# Approaches for analyzing security properties of smart objects

Florian Lugou

UNIVERSITÉ CÔTE D'AZUR

TELECOM ParisTech

Une école de l'IMT

École doctorale : Sciences et Technologies de l'Information et de la Communication
Unité de recherche : IMT, Telecom ParisTech

# Thèse de doctorat

Présentée en vue de l'obtention du
grade de docteur en Informatique
de
l'Université Côte d'Azur

par
Florian Lugou

# Environments for analyzing the security of smart objects

Dirigée par Ludovic Apvrille, Telecom ParisTech et
codirigée par Aurélien Francillon, EURECOM

Soutenue le 8 Février 2018
Devant le jury composé de :

| | | |
|---|---|---|
| Dr. Ludovic Apvrille | Maître de conférences, HDR, Telecom ParisTech | Directeur de thèse |
| Prof. Jean-Michel Bruel | Professeur, IRIT | Rapporteur |
| Prof. Paolo Camurati | Professeur, Politecnico di Torino | Examinateur |
| Dr. Stéphanie Delaune | Directrice de recherche, IRISA | Rapporteure |
| Dr. Aurélien Francillon | Maître de conférences, EURECOM | Codirecteur de thèse |
| Dr. Jair Gonzalez | Ingénieur, ANSYS | Examinateur |
| Prof. Vincent Nicomette | Professeur, LAAS | Examinateur |

# Remerciements

Si cette thèse a pu voir le jour, c'est avant tout grâce à l'exemple et au soutien des personnes qui m'ont été proches jusqu'aujourd'hui, et tout particulièrement celles qui m'ont accompagné durant ces trois dernières années. Je leur dédie naturellement le premier moment de cet écrit.

Je pense d'abord bien sûr à Aurélien Francillon et Ludovic Apvrille qui furent pour moi des professeurs passionnants avant d'accepter de devenir mes directeurs de thèse. Leur enthousiasme est à l'origine de mon goût pour une approche de l'informatique à la fois immédiate et rigoureuse. Je les remercie pour avoir cru en moi, pour m'avoir soutenu dans les moments difficiles et encouragé dans mes regains de dynamisme. Je leur suis surtout reconnaissant pour leur disponibilité, leur attention, leur investissement et leur optimisme.

Je remercie particulièrement Stéphanie Delaune et Jean-Michel Bruel qui ont accepté d'être les rapporteurs de cette thèse. Les commentaires qu'ils m'ont faits à l'issue de la relecture de ce manuscrit m'ont permis d'en améliorer le contenu et la forme. Merci également à tous les autres membres du jury qui se sont intéressés à mon travail : Paolo Camurati, Jair Gonzalez et Vincent Nicomette.

Mon exil méridional loin des liens familiaux m'a été adouci par la présence d'amis avec lesquels j'ai pu partager mes soirées, week-end et vacances. Merci notamment à Adrien pour son insupportable excentricité, à Guillaume pour avoir été clément envers nos alter ego lors de nos soirées entre rôlistes et à Zoé pour avoir apporté un peu de soleil poitevin dans la grisaille azuréenne.

Le travail de doctorant au sein du LabSoC n'aurait pas été aussi agréable sans l'aide, les débats et les fous rires échangés avec mes collègues. Merci en particulier à Letitia de m'avoir supporté lors de nos travaux communs ; à Tullio de m'avoir appris qu'il existait des pirogues estampillées CNRS ; à Rabéa de m'avoir guidé dans mon exploration des méthodes formelles ; à Renaud de m'avoir montré qu'on pouvait étendre l'excellence scientifique au jardinage, à la cuisine, à l'Histoire, à l'économie, à la langue, à la physique et à la littérature sans jamais oublier que l'on est humain.

Finalement, merci à ma famille pour m'avoir porté jusqu'ici. À ma mère Frédérique pour m'avoir appris à regarder et, à travers un regard sans tache, à voir. À mon père Philippe pour m'avoir appris à agir et à transformer ce que je voyais. À mes sœurs : Audrey pour m'avoir appris ce qu'était la famille ; Solenne, ce qu'était la force ; Cécile, la vie.

Évidemment, un immense merci à Elsa. Pour avoir été à mes côtés ces quatre dernières années, pour avoir été ma joie et ma force quand elles me manquaient, pour m'avoir donné un lieu hors du temps où me retrouver, pour ton amour : merci.

# Résumé

## 1 Introduction

Il peut être surprenant de s'interroger sur la différence même entre logiciel et matériel d'un point de vue théorique. En effet, à l'inverse d'un langage de modélisation, un code logiciel n'a pas de sens intrinsèque. Son utilité finale est d'être exécuté par une machine physique et son effet sensible est indissociable de l'objet matériel qui l'a *interprété*. Le terme *interprétation* relève cette spécificité : le code logiciel n'a pas de sens en soi, il doit être interprété. Et interprété par quelque chose.

Inversement une implémentation matérielle, en tant qu'objet physique, a un comportement que l'on peut considérer déterministe et qui découle des lois fondamentales de la physique. Son état actuel est déterminé par une fonction plus ou moins chaotique de son état précédent et de son environnement. Certaines classes de matériel (que l'on nomme processeur ou contrôleur) ont cependant un comportement paramétré par un ensemble de règle stockées dans un médium spécifique (RAM, ROM, mémoire flash, etc.). Leur comportement observable change complètement en fonction de l'état de ce médium. Décrire un tel objet matériel en omettant ces règles (que l'on nomme logiciel), ne donne alors que peu d'indications quant à l'évolution de l'objet.

Afin de donner un sens à cette distinction entre logiciel et matériel, il nous faut y intégrer une autre composante. En effet, un code logiciel n'est probablement jamais écrit par un développeur qui l'envisagerait comme un ensemble de paramètres d'une machine complexe car cela requerrait d'imaginer toute la logique implémentée par le processeur chaque fois qu'une instruction logicielle est écrite. Un développeur logiciel voit dans chaque instruction une action opérant sur une machine abstraite que l'on pourrait appeler un *modèle abstrait du matériel*.

#### Systèmes embarqués, sécurité et vérification formelle

Cette distinction entre logiciel et matériel est particulièrement apparente dans le domaine de la conception des systèmes embarqués. En effet, la séparation des fonctionnalités du système entre composants logiciels et composants matériel impacte fortement la suite du processus de conception en termes de méthodologie, de langage et de méthodes de vérification (tests ou vérifications formelles). Cette divergence complexifie l'analyse formelle de systèmes composés d'éléments logiciels et matériels.

La sécurité des systèmes embarqués est cependant un problème central du développement de l'Internet des objets tel qu'il est perçu aujourd'hui. Comme l'a montré récemment le botnet Mirai [152], des attaques d'envergures peuvent exploiter ces systèmes qui sont de plus en plus nombreux et ne bénéficient historiquement pas de standards de sécurité à la hauteur des autres systèmes informatiques alors qu'ils sont de plus en plus connectés et en charge de tâches de plus en plus critiques (dans la santé ou les transports par exemple).

Bien que les méthodes d'analyse formelle soient en mesure d'améliorer considérablement la

confiance que l'on accorde à un système informatique, leur application dans le cadre de la conception de systèmes embarqués se heurte à des défis théoriques (algorithmiques dans le cas où l'intrication entre logiciel et matériel est étroite) et pratiques (leur intégration à un processus de conception). La question à laquelle nous nous intéressons dans cette thèse est donc la suivante : comment les méthodes de vérification formelle automatiques peuvent elles aider les développeurs de systèmes embarqués à évaluer l'impact d'une architecture ou d'un composant matériel personnalisé sur la sécurité d'un système embarqué lors de sa conception ?

### Contributions

Pour répondre à cette question, nous avons proposé durant cette thèse différents éléments de solution.

Premièrement nous nous sommes intéressés à la façon dont les modifications matérielles sont à même de garantir la sécurité d'un système de sorte qu'une évaluation purement logicielle du système ne permettrait pas d'en déduire la correction. Nous avons travaillé pour cela sur un prototype exploitant des modifications matérielles afin d'établir un canal de communication sécurisé entre un périphérique et une application protégée par la technologie Intel Software Guard eXtension (SGX).

