire.

La Figure 6 montre les accès en lecture et écriture sur la pile pendant une simple exécution du binaire. On peut observer dix motifs répétitifs sur la gauche entrelacés avec neuf autres sur la droite. Cela indique (avec une haute probabilité) un chiffrement ou un déchiffrement AES avec une clé de 128 bits (et ses dix tours). Le dernier tour est réduit puisqu'il omet l'opération *MixColumns* selon la définition de l'AES.

Nous avons capturé quelques dizaines de traces de l'exécution complète, sans chercher à nous limiter au premier tour. À cause de la surcharge due à l'utilisation de Wine et par l'interaction avec l'interface graphique, l'acquisition a tourné plus lentement que d'ordinaire : l'obtention d'une seule trace prend trois secondes. À nouveau, nous avons appliqué notre technique de DCA sur les traces des octets écrits sur la pile. La clé secrète a pu être complètement récupérée avec seulement 16 traces lorsque la

[7] `http://www.semicomplete.com/projects/xdotool/`

sortie du *SubBytes* du premier tour est utilisée comme valeur intermédiaire d'un chiffrement AES-128.

Comme SysK l'a fait remarquer dans [63], ce challenge a été conçu pour être faisable en deux jours de conférence et, par conséquent, il n'employait pas d'encodage interne, ce qui signifie que les états intermédiaires peuvent être observés directement. C'est pourquoi dans notre DCA, la corrélation entre les états internes et les valeurs tracées atteint la valeur la plus haute possible, ce qui explique le faible nombre de traces requis pour réussir l'attaque.

5.3 Le challenge SSTIC 2012

Chaque année à l'occasion du *Symposium sur la sécurité des technologies de l'information et des communications*, un challenge est publié, consistant en une suite d'étapes telle une poupée russe. En 2012, l'une des étapes [45] était de valider une clé par un *bytecode* Python «check.pyc», c.-à-d. un objet *marshal*[8]. En interne, ce bytecode génère un message aléatoire et l'envoie à travers une «white-box» (créée par Axel Tillequin) *et* à une implémentation traditionnelle d'un chiffrement DES utilisant la clé fournie par l'utilisateur puis compare les deux résultats chiffrés. Cinq participants sont parvenus à retrouver la clé secrète correspondante et leurs rapports sont disponibles sur [45]. Un certain nombre de solutions ont identifié l'implémentation comme une WB-DES dépourvue d'encodages (variante dite nue) telle que décrite dans [21]. Certains ont extrait la clé en suivant les approches de la littérature académique tandis que d'autres ont employé leur propre attaque algébrique.

Tracer entièrement l'interpréteur Python avec notre outil, que ce soit la variante avec PIN ou avec Valgrind, afin d'obtenir une trace d'exécution du bytecode Python serait faire face à une sérieuse surcharge. Nous avons choisi d'instrumenter directement l'environnement Python. En fait, le bytecode Python peut être décompilé assez facilement comme l'a montré Jean Sigwald dans son rapport qui contient une version décompilée du fichier «check.pyc». Néanmoins la partie «white-box» y est laissée sérialisée sous forme d'objet *pickle*[9]. La «white-box» fait usage d'une classe *Bits* séparée pour manipuler ses variables, ce qui nous permet d'ajouter une instrumentation simpliste qui enregistre toutes les instances nouvellement créées de cette classe.

À nouveau, comme pour la WB-AES du challenge de Hack.lu 2009 (cf. Section 5.2), 16 traces sont suffisantes pour récupérer la clé de cette

[8] `https://docs.python.org/2/library/marshal.html`
[9] `https://docs.python.org/2/library/pickle.html`

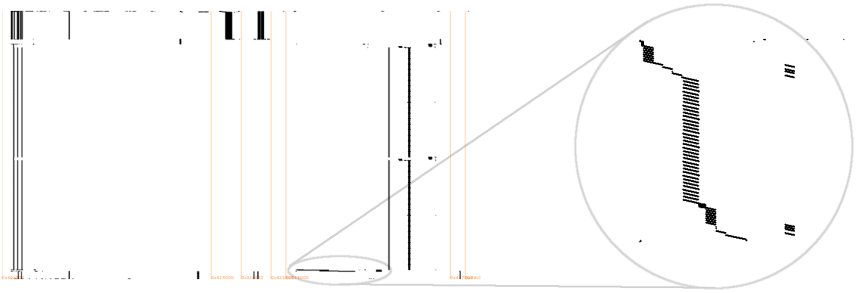

Fig. 7. Visualisation des instructions dans une trace d'exécution de la WB-AES de Karroumi implémentée par Klinec, avec un agrandissement du cœur de la «white-box».