Dans un second temps, nous nous sommes intéressés à l'intégration de méthodes de vérification formelle automatiques à une méthodologie de conception de systèmes embarqués basée sur des modèles. Nous nous basons pour cela sur un langage de modélisation ciblant spécifiquement la conception de systèmes embarqués et présentant des capacités spécifiques à la modélisation de propriétés de sécurité. La méthode proposée [166] donne au concepteur la possibilité de vérifier formellement et automatiquement des propriétés lors des phases d'analyse [167], de partitionnement [165] et de conception logicielle [179].

Cette méthode de vérification permet d'analyser des systèmes décrits en termes de composants interagissant entre eux au moyen d'une interface simple (envoi et réception de messages). Afin de permettre la vérification de systèmes où composants logiciels et matériels interagissent de manière plus complexe, nous avons proposé une nouvelle méthode [177,178] permettant à un développeur logiciel d'intégrer l'effet de modifications matérielles dans un algorithme de vérification de code logiciel. Cette méthode repose sur un outil qui traduit la spécification d'un système décrit par une représentation bas niveau du logiciel (code assembleur) et une représentation haut niveau de l'architecture matérielle en un modèle vérifiable par un outil externe. Cette transformation a été pensée de manière à laisser à l'utilisateur la possibilité d'ajouter de nouveaux modules exprimant l'effet d'un composant matériel sur l'algorithme de vérification.

## 2 Logiciel/Matériel et sécurité

Les composants matériels jouent un rôle important dans de nombreuses solutions garantissant la sécurité d'un programme logiciel où ils peuvent être utiles pour améliorer les performances ou pour ancrer la racine d'une chaîne de confiance dans un élément matériel. Nous proposons dans ce chapitre d'étudier un exemple de système permettant d'établir un canal de communication sécurisé entre une application et un périphérique. Ce système s'appuie sur l'architecture proposée par Intel SGX et sur quelques modifications matérielles afin de protéger les entrées/sorties d'une application contre un attaquant contrôlant l'ensemble de la pile logicielle et ayant un accès physique au bus qui connecte l'application et le périphérique.

### L'architecture Intel SGX

Dans un environnement virtualisé, il existe deux classes de méthodes permettant à une machine virtuelle d'accéder à des données traitées par une autre machine virtuelle colocalisée : les méthodes qui n'utilisent que des données directement accessibles par la machine virtuelle et celles qui présupposent l'exploitation d'une vulnérabilité dans l'hyperviseur. Ces vulnérabilités, malheureusement existantes [205], ont motivé la recherche de solutions permettant d'isoler une machine virtuelle d'un hyperviseur potentiellement corrompu. C'est en partie pour répondre à ce problème que la technologie Intel SGX a vu le jour. Celle-ci garantit la stricte isolation d'*enclaves* en se reposant sur une extension au jeu d'instruction et un contrôle des accès mémoire lorsqu'une entrée est ajoutée au Translation Lookaside Buffer. De plus, les données stockées dans la mémoire sont chiffrées par le contrôleur mémoire, ce qui protège une application contre un attaquant contrôlant physiquement tout l'environnement du processeur (mais ne pouvant pas accéder aux signaux internes au processeur).

### Canal de communication avec un périphérique sécurisé

Cette protection n'est cependant valable que si l'enclave ne se sert que de données présentes en mémoire. Si l'interaction avec un périphérique est nécessaire, le modèle d'attaquant doit être considérablement affaibli ou bien l'enclave doit mettre en œuvre une protection cryptographique (logicielle) et transmettre les données chiffrées directement au périphérique (MMIO) ou les réécrire en mémoire afin d'initier un transfert DMA. *Nous proposons une addition à l'architecture Intel SGX qui permettrait d'établir un canal sécurisé entre une enclave et un périphérique pour un coût en performance limité.*

Afin de décharger le processeur, le chiffrement de la communication avec le périphérique est délégué à une unité matérielle. Pour ne pas diminuer considérablement le modèle d'attaquant, cette unité matérielle devra se situer sur le die du processeur. Cela impose de fortes contraintes concernant la taille de l'unité (et donc concernant la logique qu'elle peut implémenter). Cependant, il est nécessaire que l'unité de chiffrement puisse différentier les zones mémoire des enclaves afin d'utiliser du matériel cryptographique différent d'une enclave à l'autre. L'unité de chiffrement doit donc implémenter une logique lui permettant de choisir une clé de chiffrement adaptée à l'enclave ciblée par un accès mémoire. Afin de minimiser la taille de cette unité, nous avons décidé de réutiliser le composant matériel (appelée *DMA remapping unit* dans les architectures Intel) qui est chargé de parcourir la table des pages d'entrée/sortie. Cette table est cependant contrôlée par le système d'exploitation (ou hyperviseur) qui est considéré comme potentiellement malveillant. Le matériel cryptographique utilisé par la DMA remapping unit ne peut donc pas être stocké dans cette table. Pour stocker la clé de chiffrement, nous proposons donc d'y consacrer un champ dans chaque entrée de l'*Enclave Page Cache Map* (EPCM). Ces entrées sont utilisées par SGX pour stocker des informations concernant les pages de mémoire protégées par SGX : à chaque page protégée est associée une entrée de l'EPCM (stockée dans la mémoire) dont l'adresse est calculée à partir de l'adresse de la page.

Cette architecture permet à une enclave d'autoriser les transferts DMA ciblant une de ses pages en utilisant la procédure suivante :

1. L'enclave utilise une instruction nouvellement créée afin de définir le matériel cryptographique à utiliser pour une page appartenant à l'enclave courante. Ce matériel (clé et numéro de séquence) est sauvegardé dans l'entrée de l'EPCM correspondant à la page ciblée.

2. Le système d'exploitation peut alors ajouter une entrée dans la table des pages d'entrée/sortie faisant correspondre une adresse périphérique à l'adresse physique de la page protégée.

3. Lorsque le périphérique cherche à accéder à cette page, la DMA remapping unit parcourt la table des pages d'entrée/sortie afin de déterminer l'adresse physique associée à cette page.

4. Puisque cette adresse appartient à la plage réservée au processeur, la DMA remapping unit récupère le matériel cryptographique correspondant stocké dans l'entrée de l'EPCM. Si une clé a été définie, la correspondance entre adresse périphérique et adresse physique, la clé et le numéro de séquence sont inscrits dans l'I/O Translation Lookaside Buffer (I/O TLB). L'accès mémoire est alors transmis au contrôleur mémoire.

5. Lorsque le contrôleur mémoire retourne la donnée requise, celle-ci est sauvegardée dans le cache L3.

6. La donnée et le numéro de séquence sont concaténés et chiffrés et un MAC est calculé par la DMA remapping unit sur le MAC et l'adresse périphérique. La donnée chiffrée et le MAC sont transmis au périphérique. Finalement, le numéro de séquence est incrémenté.

7. Si le périphérique cherche à accéder à cette même page plus tard, l'adresse physique correspondante, la clé et le numéro de séquence sont déjà présents en cache.

**Conclusion**

Cette architecture permet de garantir un canal sécurisé entre un périphérique et une enclave. La protection cryptographique envisagée garantie une protection plus complète de ces communications que les approches traditionnelles focalisée sur un modèle d'attaquant purement logiciel.

Bien que cette protection entraine un coût non négligeable en termes de performance, l'implémentation matérielle des algorithmes cryptographiques et la réutilisation des techniques de caches présentes dans les processeurs Intel permettent d'atténuer les pertes en contrepartie de modifications physiques limitées.

La vérification formelle des architectures composées d'éléments logiciels et matériels comme celle présentée dans ce chapitre pose des problèmes théoriques et pratiques dont la nature dépend du degré d'interaction entre logiciel et matériel.

# 3 Vérification formelle de systèmes embarqués à partir de modèles de conception

Une des spécificités des problématiques de sécurité dans le domaine des systèmes embarqués réside dans la difficulté d'appliquer des correctifs de sécurité purement logiciels à une étape avancée du développement du système, en particulier lorsque la vulnérabilité à corriger découle d'un choix architectural. Intégrer l'évaluation de sécurité aux méthodologies de conception de systèmes embarqués dès leurs premières étapes est donc un des enjeux importants d'une meilleure prise en compte de ces problématiques dans une industrie où la sécurité a souvent été reléguée au second plan.