WB-DES en utilisant la sortie du premier tour comme valeur intermédiaire ciblée. Cette approche fonctionne avec un nombre de traces aussi faible car les états intermédiaires ne sont pas encodés.

5.4 Une implémentation de la «white-box» de Karroumi

Une implémentation «white-box» de l'AES d'après l'approche originale [20] et d'après les *dual ciphers* de Karroumi [33] est fournie avec la thèse de Dušan Klinec [35][10]. Comme expliqué dans la Section 2.1, il s'agit de la dernière variante académique de [20]. Comme il n'y a pas de challenge disponible, nous avons utilisé l'implémentation de Klinec pour créer deux challenges : l'un avec et l'autre sans encodages externes. Cette implémentation est écrite en C++ avec un usage intensif des bibliothèques Boost[11] pour charger dynamiquement et dé-sérialiser les tables «white-box» depuis un fichier.

Une première trace réalisée lors de l'exécution de cette implémentation d'AES en «white-box» (visible sur la partie gauche de la Figure 7) montre que le code de la «white-box» proprement dite ne constitue qu'une fraction du nombre total d'instructions (la plupart des instructions servent à initialiser les bibliothèques Boost et charger les tables). Sur la partie droite de la Figure 7 qui agrandit le cœur de la «white-box», on distingue les neuf *MixColumns* opérant sur les quatre colonnes. Cette structure est d'autant plus visible sur la trace de la pile (cf. Figure 8). Par conséquent, nous avons utilisé un filtre d'adresses pour nous concentrer sur la «white-box» et ignorer toutes les opérations Boost.

Les meilleurs résultats ont été obtenus en traçant l'octet de poids faible des adresses mémoires lors des accès en lecture (pile exclue). Initialement,

[10] Le code est disponible sur `https://github.com/ph4r05/Whitebox-crypto-AES`.
[11] `http://www.boost.org/`

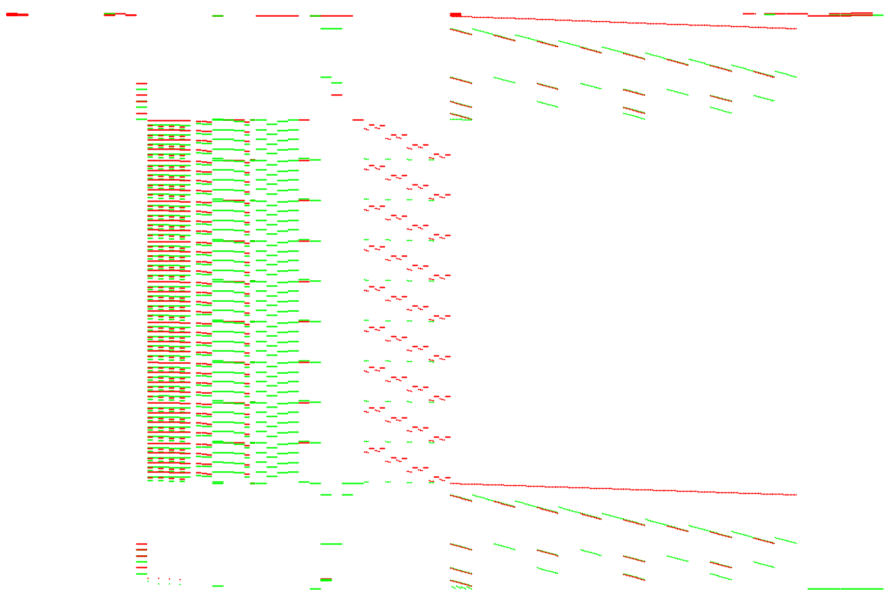

Fig. 8. Visualisation des lectures et écritures sur la pile dans une trace d'exécution limitée au cœur de la WB-AES de Karroumi.