Parmi les différentes approches cherchant à systématiquement prendre en compte les considérations de sécurité dans la conception de systèmes, nous nous intéressons particulièrement à celles permettant d'intégrer ces considérations à des modèles utilisés pour la conception. En effet, ceux-ci rendent possible l'analyse des comportement des systèmes et nous permettent de confronter ces comportements à des objectifs de sécurité. Dans le cadre de cette thèse, notre choix s'est porté sur un langage de modélisation dédié à la conception de systèmes embarqués sécurisés : SysML-Sec.

### SysML-Sec

SysML-Sec supporte trois phases principales de modélisation :

- Une phase d'analyse d'exigences qui détaille ce que le système doit réaliser et non comment les comportements attendus sont implémentés.
- Une phase de partitionnement logiciel/matériel au cours de laquelle le concepteur décrit une vue fonctionnelle et une vue architecturale du système, ainsi qu'une association des éléments fonctionnels aux éléments architecturaux.
- Une phase de conception logicielle au cours de laquelle les comportements des éléments fonctionnels modélisés dans la phase précédente sont détaillés.

Différentes méthodes de vérification permettent, durant chacune de ces phases, d'évaluer des propriétés de sûreté, de sécurité et de performance. De plus, ce langage et ces méthodes de vérification sont supportées par un outil open source *TTool* qui nous a permis d'implémenter les algorithmes présentés dans ce chapitre.

### Vérification de propriétés de sécurité sur des diagrammes SysML-Sec de conception logicielle

Dans le cadre de cette thèse, mon travail s'est surtout porté sur la vérification de propriétés de sécurité (confidentialité, authenticité) lors de la phase de conception logicielle.

Ce travail a consisté à implémenter un algorithme de transformation d'un modèle de conception logicielle SysML-Sec en un modèle décrit en *pi calcul appliqué*. Ce langage de description, fondé sur les algèbres de processus, permet la modélisation d'acteurs communicant au travers de canaux formellement définis. Ce modèle est ensuite interprété par un outil de vérification de protocoles cryptographiques (ProVerif) afin de prouver mathématiquement des propriétés de sécurité sur cette description du système.

L'algorithme de traduction proposé transforme le modèle du système établi lors de l'étape de conception logicielle et qui comporte des diagrammes d'architecture (blocs) et de comportements (machines à états) en des processus correspondants. Les propriétés de sécurité spécifiées sur les diagrammes de blocs au moyen de notes formelles (pragmas) sont également traduites et l'outil ProVerif permet de vérifier formellement que ces propriétés sont valides sur la description du système en pi calcul appliqué. Lorsque ProVerif exhibe une trace permettant de contredire une de ces propriétés, cette trace est transformée par notre outil afin d'afficher un diagramme de séquence illustrant les étapes qui ont mené à la violation de la propriété dans les diagrammes de conception logicielle.

Cet algorithme a été implémenté dans l'outil TTool et nous a permis de vérifier formellement des propriétés de sécurité intéressantes sur des modèles de systèmes tels que le protocole d'échange de clé décrit au chapitre précédent ou une implémentation de TLS.

Afin d'améliorer la fiabilité des preuves fournies par notre outil, nous avons décrit l'algorithme de manière formelle et avons prouvé mathématiquement la correction de notre transformation. Cette preuve, présentée en annexe, s'appuie sur une expression formelle de la sémantique du pi calcul appliqué et de la sémantique des diagrammes SysML-Sec afin de garantir qu'un modèle SysML-Sec présentant une vulnérabilité engendre bien un modèle en pi calcul appliqué également vulnérable.

### Analyse de sécurité durant le partitionnement

Bien que la sécurité d'un système soit intimement liée aux détails d'implémentation, certaines vulnérabilités architecturales peuvent découler de décisions prises durant l'étape de partitionne-

ment. Ces vulnérabilités sont alors très compliquées à corriger. Cela nous a motivé à explorer la possibilité d'intégrer des méthodes de vérification de sécurité dès la phase de partitionnement.

Pour cela, nous avons introduit des éléments graphiques appelés *Cryptographic Configuration* qui permettent d'étiqueter les communications représentées dans les diagrammes fonctionnels afin d'indiquer que cette communication sera en fin de compte protégée par un algorithme cryptographique. Cet artéfact de modélisation précise les propriétés de sécurité garanties par l'algorithme cryptographique, son impact sur le temps de calcul et sur la taille des données échangées. De cette façon, le concepteur peut prendre en compte l'impact sur les performances qu'auront les mécanismes de sécurité implémentés.

Un algorithme de traduction vers un modèle en pi calcul appliqué est proposé afin de vérifier formellement la validité des protections cryptographiques mises en place.

#### Conclusion

La technique de vérification proposée dans ce chapitre repose sur une claire définition de l'interaction entre modules matériels et modules logiciels (au travers de canaux bien définis). Dans le cas où cette interaction est plus complexe, une autre méthodologie doit être envisagée.

## 4 Vérification formelle de matériel et logiciel imbriqués

Lorsque les modifications matérielles apportées à un système affectent la sémantique de la partie logicielle, utiliser un outil de vérification générique pour analyser le code logiciel montre ses limites.

Dans ce dernier chapitre, nous explorons la possibilité d'une méthode de vérification de code logiciel prenant en compte l'impact que des modifications matérielles peuvent avoir sur cette preuve. Pour cela, nous proposons un algorithme de transformation d'un code assembleur en une spécification ProVerif (en pi calcul appliqué). Cet algorithme (implémenté dans un outil appelé SMASHUP) repose sur une architecture modulaire afin de proposer au concepteur d'adapter la vérification à l'architecture matérielle du système.

Cette adaptation prend deux formes. Premièrement, l'algorithme de traduction repose sur une description haut niveau de l'architecture matérielle du système afin de décider quels modules de l'algorithme utiliser et comment les paramétrer. Deuxièmement, un concepteur qui cherche à vérifier un système faisant intervenir un élément matériel pour lequel aucun module de l'algorithme de vérification n'existe, peut créer un nouveau module afin de modifier comment la traduction (et donc la vérification) se fera.

Bien que cette méthode ne permette pas de vérifier n'importe quel système, elle ouvre la possibilité de prendre en compte un certain degré de personnalisation du matériel, tout en limitant l'effort requis pour réécrire complètement un algorithme de vérification.

## 5 Conclusion

Dans cette thèse, nous nous sommes intéressés au rôle des méthodes de vérification formelle automatiques dans le contexte de la conception de systèmes embarqués. Nous avons montré que les termes dans lesquels cette question se posait étaient très différents selon le degré d'intrication des éléments logiciels et matériels. En effet, lorsqu'il s'agit de vérifier un système composé d'éléments logiciels et matériels, utiliser un outil de vérification logicielle existant n'est possible que lorsque les modifications apportées par les éléments matériels au modèle du matériel envisagé par l'outil

de vérification n'invalide pas la preuve que celui-ci fournit. Ces considérations nous ont permis d'envisager le problème de la vérification de systèmes embarqués sous deux angles :

- Comment peut on intégrer la vérification automatique de propriétés de sécurité à un processus de conception de système embarqué lorsque les fonctionnalités de haut niveau du système peuvent être clairement réparties entre composants logiciels et composants matériels ?

- Comment peut on envisager une méthode de vérification formelle qui prendrait en compte des modifications matérielles afin d'analyser un code logiciel destiné à être exécuté par ce matériel personnalisé ?

### Résumé des contributions

Les contributions que j'ai faites durant cette thèse s'organisent autour de ces deux problématiques.

Durant la conception de systèmes embarqués, l'affectation d'éléments fonctionnels sur des éléments architecturaux entraine, d'un point de vue de la sécurité du système, une différence en termes de capacités de l'attaquant. Par exemple, la confidentialité de données échangées sur un canal abstrait dépend de si ce canal sera implémenté par un bus matériel auquel l'attaquant a – ou n'a pas – accès, ou si il sera implémenté par un logiciel et si le concepteur a confiance dans la capacité de ce logiciel à garantir la confidentialité des données transitant sur le bus. Dans cette thèse, nous avons proposé des méthodes permettant de vérifier formellement des propriétés de sécurité durant les phases de partitionnement et de conception logicielle. Cette vérification se fait sur des modèles à partir desquels des propriétés de sûreté et de performance peuvent également être vérifiées. Cette polyvalence limite les efforts et les risques inhérents à la réécriture de modèles pour chaque type de propriété. Lorsqu'une propriété est démontrée fausse, la trace établie par l'outil de vérification est transformée afin d'être lisible pour le concepteur, l'aidant ainsi dans la compréhension et la correction de l'erreur.