Tableau 1. Classement par DCA de l'octet de clé correct pour une implémentation «white-box» de Karroumi lorsque la sortie de l'étape *SubBytes* dans le premier tour est ciblée, sur base de l'octet de poids faible de l'adresse des lectures en mémoire.

		octet de la clé															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
bit ciblé	**0**	1	256	255	256	255	256	253	1	256	256	239	256	1	1	1	255
	1	1	256	256	256	1	255	256	1	1	5	1	256	1	1	1	1
	2	256	1	255	256	1	256	226	256	256	256	1	256	22	1	256	256
	3	256	255	251	1	1	1	254	1	1	256	256	253	254	256	255	256
	4	256	256	74	256	256	256	255	256	254	256	256	256	1	1	256	1
	5	1	1	1	1	1	1	50	256	253	1	251	256	253	1	256	256
	6	254	1	1	256	254	256	248	256	252	256	1	14	255	256	250	1
	7	1	256	1	1	252	256	253	256	256	255	256	1	251	1	254	1
Tous		✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓

nous avons suivi la même approche que précédemment : nous avons ciblé la sortie de l'étape *SubBytes* dans le premier tour. Mais, contrairement aux autres challenges présentés, ce ne fut pas suffisant pour récupérer immédiatement l'intégralité de la clé. Pour certains des bits considérés dans la valeur intermédiaire, nous observons un pic de corrélation significatif : c'est une indication que l'octet candidat qui arrive en tête est très probablement correct. La Table 1 montre le classement de l'octet de clé correct parmi les candidats après 2000 traces, lorsqu'elles sont triées selon la différence des moyennes (cf. Section 3). Si l'octet de clé est à la position 1 cela signifie qu'il a correctement été extrait par l'attaque. Au total, pour

Tableau 2. Classement par DCA de l'octet de clé correct pour une implémentation «white-box» de Karroumi lorsque le résultat de l'opération inverse au sein de l'étape *SubBytes* dans le premier tour est ciblé, sur base de l'octet de poids faible de l'adresse des lectures en mémoire.

		octet de la clé															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
bit ciblé	**0**	256	256	1	1	1	256	256	256	254	1	1	1	255	256	256	1
	1	1	1	253	1	1	256	249	256	256	256	226	1	254	256	256	256
	2	256	256	1	1	255	256	256	256	251	1	255	256	1	1	254	256
	3	254	1	69	1	1	1	1	1	252	256	1	256	1	256	256	256
	4	254	1	255	256	256	1	255	256	1	1	256	256	238	256	253	256
	5	254	256	250	1	241	256	255	3	1	1	256	256	231	256	208	254
	6	256	256	256	256	233	256	1	256	1	1	256	256	1	1	241	1
	7	63	256	1	256	1	255	231	256	255	1	255	256	255	1	1	1
	Tous	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

le premier challenge que nous avons construit, 15 des 16 octets de la clé sont sortis en position 1 pour au moins un des bits ciblés et un octet de la clé (octet 6 dans la table) n'a pas eu de bons candidats. Il est néanmoins trivial de récupérer ce seul octet par énumération.

Comme l'approche par *dual ciphers* [33] utilise des équivalences affines de la S-box originale, il pourrait être intéressant de se servir d'une autre cible pour calculer les hypothèses : l'inverse dans le corps fini de 2^8 éléments (opération interne à l'étape *SubBytes*) lors du premier tour, avant la transformation affine. Cette seconde attaque donne les résultats présentés dans la Table 2, semblable à la première table mais avec une distribution différente. Avec cette seule attaque, les 16 octets de la clé sont proprement extraits (on peut même réduire le nombre de traces à 500) mais les résultats peuvent varier pour d'autres générations de la «white-box» car la distribution des fuites parmi les bits dans ces deux attaques dépend de la source d'entropie utilisée par le générateur de «white-box». Cependant, lorsqu'on combine les deux attaques, il est toujours possible de récupérer l'intégralité de la clé.

Il est intéressant d'observer dans les Tables 1 et 2 que lorsqu'un bit ciblé pour retrouver un octet de la clé ne fuit pas, le bon candidat est très souvent classé à la dernière position : 256. Cette observation, qui se confirme même avec des nombres plus élevés de traces, est également utilisable pour casser la clé.