Dans le cas où les spécificités du matériel nécessitent d'adapter l'outil de vérification, une modification manuelle de l'outil (pour le porter sur une architecture différente par exemple) représente un effort considérable, en particulier dans le cadre des systèmes embarqués où la phase d'exploration de l'architecture repose sur une évaluation rapide des impacts dus à la modification de l'architecture matérielle. Afin d'apporter une réponse à ce problème, nous avons proposé une méthode basée sur un algorithme de transformation d'un code logiciel en une spécification ProVerif permettant d'intégrer un certain degré de modifications matérielles dans la vérification du logiciel.

### Perspectives

Le travail exposé ici ouvre des perspectives dans plusieurs des domaines abordés.

Il serait intéressant de faire le lien entre les propriétés prouvées sur des modèles haut niveau tels qu'ils sont exposés dans la première partie de cette thèse et des implémentation de ces modèles afin d'augmenter la fiabilité des preuves. Pour cela, des méthodes classiques de vérification (automatiques ou semi-automatiques) pourraient être adaptées.

Dans le cadre de l'architecture proposée autour d'Intel SGX, nous avons fourni dans cette thèse une preuve de la confidentialité garantie par le protocole d'échange de clé proposé en utilisant l'algorithme de traduction présenté précédemment. Une autre propriété qu'il serait intéressant d'analyser formellement concerne la validité du protocole d'autorisation d'accès DMA à la mémoire d'une enclave. Moyennant l'ajout à SysML-Sec de capacités à modéliser certains

éléments tels que les tableaux, cette preuve pourrait également être effectuée sur un modèle SysML-Sec en utilisant l'algorithme proposé dans cette thèse.

Bien que nous ayons proposé une évaluation théorique de l'impact sur les performances qu'aurait notre projet de sécurisation des communication dans une architecture Intel SGX, cette analyse théorique gagnerait à être accompagnée de tests pratiques sur un prototype.

Un angle d'amélioration de l'évaluation de sécurité sur des modèles SysML-Sec, serait d'étudier les possibilités de modélisation des propriétés de sécurité sur des diagrammes à un haut niveau d'abstraction. En effet, certaines propriétés telles que l'authenticité ont une sémantique relativement complexe et requièrent une bonne compréhension de l'outil de vérification, ce qui rend l'analyse de sécurité plus difficile pour des concepteurs dont la vérification formelle n'est pas la spécialité.

L'algorithme de traduction de modèles SysML-Sec présenté ici a été conçu dans l'idée de prouver des propriétés de sécurité avec l'outil ProVerif. Cependant, la transformation resterait valide pour d'autres outils acceptant des spécifications en pi calcul appliqué. Ces autres outils, du fait des différentes stratégies et heuristiques implémentées, pourraient éventuellement conclure sur des modèles pour lesquels ProVerif se montre incomplet. Permettre au concepteur de changer d'outil de vérification serait donc une amélioration intéressante de notre transformation. De même, dans le cadre de notre présentation de SMASHUP, il pourrait être intéressant de varier les outils de vérification afin d'éviter au mieux les cas où l'outil ne permet pas de conclure.

Un autre sujet de réflexion autour de SMASHUP concerne la description du matériel. Notre représentation comme ensemble de modules est très haut niveau et la méthode bénéficierait d'une expressivité plus fine. Dans le cas d'une représentation très bas niveau (RTL par exemple), exprimer comment chaque élément affecte la vérification du logiciel semble cependant complexe.

# Abstract

The recent Mirai botnet [152] has shown that the security vulnerabilities commonly found in Internet of Things (IoTs) can be leveraged to mount practical, massive and threatening attacks. As embedded systems become more connected and more involved in critical tasks (e.g., health, driving), the question of how strict security analysis can be performed during embedded system design needs to be thoroughly addressed.

Formal mathematical analysis of a system is the verification method that is able to provide the most complete evaluation of a system as it does not rely on approximations to analyze its behaviour. While formal verification usually requires specific skills and involves a lot human work, it can be considerably reduced by automated formal verification algorithms. In this thesis, we will study how automated formal verification can help embedded system designers in evaluating the impact of hardware and software modifications on the security of the whole system.

One of the specificities of embedded system design—which is of particular interest for formal verification—is that the system under design is described as interacting hardware and software components. Formally verifying these systems requires to take both into account. To illustrate this fact, we propose an example of a hardware/software co-design that would benefit from formal security analysis. This design relies on customized hardware to provide a secure channel between a peripheral and an application running on an untrusted software stack (named trusted path). It leverages the Intel SGX technology which enables to build secure software enclaves on top of an untrusted OS and includes a mechanism to secure the communication between an enclave and a peripheral. Formal verification can be performed on this system at different levels: one possibility is to verify that the key exchange protocol implemented by the various components (enclave, peripheral, processor, trusted third party) is correct from a high-level view (without describing the implementations). The other possibility is to verify that an all powerful software (the OS) running on the customized hardware is not able to access the memory of the enclave. These two cases differ in terms of how tightly coupled the hardware and software components are.

In the first case, the components of the system can be abstracted behind the high-level functionalities they provide. While the security of the system depends on its architecture as it defines how the components will eventually communicate between each other, the fact that a component will be implemented in software or hardware is not relevant for the proof. In order to allow embedded system designers to evaluate the impact of architectural modifications on the security of the system, we propose a methodology to perform formal security verification from design models. To this end, we propose a translation algorithm which transforms embedded system diagrams from the partitioning phase or from the software design phase to a specification suitable for formal security analysis. Integrating formal security verification into a model driven engineering method enables to transform a unique model (described in an extension of SysML in our case) to multiple specifications to prove safety, performance or security properties (with the ProVerif tool). This transformation relies on the similarities between the SysML-Sec software

design modeling language and the ProVerif language. Indeed, in both languages, systems are described as components communicating through channels.

In the second case, modeling the hardware and software components as communicating actors using a defined channel is not possible. Indeed in this case, the customized hardware impacts how software is executed. While it may be possible to model such tightly coupled hardware/software co-designs with interacting entities, the description of the system would not be intuitive. We thus propose a translation algorithm which takes as input a software implementation and a high-level description of the hardware architecture and outputs a ProVerif specification. The hardware description impacts how the software implementation is translated and thus how it is verified. It is thus possible to include hardware customizations either as architectural specificities by changing the hardware description or as custom hardware components by adding new modules to the translation algorithm.

Both of these verification methods were implemented in the TTool toolkit and in the SMASHUP compiler. They both aim at providing intuitive modeling and verification capabilities to integrate automated and reliable security analysis during embedded system design.

# Contents

# Foreword

The content of this manuscript has been presented in various scientific publications over the duration of my PhD thesis. The core of the work I have done has been described in [177–179]. I have also been cooperating with Letitia W. Li for the publication of [165–167].

### International journals

[178] Florian Lugou, Ludovic Apvrille, and Aurélien Francillon. **SMASHUP: a toolchain for unified verification of hardware/software co-designs**. In *Journal of Cryptographic Engineering*, April 2017.

### International conferences

[165] Letitia W. Li, Florian Lugou, and Ludovic Apvrille. **Security-aware modeling and analysis for HW/SW partitioning**. In *5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, February 2017.

[166] Leitia W. Li, Florian Lugou, and Ludovic Apvrille. **Security modeling for embedded system design**. In *4th International Wlrkshop on Graphical Models for Security (GraMSec)*, August 2017.

[167] Letitia W. Li, Florian Lugou, and Ludovic Apvrille. **Evolving attacker perspectives for secure embedded system design**. In *6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, January 2018.

[177] Florian Lugou, Ludovic Apvrille, and Aurélien Francillon. **Toward a methodology for unified verification of hardware/software co-designs**. In *Security Proofs for Embedded Systems (PROOFS)*, September 2015.

[179] Florian Lugou, Letitia W. Li, Ludovic Apvrille, and Rabéa Ameur-Boulifa. **SysML models and model transformation for security**. In *4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, February 2016.

### National journals

Florian Lugou and Ludovic Apvrille. **Quelques épreuves du challenge sécurité Trust the Future**. In *Multi-System & Internet Cookbook (MISC)*, Ed. Diamond, No 79, May/June 2015.

**Under preparation**

Florian Lugou, Aurélien Francillon, and Ludovic Apvrille. **A cryptographically protected trusted path on Intel SGX**. *Under preparation.*

Florian Lugou, Rabéa Ameur-Boulifa, and Ludovic Apvrille. **SysML model transformation for formal security verification**. *Under preparation.*

# Acronyms and Abbreviations

| | |
|---|---|
| AES | Advanced Encryption Standard |
| AES-NI | Advanced Encryption Standard New Instructions |
| AGP | Accelerated Graphics Port |
| AMBA | Advanced Microcontroller Bus Architecture |
| AMD | Advanced Micro Devices |
| AXI | Advanced eXtensible Interface |
| BIOS | Basic Input/Output System |
| CEGAR | Counter-Example Guided Abstraction Refinement |
| CFG | Control Flow Graph |
| CFI | Control Flow Integrity |
| CIM | Computation Independent Model |
| CNF | Conjunctive Normal Form |
| COTS | Commercial Off-The-Shelf |
| CPU | Central Processing Unit |
| CTL | Computation Tree Logic |
| DDR | Double Data Rate |
| DES | Data Encryption Standard |
| DMA | Direct Memory Access |
| DMI | Direct Media Interface |
| DPLL | Davis-Putnam-Logemann-Loveland |
| DRAM | Dynamic Random Access Memory |
| DRM | Digital Right Management |
| DRTM | Dynamic Root of Trust for Measurement |
| EPC | Enclave Page Cache |
| EPCM | Enclave Page Cache Map |
| FIFO | First In First Out |
| FPGA | Field-Programmable Gate Array |
| FSM | Finite State Machine |
| FW | FirmWare |
| GPU | Graphics Processing Unit |
| HDMI | High-Definition Multimedia Interface |
| HL | High-Level |
| HMAC | keyed-Hash Message Authentication Code |
| HSM | Hardware Security Module |
| HW | HardWare |
| IDE | Integrated Drive Electronics |
| IO | Input/Output |
| IOMMU | Input/Output Memory Management Unit |

| | |
|---|---|
| IOTLB | Input/Output Translation Lookaside Buffer |
| LLC | Last Level Cache |
| LPC | Low Pin Count |
| LTL | Linear Temporal Logic |
| MC | Memory Controller |
| MCH | Memory Controller Hub |
| MDE | Model-Driven Engineering |
| ME | Management Engine |
| MEE | Memory Encryption Engine |
| MLE | Measured Launch Environment |
| MMIO | Memory-Mapped Input/Output |
| MMU | Memory Management Unit |
| MPU | Memory Protection Unit |
| NIC | Network Interface Card |
| OMG | Object Management Group |
| ORAM | Oblivious Random Access Memory |
| OS | Operating System |
| PCH | Platform Controller Hub |
| PCI | Peripheral Component Interconnect |
| PCIe | PCI Express |
| PCR | Platform Configuration Register |
| PIM | Platform Independent Model |
| PRM | Processor Reserved Memory |
| PSM | Platform Specific Model |
| PT | Page Table |
| QPI | Quick Path Interconnect |
| RAM | Random Access Memory |
| RISC | Reduced Instruction Set Computing |
| ROM | Read-Only Memory |
| ROP | Return Oriented Programming |
| RSA | Rivest Shamir Adleman |
| SATA | Serial AT Attachment |
| SECS | SGX Enclave Control Structure |
| SGX | Software Guard eXtensions |
| SMART | Secure and Minimal Architecture for Root of Trust |
| SMM | System Management Mode |
| SMT | Satisfiability Modulo Theory |
| SPI | SCSI Parallel Interface |
| SRAM | Static Random Access Memory |
| SRTM | Static Root of Trust for Measurement |
| SVM | Secure Virtual Machine |
| SW | SoftWare |
| TCB | Trusted Computing Base |
| TCTL | Transistor-Coupled Transistor Logic |
| TEE | Trusted Execution Environment |
| TLB | Translation Lookaside Buffer |
| TOCTOU | Time Of Check Time Of Use |
| TPM | Trusted Platform Module |
| UEFI | Unified Extensible Firmware Interface |

| | |
|---|---|
| UML | Unified Modeling Language |
| USB | Universal Serial Bus |
| VM | Virtual Machine |
| VMM | Virtual Machine Monitor |

# Chapter 1

# Introduction

It may be surprising to even think about the discrepancy between hardware and software from a theoretical point of view. Indeed, contrary to a modeling language, a piece of software does not have any meaning in itself. Its purpose is to be executed by a physical machine and the result of a procedure described in software is fundamentally tied to the actual hardware that will be *interpreting* it. The term of *interpretation* does relay this specificity: the piece of software does not have a meaning per se, it needs to be interpreted. And interpreted by something.

On the contrary, hardware—as a physical object—has a defined and arguably deterministic behaviour that is implied by physical laws. The current state of a hardware is determined by a—more or less chaotic—function of their previous state and of the environment. What we see as an indetermination actually comes from the limits of our knowledge about the state of the system or its environment (non initialized memory cells for instance). Some class of hardware—that we call processors or controllers—are however heavily parameterized by a set of rules stored in a specific medium (RAM, ROM, flash memory, etc.). Their observable behaviour changes drastically depending on the state these media were earlier. Thus, describing such a piece of hardware and omitting the rules (that we call software), leaves only few certitudes about the future state of the object.

For the distinction between hardware and software to be relevant, we need something else. Most likely, software developers do not see instructions as parameterizing rules for a specific piece of hardware—which would require to picture the whole decoding and pipelining logic each time an assembly instruction is written—but as abstract instructions which have a well-defined effect on an abstract machine. We will call this theoretical machine an *abstract model of the hardware*. This model is described by a set of objects and a set of properties that relate the software instructions to these objects. Such a property could for instance be that given a description of how an instruction $a$ and an instruction $b$ affect the objects, executing a list of instructions where $b$ follows $a$ will have the same effect (as far as the objects are concerned) as if instruction $a$ was executed and then $b$ was executed. Actually, if we conceptually match the abstract objects of the model to physical parts of the hardware (a set of transistors in the register file for example), this property may well prove to be wrong on most of the recent processors which often implement a kind of out-of-order execution.

We should note that this abstract model could also encompass some of the software rules. It could be an operating system (when an application makes a system call for instance), a library (if the application uses a user-mode thread implementation) or a compiler (if the application is described in a higher-level language). From this point of view, machine instructions are not different from higher-level language instructions. The difference comes from the complexity of

the physical machine that is implementing the model assumed by the software developers.

## 1.1 Embedded systems and formal security verification

The duality of hardware and software is particularly relevant for embedded system designers. Contrary to most higher-level software designers, embedded system designers have to make early architectural choices that will partition the system components into two very different categories (hardware and software). These categories will differ in terms of design methodologies, implementation languages, testing and formal verification methods.

Due to this hardware/software discrepancy, formal security analysis of a system that is partly described as a set of hardware components and a set of software components raises some theoretical and practical challenges.

These challenges have benefited from a growing interest from the academic and industrial communities. We can give some reasons why this is the case. First, formal specification and analysis has only recently started to be acknowledged by the industry as desirable for products outside of the hardware and critical software design industry. This interest is illustrated by the involvement of industrial groups with academic partners working on the subject. This can be either as part of the advisory board of a research project (Microsoft, Amazon or Google in DeepSpec[1]), by dedicating part of their work force to research in this field (e.g., Facebook integration of static analysis into their development cycles [54]) or by commercializing products or services based on an academic project (AbsInt[2] uses CompCert, CSIRO[3] advises industrial groups about seL4). It is also notable that the first common criteria certification level EAL7/EAL7+ were awarded in 2009[4]. Indeed, Common Criteria requires to apply formal verification to prove security functional and assurance requirements on whole systems, as opposed to traditional hardware design verification that targets logical equivalence or safety properties on specific parts of a design [144].

The second motivation that argues in favor of a better understanding of verification of hardware and software components is the pertinence of low-level threats. The reality of widespread malware exploiting low-level vulnerabilities in applications (e.g., shellshock[5]), libraries (e.g., HeartBleed[6]) or even kernel features (e.g., Dirty COW[7]) strengthened the claim that low-level software (as present in embedded systems) should be submitted to thorough security analysis.