Pour se faire une idée de ce qui est réalisable avec une attaque *automatisée* contre de nouvelles instances de cette «white-box» avec d'autres clés, voici quelques chiffres : l'obtention de 500 traces prend à peu près 200s sur une machine standard (CPU dual-core i7-4600U à 2.10GHz). Il

en résulte 832 kbits (104 ko) de traces si on se limite à tracer le premier tour. Exécuter les deux attaques décrites dans cette section prend moins de 30s.

L'attaque sur le second challenge pourvu d'encodages externes donne des résultats similaires. Cela était prévisible puisqu'il n'y a pas de différence du point de vue de l'adversaire entre appliquer ou non un encodage externe tant que dans les deux cas nous avons connaissance des messages originaux avant application de l'encodage.

5.5 Le challenge NoSuchCon 2013

En avril 2013, un challenge conçu par Eloi Vanderbéken a été publié à l'occasion de la conférence NoSuchCon[12]. Le challenge consistait en un exécutable pour Windows embarquant une implémentation «white-box» d'AES. Il était du type «keygen-me», ce qui signifie qu'on devait fournir un nom et le numéro de série correspondant pour réussir. En interne, le numéro de série est chiffré par une «white-box» et le résultat comparé au condensat MD5 du nom fourni.

Le challenge a été résolu par un certain nombre de participants (cf. [43, 62]) mais sans jamais en avoir cassé la clé. Cela montre un autre problème auquel les concepteurs d'implémentations «white-box» doivent faire face en pratique : un adversaire peut convertir une routine de chiffrement en routine de déchiffrement sans pour autant devoir en extraire la clé.

Pour une fois la conception n'est pas dérivée de Chow [20]. Cependant, la «white-box» a été conçue avec des encodages externes qui *ne* faisaient *pas* partie du binaire. Par conséquent, l'entrée fournie en clair par l'utilisateur était considérée comme encodée avec un schéma inconnu et la sortie encodée directement comparée à la référence. Dans ces conditions, sans aucune connaissance de la relation entre le clair ou le chiffré d'un AES standard et les entrées ou sorties réelles de la «white-box» il est infaisable d'appliquer correctement une attaque par DPA puisque pour une attaque par DPA, nous devons être en mesure de construire des hypothèses sur les valeurs intermédiaires. Notons donc que, comme discuté dans la Section 2, cette implémentation «white-box» *n*'est *plus* compatible avec l'AES mais calcule une variante $E'_k = G \circ E_k \circ F^{-1}$. Néanmoins, nous sommes parvenus à en extraire la clé et les encodages avec une nouvelle attaque algébrique décrite dans [65]. Cela ne fut possible qu'après un pénible dés-obscurcissement du binaire (quasi complètement réalisé par

[12] `http://www.nosuchcon.org/2013/`

les *write-ups* disponibles [43, 62]), ce qui est une étape nécessaire pour remplir les prérequis de telles attaques comme décrit dans la Section 2.2.

On retrouve la même «white-box» parmi les challenges CHES 2015 [13] dans une ROM de GameBoy et la même attaque algébrique s'applique comme expliqué dans [64] une fois les tables extraites.

6 Contre-mesures contre la DCA

Dans une puce, les contre-mesures contre la DPA reposent typiquement sur une source d'entropie. Sa sortie est utilisée pour masquer des valeurs intermédiaires, pour réordonner des instructions, ou pour ajouter des délais (cf. par exemple [18, 29, 60]). Pour des implémentations «white-box», on ne peut pas compter sur une source d'entropie puisque dans le modèle d'attaque «white-box», une telle source peut simplement être désactivée ou fixée à une valeur constante. Malgré ce manque d'entropie *dynamique*, nous supposons que l'implémentation qui génère l'implémentation «white-box» a accès à suffisamment d'aléa à injecter dans le code et les tables générés. Comment utiliser cet aléa *statique* incorporé à une implémentation «white-box» ?

Ajouter des délais (aléatoires) dans une tentative de désynchroniser les traces est contré trivialement en utilisant une trace d'instructions à côté de la trace des accès mémoire afin de réaligner les traces automatiquement. Dans [23] il est proposé d'utiliser des encodages *variables* pour accéder aux tables de correspondance sur base d'équivalences affines de S-boxes bijective (cf. [13] pour des algorithmes de résolution du problème d'équivalence affine pour des permutations arbitraires). Comme contre-mesure potentielle contre la DCA, l'aléa incorporé statiquement (et éventuellement couplé à d'autres fonctionnalités) est utilisé pour sélectionner quelle équivalence affine doit être appliquée dans l'encodage d'accès d'une table de correspondance donnée. Cela résulte en un encodage variable (pendant l'exécution) par opposition à l'utilisation d'un encodage statique. Une telle approche peut être vue comme une forme de masquage pour se prémunir d'une attaque DPA classique du premier ordre.