This last point leads to the following consideration: the ubiquity of software bugs argues in favor of robust hardware guarantees. Indeed, many recent academic and industrial projects rely on hardware features to mitigate or cancel the effects of software bugs. These features may take the form of software isolation [10], trusted execution environment [77], software attestation [100] or control flow integrity [85] for instance.

Last but not least, the considerable growth of embedded systems and low-energy, highly specialized devices used in the internet of things (abbreviated as IoTs from now on) strengthens the previous reasons by multiplying the potential targets. These systems are ever more present, more complex (increasing their attack surface), more involved in critical tasks (cars, avionics, health) and more connected (and thus more vulnerable).

[1] https://deepspec.org/main
[2] https://www.absint.com
[3] http://ts.data61.csiro.au/
[4] https://www.commoncriteriaportal.org/products/stats/
[5] https://nvd.nist.gov/vuln/detail/CVE-2014-6271
[6] https://nvd.nist.gov/vuln/detail/CVE-2014-0160
[7] https://nvd.nist.gov/vuln/detail/CVE-2016-5195

## 1.2 The problem of accessible formal security verification in embedded system design

While formal verification can increase the trust we put in embedded systems, its application suffers from both theoretical (algorithms to verify systems described as tightly coupled hardware and software) and practical (integrating formal verification to embedded system design process) problems.

**These considerations lead to the following question that we will address in this thesis: how can automated formal verification help embedded system designers in evaluating the impact of customized hardware architecture or components on the security of a system during its design?**

## 1.3 Contributions

This problem can be decomposed in multiple sub-questions: how could hardware customization impact the security of a system as a whole? How does software and hardware interaction affect the ability to perform formal verification? How can security properties be modeled and verified on high-level design models? How is it possible to leverage automated formal verification techniques to provide strong guarantees to untrained designers?

To answer these questions, we made the following contributions during the thesis:

#### An example of a hardware-assisted security solution

As an example of a design where customized hardware needs to be taken into account to verify security properties on the whole system, we proposed to rely on an existing trusted execution environment to provide a secure trusted path to a peripheral. We show how it is possible to efficiently extend the protections provided by trusted execution environments to cover external peripherals when the physical environment of a system is untrusted. We demonstrate this with a system relying on limited hardware customization to enable Intel Software Guard eXtension (SGX) enclaves to securely and efficiently communicate with a peripheral. Compared to similar approaches [257], we assume a more powerful attacker model (close to the one assumed by SGX) and focus on limiting the performance overhead incurred by the additional security features.

#### A formal security verification method for embedded system design models

Integrating security requirements to system design process has taken various forms depending on whether they focus on assets [207], attacks [155] or design models [141]. In this thesis, we show how model-based security approaches can be leveraged to perform automated formal verification to provide real-time feedback to an embedded system designer. Formal verification from models created during the conception of a system enables to reuse design diagrams to verify safety, security or performance properties. We have presented a method to formally verify security properties on extended SysML diagrams [166]. Verification happens during the analysis [167], hardware/software partitioning [165] and software design [179] phases. This method was implemented in a toolkit to enable designing and safety, security and performance verification in the same environment. I personally focused on formal security verification from SysML block and state machine diagrams.

**A verification method for tightly coupled hardware and software components**

The aforementioned method relies on a high-level description of a system as communicating hardware or software components. Likewise, all of the automated verification methods targeting hardware/software co-designs rely, to our knowledge, either on a simple interface to compose hardware and software components [71, 157] or on a full simulation of the hardware [133]. The former may not be suitable for verifying tightly coupled hardware and software while the latter would greatly limit the size of the design to be verified. We have thus proposed a new method [177,178] to show how automated formal verification can cope with hardware customization when verifying tightly coupled hardware and software components. This method was implemented in a tool which automatically translates a system described as a software implementation and an abstract description of the hardware architecture to generate a specification suitable for formal security verification. We designed it so that custom hardware architectures can be easily described and new hardware components can be introduced as modules of the translation tool.

## 1.4 Plan

In this report, we will be first interested in how hardware is able to provide strong security guarantees to a system, even against higher-level (as software) attacks. This will help us understand why, for some systems, security analysis needs to take into account a description of the hardware. We will propose an architecture which illustrates an example of such a system. This design will be helpful both to motivate the work presented in the following sections and to test the results of this work on a concrete example. In order to perform reliable security analysis of such a system—and more generally of any system with hardware and software components—we will then study how formal security verification can be integrated to the design process of embedded systems. First, we will describe a method that leverages high-level design models to verify security properties. This method is suitable to analyze systems where functional components communicate through a simple and pre-defined interface. However, it fails at verifying tightly coupled hardware and software components. This is why we present another method in the last part of this work which targets verification of software and can be parameterized to take hardware specificities into account.

More precisely, in chapter 2, we present a survey of existing techniques related to hardware and software security and formal verification. This chapter shows that the problem we are addressing in this thesis is involving various diverse fields. In chapter 3 we present an architecture to extend trusted execution environments' protection to a trusted peripheral relying on hardware customization in order to illustrate how hardware and software can cooperate to guarantee the security of a system. As verifying such systems is a difficult problem, we present, in chapter 4, a method for formally verifying security properties on SysML software design diagrams. We also explain why this method is only applicable if hardware and software components interact through a well defined interface. In chapter 5, we thus show how formal security verification can be achieved for tightly coupled hardware and software components. Finally, we conclude in chapter 6 and discuss openings and perspectives of this work.

**Acknowledgements**

This work was partly funded by the French Government (National Research Agency, ANR) through the "Investments for the Future" Program reference #ANR-11-LABX-0031-01.

# Chapter 2

# State of the art

Studying environments for formally verifying the security of hardware/software co-designs raises many interrogations which we will split into two categories and address in the two parts of this chapter:

- What is the role of the software abstraction in the security of a design? Indeed, a piece of software always executes in the context of a specific hardware. How can the specificities of the hardware endanger the security of the system? How can they help protecting it? In this section, we will first present studies of attacks abusing the defects of hardware due to the fact that it is implemented in a real physical machine. Secondly, we will present propositions of hardware platform customizations aiming at providing software-level protections.

- How are hardware/software co-designs special when it comes to formally verifying their security properties? How does formal security verification integrate into the design methodologies of hardware/software systems? What are the technical challenges raised by the formal verification of hardware/software co-designs? In this second section, we will discuss design methodologies that propose to integrate security considerations into the modeling steps. Then, we will present various formal verification methods and discuss their suitability for verifying hardware/software co-designs.

Figure 2.1 gives an overview of this presentation and highlights the domains our main contributions target.

## 2.1 Hardware role in software execution integrity

In this section, we are interested in the role the distinction between hardware and software may play as far as security is concerned. This boils down to two main domains:

- First are attacks that take advantage of the fact that a piece of hardware does not guarantee a property of the model that the software developer assumed to be true. Such an attack would typically enable the attacker to redirect the execution flow of a program or make secret data leak. This kind of erroneous assumption may come from a misunderstanding of the hardware guarantees on the side of the software developer (e.g., cache timing side channels or low-level characteristics abstracted away in the model of the hardware [22]), or from an undocumented behaviour of the hardware (e.g., rowhammer [147]).

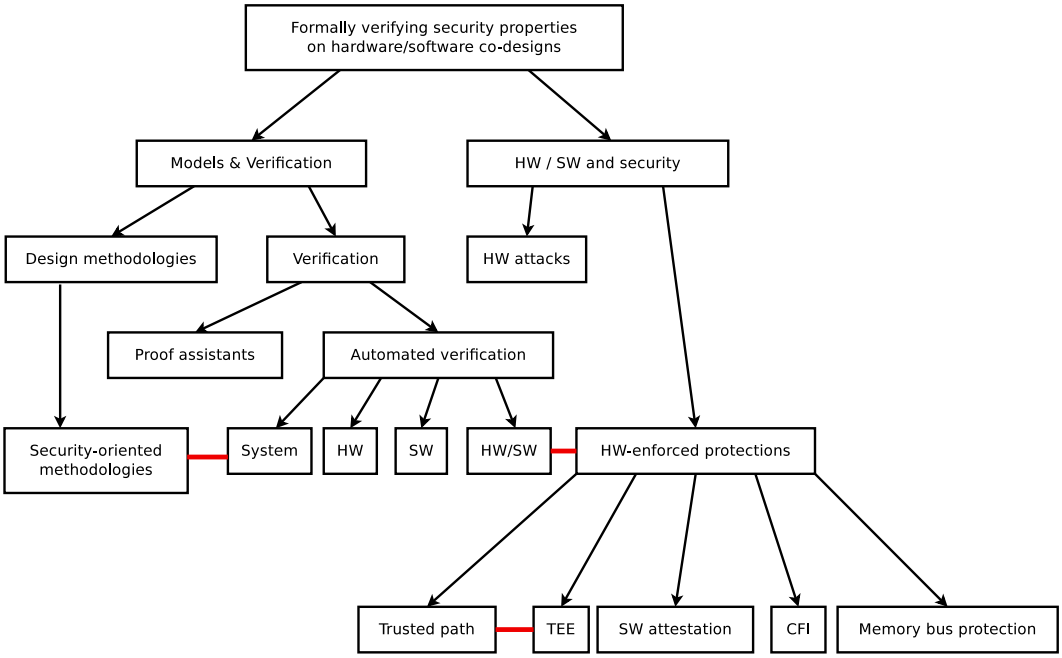



Figure 2.1 – Our contributions and their context: establishing new links.

- Then, some other works present new models (and potentially physical machines that conform to these models) that guarantee properties that are useful to software writers to protect assets or mitigate potential flaws.

As far as hardware-assisted architectural security features are concerned, we split our presentation into three subsections: features that help in guaranteeing the integrity of the control flow of a piece of software, features that attempt to protect the confidentiality of the data handled by the software, and lastly, we dedicate a part of this chapter to isolated execution—which would otherwise partly fall into both of the other two subsections—as it is of particular importance for a part of our work.

### 2.1.1 Hardware-based attacks

The purpose of a hardware-based attack targeting a software procedure is to deliberately provoke and exploit a behaviour that was outside of the expectations of the software designer. We do not call these attacks *hardware-based* because they involve a kind of manual intervention from the attacker. We say they are hardware-based because they rely on a—potentially well-documented—behaviour of the real hardware that is not modelled. Note that these attacks are not exclusively limited to attacking procedures implemented in software. It could also target blocks of hardware logic. In this case, the discrepancy is between the real piece of hardware and the model that the designer had in mind when implementing the algorithm (in a hardware description language such as Verilog for instance).

#### 2.1.1.1 Side-channels

Side-channel attacks exploit information produced by the execution of an algorithm on a piece of hardware to deduce a knowledge about the algorithm or the data it handles. Even though the first presentation of timing attacks by Paul Kocher in [149] is now 20 years old, the subject is still an active research field as new side-channels are exposed and infrastructure sharing bring potential attackers close to their targets.

**Different types of side-channels**

A natural way to classify side-channel attacks is to look at the type of side-channels they exploit. In [149], which is widely considered as the starting point of this type of cryptanalysis, Paul Kocher presented an attack exploiting timing variations in an implementation of modular exponentiation. Since the computation time depends on the bits of the exponent and on the base, the attacker is able to make two different timing models depending on whether a specific bit of the exponent is set or not. These models associate to each possible base a time that the algorithm should take. Given sufficient timing measurements, the attacker is able to choose which of the two models best matches the measured results, and thus deduce the bit of the exponent.

Timing-based attacks are still widely studied since, contrary to many other side-channels, measuring time is easily doable even for a non-privileged software application. Moreover, getting rid of the time variation is hard since many implementation details introduce data-dependent timing variation. For instance in [4], Acıiçmez et al. show how an attacker could recover the exponent in a modular exponentiation if she is able to predict if a branch misprediction will happen given a known base and a number of bits known in the exponent. Likewise, a lot of work has focused on key-dependent memory accesses (when a cipher uses precomputed tables for instance) as memory accesses duration is heavily impacted by some architectural features like caches. Cache attacks were mentioned as soon as 1998 in [142] and implemented in [29, 43, 200, 201, 249]. More recently, cache attacks have been proved to be effective across virtual machines [272] and across physical cores [138, 173].

Another type of side-channel attack pioneered by Paul Kocher is related to power consumption [150]. Indeed, as an algorithm is eventually implemented using semiconductor logic gates and transistors, monitoring the electrical consumption of a system gives hints about how many transistors have switched at a particular step of the algorithm. As the size of the system impacts the signal-to-noise ratio, these kinds of attacks have mostly been successfully mounted against smartcards or specific cryptographic algorithms implemented on FPGAs [61, 233].

Other side-channels that have been exploited include electromagnetic waves [109, 175, 210], acoustic waves [110] or photonic emission [56].

**Countermeasures**

Countermeasures to side-channel leakage mostly depend on the type of side-channel we are trying to silence. Theoretically, these mainly fall under three domains: getting rid of the data-dependent variation, masking the input data before applying the algorithm or decreasing the signal-to-noise ratio.

Creating constant-time algorithms [115] is easier than performing a computation with constant power consumption [246], although getting rid of the time variation induced by architectural features such as caches is still an open research subject [46, 153, 251]. Likewise, decreasing signal-to-noise ratio depends on the type of side-channel we are interested in. It could be achieved by performing random computations, waiting for random cycles or shuffling [252].

One of the most researched countermeasures is masking. Indeed, a single masking implementation may be effective against various kind of side-channel attacks. Masking finds its origins in blinding schemes that enable an algorithm to compute a transformation on derived data without seeing the actual original data. The result of the transformation applied to the original data can then be deduced by reversing the derivation function applied to the original data and by applying it to the result of the algorithm. It was already presented as a possible countermeasure in the original paper from Paul Kocher [149] for Diffie-Hellman and RSA. It was then applied to DES and AES in [7], which is more difficult because the S-boxes are non-linear functions. Since then, masking techniques have continuously be researched to improve their reliability [196] and efficiency [118, 209].

#### 2.1.1.2 Fault injection

**The attack**

Contrary to side-channel analysis which is a passive technique (at least during the algorithm execution), fault injection attacks rely on a—most likely intentionally provoked—hardware fault to induce a behaviour that was unexpected by the designer. The first reported—theoretical—fault injection attack was published by Boneh et al. in [42]. In this paper, the authors showed how RSA signing algorithm could leak erroneous data that would enable to factor a public RSA key if a hardware fault were to happen during the computation of the modular exponentiation using the Chinese remainder theorem.

Since then, community has researched ways to induce faults: by modifying the environment through time glitches [213], temperature or power variation [23, 24, 135], electro-magnetic variation [92] or white light exposition [221]; by precise and reliable fault inducing techniques with lasers [5]; by unintended hardware behaviour (due to physical phenomenon as in [147] or due to hardware trojans introduced during the foundry stage [111, 172]). Fault injection attacks have also been successfully applied to different algorithms [34, 105, 112].

**Countermeasures**

Countermeasures to fault attacks can first be passive or active hardware protections that attempt to prevent the fault injection method or to detect it. For instance, circuits may include a voltage regulator which filters out rapid power supply changes and compares the output of the voltage regulator to the external supply to detect a potential attack. Another more common countermeasure is to compute redundant information and compare it to the result of the algorithm before outputting the result. The redundant computation could be implemented either in software or hardware and often takes the form of error detection codes [247].

#### 2.1.1.3 Others

In the last two subsections, we have presented side-channel analysis and fault injection attacks as they are currently the most studied hardware-based types of attack. However, many other attacks may fall under this category. Static ones: probing in order to spy on a bus or memory, reverse-engineering of a chip (which includes de-packaging, layout reconstruction, memory content recovery). Dynamic ones: cold boot attack [125] (resetting the computer and taking advantage of the short time during which memory retains its content even when power is not supplied to dump it), evil maid, etc.

#### 2.1.1.4 Conclusion

We have described these attacks briefly because they have all proved to be very efficient to attack cryptographic algorithms and reveal secret data. As such, from the point of view of software security, they should not be underestimated and special care should be taken to implement effective countermeasures.

These attacks help in understanding how the analysis of the security of a system is complicated by taking into account the hardware specificities. In particular, this section has shown that formally capturing these attacks requires to work on a very low-level model of the hardware. While the perfect theoretical verification method would take into account the system down to an atomic view of its hardware, a trade-off needs to be made between the size of the system and the range of the proof.