Il est également envisageable de tirer parti de certaines idées des implémentations par seuil (*threshold implementations*) [54]. Une implémentation par seuil est une technique de masquage basée sur le partage de secret et le calcul multipartite. On peut par exemple partager l'entrée en multiples parts de telle sorte que les parts ne sont pas toutes dans la même classe d'équivalence affine. Si ce partage et l'attribution à ces (différentes)

[13] `https://ches15challenge.com/static/CHES15Challenge.zip`

classes d'équivalences affines est faite pseudo-aléatoirement, où l'aléa vient de l'entropie embarquée statiquement et du message d'entrée, alors cela pourrait offrir une certaine résistance aux attaques du type DCA.

Une autre contre-mesure potentielle contre la DCA est l'utilisation d'encodages externes. C'est la raison pour laquelle nous ne sommes pas parvenus à extraire la clé du challenge décrit dans la Section 5.5. Cependant, typiquement, l'adversaire peut obtenir de l'information sur l'encodage externe employé lorsqu'il observe le comportement de la «white-box» au sein de l'application complète (surtout si l'adversaire a le contrôle des entrées utilisées ou peut observer la sortie finale en clair). Nous attirons l'attention que davantage de recherche est nécessaire pour vérifier la robustesse de telles approches et améliorer les idées présentées ici.

En pratique, on peut recourir à des techniques pour compliquer le travail de l'adversaire. Les contre-mesures logicielles typiques sont l'obscurcissement, les méthodes anti-débogage et les tests d'intégrité. Il faut noter cependant que pour mettre en œuvre une attaque par DCA, il n'est pas nécessaire de rétro-concevoir l'exécutable binaire. Les systèmes de DBI réussissent bien à passer outre ces techniques et même s'il existe des efforts pour détecter spécifiquement une DBI [26, 39], les solutions de DBI deviennent également plus discrètes [34].

7 Conclusions et travaux futurs

Comme annoncé dans les premiers articles qui introduisirent le modèle «white-box», il n'y a pas de défense à long terme contre les attaques sur les implémentations «white-box». Cependant, comme nous venons de le montrer dans ces résultats, aucune des implémentations «white-box» actuelles disponibles publiquement n'offre de sécurité même à court terme puisque l'attaque par DCA que nous avons présentée peut en extraire les clés secrètes en quelques secondes. Nous n'avons pas investigué la robustesse de produits «white-box» commercialement disponibles puisque aucune compagnie, pour autant que nous sachions, n'a publié de challenge semblablement au challenge de factorisation de clé RSA [25] (interrompu en 2007), au challenge sur la cryptographie à courbe elliptique [17], ou encore au challenge PRINCE [57]. Cependant, au vu des résultats présentés dans cet article, nous ne pensons pas qu'il soit sage d'utiliser des implémentations «white-box» au sein d'applications qui traitent des informations sensibles telles que par exemple le traitement de cartes de crédit comme suggéré dans [71].

Bien que nous ayons mentionné quelques idées de contre-mesures, comment se prémunir de ces types d'attaques reste une question en suspens. Les contre-mesures contre les attaques par DPA, telles que déployées dans les domaines d'applications très sensibles ne semblent pas transposables directement à cause de la capacité de l'adversaire de désactiver ou interférer avec une éventuelle source d'entropie.

Une autre direction de recherche intéressante est de voir si les techniques plus avancées et plus puissantes utilisées en analyse par canaux auxiliaires par la communauté de cryptographie matérielle peuvent améliorer les résultats dans le contexte «white-box», par exemple l'analyse de corrélation de la consommation (CPA) et les attaques d'ordre plus élevé.

L'étude d'autres points d'attaque tels que l'inverse interne à la S-box dans le premier tour de l'AES donne des résultats intéressants (cf. Section 5.4). Il peut valoir la peine d'investiguer d'autres positions comme cibles dans l'approche DCA.

En conclusion, la cryptographie «white-box» donne une fois de plus raison à Auguste Kerckhoffs : rien ne sert de cacher les détails du design d'une implémentation dans l'espoir de tenir les attaquants à distance.

Références

1. Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001.
2. Bob Amstadt and Michael K. Johnson. Wine. *Linux Journal*, 1994(4), August 1994.
3. Chung Hun Baek, Jung Hee Cheon, and Hyunsook Hong. Analytic toolbox for white-box implementations : Limitation and perspectives. Cryptology ePrint Archive, Report 2014/688, 2014. `http://eprint.iacr.org/2014/688`.
4. Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 221–238, Copenhagen, Denmark, May 11–15, 2014. Springer, Berlin, Germany.
5. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Berlin, Germany.
6. Jean-Baptiste Bédrune. Hack.lu 2009 reverse challenge 1. online, 2009. `http://2009.hack.lu/index.php/ReverseChallenge`.
7. Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
8. Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation : An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003.
9. Eli Biham and Adi Shamir. Differential cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*,

pages 156–171, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Berlin, Germany.
10. Olivier Billet and Henri Gilbert. A traceable block cipher. In Chi-Sung Laih, editor, *ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 331–346. Springer, Berlin, Germany, 2003.
11. Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In Helena Handschuh and Anwar Hasan, editors, *SAC 2004*, volume 3357 of *LNCS*, pages 227–240, Waterloo, Ontario, Canada, August 9–10, 2004. Springer, Berlin, Germany.
12. Alex Biryukov, Charles Bouillaguet, and Dmitry Khovratovich. Cryptographic schemes based on the ASASA structure : Black-box, white-box, and public-key (extended abstract). In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 63–84, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Berlin, Germany.
13. Alex Biryukov, Christophe De Canniére, An Braeken, and Bart Preneel. A toolbox for cryptanalysis : Linear and affine equivalence algorithms. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 33–50, Warsaw, Poland, May 4–8, 2003. Springer, Berlin, Germany.
14. Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential computation analysis : Hiding your white-box designs is not enough. Cryptology ePrint Archive, Report 2015/753, 2015. `http://eprint.iacr.org/`.
15. Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 1–25, San Diego, CA, USA, February 24–26, 2014. Springer, Berlin, Germany.
16. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES 2004*, volume 3156 of *LNCS*, pages 16–29, Cambridge, Massachusetts, USA, August 11–13, 2004. Springer, Berlin, Germany.
17. Certicom. The certicom ECC challenge. Webpage. `https://www.certicom.com/index.php/the-certicom-ecc-challenge`.
18. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Berlin, Germany.
19. Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 13–28, Redwood Shores, California, USA, August 13–15, 2003. Springer, Berlin, Germany.
20. Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In Kaisa Nyberg and Howard M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 250–270, St. John's, Newfoundland, Canada, August 15–16, 2003. Springer, Berlin, Germany.
21. Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box DES implementation for DRM applications. In Joan Feigenbaum, editor, *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002*, volume 2696 of *LNCS*, pages 1–15. Springer, 2003.
22. Joan Daemen and Vincent Rijmen. *The design of Rijndael : AES — the Advanced Encryption Standard*. Springer, 2002.

23. Yoni de Mulder. *White-Box Cryptography : Analysis of White-Box AES Implementations*. PhD thesis, KU Leuven, 2014.
24. Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain. White-box security notions for symmetric encryption schemes. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 247–264, Burnaby, BC, Canada, August 14–16, 2014. Springer, Berlin, Germany.
25. EMC Corporation. The RSA factoring challenge. Webpage. `http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge.htm`.
26. Francisco Falc'o and Nahuel Riva. Dynamic binary instrumentation frameworks : I know you're there spying on me. REcon, 2012. `http://recon.cx/2012/schedule/events/216.en.html`.
27. Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 40–49. IEEE Computer Society, 2013.
28. Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. Cryptanalysis of white box DES implementations. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 278–295, Ottawa, Canada, August 16–17, 2007. Springer, Berlin, Germany.
29. Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *CHES'99*, volume 1717 of *LNCS*, pages 158–172, Worcester, Massachusetts, USA, August 12–13, 1999. Springer, Berlin, Germany.
30. Yu-Lun Huang, F. S. Ho, Hsin-Yi Tsai, and H. M. Kao. A control flow obfuscation method to discourage malicious tampering of software codes. In Ferng-Ching Lin, Der-Tsai Lee, Bao-Shuh Paul Lin, Shiuhpyng Shieh, and Sushil Jajodia, editors, *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006*, page 362. ACM, 2006.
31. Matthias Jacob, Dan Boneh, and Edward W. Felten. Attacking an obfuscated cipher by injecting faults. In Joan Feigenbaum, editor, *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002, Washington, DC, USA, November 18, 2002, Revised Papers*, volume 2696 of *LNCS*, pages 16–31. Springer, 2003.
32. Markus Jakobsson and Michael K. Reiter. Discouraging software piracy using software aging. In Tomas Sander, editor, *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001*, volume 2320 of *LNCS*, pages 1–12. Springer, 2002.
33. Mohamed Karroumi. Protecting white-box AES with dual ciphers. In Kyung Hyune Rhee and DaeHun Nyang, editors, *ICISC 10*, volume 6829 of *LNCS*, pages 278–291, Seoul, Korea, December 1–3, 2011. Springer, Berlin, Germany.
34. Julian Kirsch. Towards transparent dynamic binary instrumentation using virtual machine introspection. REcon, 2015. `https://recon.cx/2015/schedule/events/20.html`.
35. Dušan Klinec. White-box attack resistant cryptography. Master's thesis, Masaryk University, Brno, Czech Republic, 2013. `https://is.muni.cz/th/325219/fi_m/`.
36. Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1) :5–27, 2011.
37. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Berlin, Germany.

38. Tancrède Lepoint, Matthieu Rivain, Yoni De Mulder, Peter Roelse, and Bart Preneel. Two attacks on a white-box AES implementation. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 265–285, Burnaby, BC, Canada, August 14–16, 2014. Springer, Berlin, Germany.
39. Xiaoning Li and Kang Li. Defeating the transparency features of dynamic binary instrumentation. BlackHat US, 2014. `https://www.blackhat.com/docs/us-14/materials/us-14-Li-Defeating-The-Transparency-Feature-Of-DBI.pdf`.
40. Hamilton E. Link and William D. Neumann. Clarifying obfuscation : Improving the security of white-box DES. In *International Symposium on Information Technology : Coding and Computing (ITCC 2005)*, pages 679–684. IEEE Computer Society, 2005.
41. Cullen Linn and Saumya K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger, editors, *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003*, pages 290–299. ACM, 2003.
42. Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200. ACM, 2005.
43. Arnaud Maillet. Nosuchcon 2013 challenge - write up and methodology. online, 2013. `http://kutioo.blogspot.be/2013/05/nosuchcon-2013-challenge-write-up-and.html`.
44. Stefan Mangard, Elisabeth Oswald, and François-Xavier Standaert. One for all - all for one : unifying standard differential power analysis attacks. *IET Information Security*, 5(2) :100–110, 2011.
45. Florent Marceau, Fabien Perigaud, and Axel Tillequin. Challenge sstic 2012. online, 2012. `http://communaute.sstic.org/ChallengeSSTIC2012`.
46. Thomas S. Messerges. Using second-order power analysis to attack DPA resistant software. In Çetin Kaya Koç and Christof Paar, editors, *CHES 2000*, volume 1965 of *LNCS*, pages 238–251, Worcester, Massachusetts, USA, August 17–18, 2000. Springer, Berlin, Germany.
47. Wil Michiels. Opportunities in white-box cryptography. *IEEE Security & Privacy*, 8(1) :64–67, 2010.
48. Wil Michiels and Paul Gorissen. Mechanism for software tamper resistance : an application of white-box cryptography. In Moti Yung, Aggelos Kiayias, and Ahmad-Reza Sadeghi, editors, *Proceedings of the Seventh ACM Workshop on Digital Rights Management*, pages 82–89. ACM, 2007.
49. Wil Michiels, Paul Gorissen, and Henk D. L. Hollmann. Cryptanalysis of a generic class of white-box implementations. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *SAC 2008*, volume 5381 of *LNCS*, pages 414–428, Sackville, New Brunswick, Canada, August 14–15, 2009. Springer, Berlin, Germany.
50. Camille Mougey and Francis Gabriel. Désobfuscation de DRM par attaques auxiliaires. In *Symposium sur la sécurité des technologies de l'information et des communications*, 2014. `www.sstic.org/2014/presentation/dsobfuscation_de_drm_par_attaques_auxiliaires`.
51. James A. Muir. A tutorial on white-box AES. In Evangelos Kranakis, editor, *Advances in Network Analysis and its Applications*, volume 18 of *Mathematics in Industry*, pages 209–229. Springer Berlin Heidelberg, 2013.

52. Yoni De Mulder, Peter Roelse, and Bart Preneel. Cryptanalysis of the Xiao-Lai white-box AES implementation. In Lars R. Knudsen and Huapeng Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 34–49, Windsor, Ontario, Canada, August 15–16, 2013. Springer, Berlin, Germany.
53. Nicholas Nethercote and Julian Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007.
54. Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security, ICICS*, volume 4307 of *LNCS*, pages 529–545. Springer, 2006.
55. Jacques Patarin and Louis Goubin. Asymmetric cryptography with S-boxes. In Yongfei Han, Tatsuaki Okamoto, and Sihan Qing, editors, *ICICS 97*, volume 1334 of *LNCS*, pages 369–380, Beijing, China, November 11–14, 1997. Springer, Berlin, Germany.
56. Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. A survey on security for mobile devices. *IEEE Communications Surveys and Tutorials*, 15(1) :446–471, 2013.
57. Ruhr University Bochum, NXP Semiconductors, and Technical University of Denmark. The PRINCE challenge. Webpage. `https://www.emsec.rub.de/research/research_startseite/prince-challenge/`.
58. Eloi Sanfelix, Job de Haas, and Cristofaro Mune. Unboxing the white-box : Practical attacks against obfuscated ciphers. Presentation at BlackHat Europe 2015, 2015. `https://www.blackhat.com/eu-15/briefings.html`.
59. Amitabh Saxena, Brecht Wyseur, and Bart Preneel. Towards security notions for white-box cryptography. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *ISC 2009*, volume 5735 of *LNCS*, pages 49–58, Pisa, Italy, September 7–9, 2009. Springer, Berlin, Germany.
60. Kai Schramm and Christof Paar. Higher order masking of the AES. In David Pointcheval, editor, *CT-RSA 2006*, volume 3860 of *LNCS*, pages 208–225, San Jose, CA, USA, February 13–17, 2006. Springer, Berlin, Germany.
61. Federico Scrinzi. Behavioral analysis of obfuscated code. Master's thesis, University of Twente, Twente, Netherlands, 2015. `http://essay.utwente.nl/67522/1/Scrinzi_MA_SCS.pdf`.
62. Axel Souchet. AES whitebox unboxing : No such problem. online, 2013. `http://0vercl0k.tuxfamily.org/bl0g/?p=253`.
63. SysK. Practical cracking of white-box implementations. Phrack 68 :14. `http://www.phrack.org/issues/68/8.html`.
64. Philippe Teuwen. CHES2015 writeup. online, 2015. `http://wiki.yobi.be/wiki/CHES2015_Writeup#Challenge_4`.
65. Philippe Teuwen. NSC writeups. online, 2015. `http://wiki.yobi.be/wiki/NSC_Writeups`.
66. Ludo Tolhuizen. Improved cryptanalysis of an AES implementation. In *Proceedings of the 33rd WIC Symposium on Information Theory*. Werkgemeenschap voor Inform.- en Communicatietheorie, 2012.
67. U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. *Data Encryption Standard (DES)*.
68. Eloi Vanderbéken. Hacklu reverse challenge write-up. online, 2009. `http://baboon.rce.free.fr/index.php?post/2009/11/20/HackLu-Reverse-Challenge`.

69. Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 264–277, Ottawa, Canada, August 16–17, 2007. Springer, Berlin, Germany.
70. Yaying Xiao and Xuejia Lai. A secure implementation of white-box AES. In *Computer Science and its Applications, 2009. CSA '09. 2nd International Conference on*, pages 1–6, 2009.
71. Nilima Yadav and Sarvesh Tanwar. Implementation of white-box cryptography in credit card processing combined with code obfuscation. *International Journal of Computer Applications*, 70(2) :35–38, May 2013.
72. Y. Zhou and S.T. Chow. System and method of hiding cryptographic private keys, December 15 2009. US Patent 7,634,091.