### 2.1.2 Data protection

Some of the countermeasures to hardware-based attacks that have been proposed require specific hardware modifications. This is for example the case of the special gates introduced in [246], whose power consumption is independent of their input data and that are intended to thwart differential power analysis.

Another mentioned attack whose countermeasures are mostly implemented in hardware parts is probing. Although no effective solution exists yet to guard against a powerful enough opponent ready to invest enough time and money to de-package and modify or probe an integrated circuit, securing easily accessible memory and buses should still be seriously considered [134].

#### 2.1.2.1 Memory bus protection

The processor-memory bus is indeed a critical medium when it comes to protecting either a software-implemented algorithm (to protect intellectual property) or the data it handles as both machine codes instructions and data will one way or another be transmitted over it. Small flash memories may be present on the processor die for performance or security purpose but their size is limited by their price. Bypassing software memory protections to get direct access to memory can be achieved in multiple ways, even when the OS kernel is trusted:

- by physically probing the processor-memory bus,
- by using a peripheral to access memory through DMA when an IOMMU is not present or when it was misconfigured,
- or by using cold boot attacks [125].

There are three possible ways an attacker could take advantage of an unprotected processor-memory bus:

- Spying on the data that is transmitted
- Injecting data to the processor
- Spying on the addresses that are accessed by the processor

These three attack vectors are independent and should be addressed separately.

### Confidentiality

In the last 20 years, many architectures have been proposed to protect the secrecy of data stored in off-chip memory. Most of them rely on encryption of data when it leaves the processor die and decryption when it is fetched by the processor.

Although some of the proposed architectures rely on software implementations to avoid modifying the hardware (as in [62]), most of them are based on a hardware encryption module that interfaces between the memory controller and the last-level cache in order to reduce the performance cost of encryption (the authors of [62] report a 37% slowdown with 512KB L2 cache).

One of the first system architectures proposing RAM encryption was the eXecute-Only Memory (XOM) architecture [170]. On cache miss or cache line eviction, a routine implemented as microcode is in charge of encrypting or decrypting cache lines so that only encrypted data are transmitted over the processor-memory bus. A similar bus protection scheme was implemented for many other projects [49, 58, 78, 90, 96, 99, 120, 243].

To prevent semantic analysis of the encrypted data, it is also important that the same data stored on multiple addresses, or stored at the same address on different point in time, does not result in the same ciphertext. This is why the encryption algorithm must use a spacial and temporal dependency that alters the result of the encryption (in the form of a block cipher with an address dependent initialization vector for instance).

Although all of these memory protection schemes have their particularities, the most important differences are:

- The encryption algorithm used. The choice of this algorithm greatly impacts the performance of the system. In [265], Yang et al. have shown how one-time pad encryption could improve XOM memory read performance by taking advantage of idle memory controller cycles by computing the pad while waiting for the ciphered data to be available on the memory bus. However a specific pad should only be used once so they propose to make it dependent on a sequence number that needs to be stored on chip, which is expensive and not scalable. From this point of view, SGX [96] interestingly uses the version needed to check the integrity of the data (as explained further down) as a counter in a kind of AES counter mode.

- The granularity of the control available to specify which data should be confidential. XOM [170] allows to set flags on cache lines to specify whether it should be encrypted, Bastion [58] and SecBus [49] allow for page-granularity control. SGX [77] encrypts a whole memory range (configurable in BIOS).

- The ability for software to control the memory protection (security policies, keys). Most of the proposed architectures assume that the OS (or a large part of it) is not trusted so they propose very limited possibilities to manage the protection process from software. This is notably not the case for SecBus [49] whose hardware security module is driven by a software security manager.

### Integrity

Encrypting data as it is written to RAM is not enough to guarantee the confidentiality of data. A famous example of this is presented in [158]: by injecting guessed encrypted instructions on the memory bus and observing the CPU reaction, Kuhn was able to reconstruct enough correct instructions to dump the content of the memory to the parallel port of the DS5002FP.

This shows that the integrity of the data stored in memory should also be protected. To do this, most of the memory protection schemes store cryptographic MACs or hashes of memory

chunks and check it when data is fetched from memory. A first naïve approach would be to store both encrypted data and a corresponding MAC in memory. This raises two issues:

- First, an attacker is able to switch two encrypted memory chunks and their respective MAC. This attack is called *splicing*.
- Also, an attacker can record a memory chunk and the associated MAC and replace a later value of this memory chunk by the recorded one. This attack is called *replay*.

Preventing splicing attacks is easy: including a location-dependent value inside the MAC prevents the attacker from switching its location. XOM [170] uses the offset of the memory chunk in the virtual address space of the process so that the software developer—which does not know the physical address this virtual address will be mapped to—is able to distribute an already encrypted software. Most of the others use (part of) the physical address.

The second issue is much harder to fix. Indeed, contrary to virtual or physical addresses, the processor does not associate a time-dependent value to memory chunks. If we want to create such a *version* number, it needs to be protected so that the attacker cannot change it to replace both memory contents and the version with an older version. A first idea would be to store them on the processor chip, but this does not bring any advantage compared to directly storing the MAC value on-chip (both would roughly have the same size). It appears that storing the MAC values on an on-chip SRAM was actually implemented by Microsoft to protect the microkernel of the Xbox 360 [240]. However, SRAM is expensive and such a scheme is not scalable (SRAM size must be adapted according to the size of the external RAM).

On the other hand, a solution would be to hash the content of the entire memory and store the resulting hash on-chip. This would require to read the entire memory for every memory read or write and is not acceptable from a performance point of view. The solution adopted by most of the systems providing protected memory is a trade-off between the size of on-chip memory and the time needed to perform integrity checks. To do this, a hash tree—or *Merkle tree* [187]—is built. The leaves of the Merkle tree are the memory chunks whose integrity we want to protect and each internal node of the tree is computed by hashing the children of the node. The root is stored on-chip so that the attacker cannot modify it. This way, any modification on the data and/or on an internal node of the tree would be detected by checking the validity of the hashes on the path from this node to the root of the tree.

When reading or writing data in memory, the tree is checked and updated in case of a write. Changing the memory chunk size and the arity of the tree enables to balance the size used by the hash tree and the performance of the verification (and thus of the memory accesses). These parameters are therefore important for the overall performance of the design [259].

#### Addressing pattern protection

Even if the confidentiality and authenticity of the data transmitted on the memory bus is protected, an attacker is still able to get valuable information by probing the address signal of the memory bus. Snooping on the bus enables the attacker to learn the control flow of the program on a relatively coarse-grained precision. Indeed, caches hide memory accesses targeting already cached memory addresses. Yet, access pattern monitoring attacks have proved to be effective even when the granularity of the accesses is coarse. For instance in [235], Shinde et al. show how a malicious OS can reduce the key space by a factor of $2^{25}$ in a particular implementation of AES just by monitoring page faults.

Protecting from access pattern monitoring attacks has been studied for a long time and most of the proposed defense rely on an implementation of Oblivious RAM (ORAM) which was introduced in [114] by Goldreich. Many ORAM implementations have since been proposed

[104, 180, 276] and regular improvement on memory accesses overhead is reported by various studies [68, 160, 239].

Efficiently combining Oblivious RAM with memory integrity and confidentiality protection remains an open problem.

### 2.1.3 Control flow protection

In order to illustrate the difficulty of verifying hardware-software schemes, we present two important classes of architectures that take advantage of hardware features to guarantee properties on the software.

#### 2.1.3.1 HW-assisted control flow integrity

Runtime software exploits traditionally attempt to divert the original control flow of a program in order to execute a stub (crafted by the attacker or by reusing already present code) and launch an attacker controlled program (as a shell). In a usual scenario, once a vulnerability is discovered, the attacker attempts to use it to set up the stack and modify a code pointer (function pointer, return address) that will divert the control flow of the program.

Original exploits use to rely on injected code so that the modified code pointer would jump to this attacker-crafted piece of code. Today, most systems defend against such exploits through two mechanisms: detecting that a return address was modified by inserted *stack canaries* and using flags on memory pages so that a page can't be both writeable and executable (this technology is sometimes called *Data Execution Prevention*, *No eXecute*, W $\oplus$ X).

These defenses effectively protect against code injection attacks. However, an attacker could still divert the control flow of a program to target already existing code. Today, most state-of-the-art software runtime exploit use a kind of code-reuse attack named Return Oriented Programming (ROP) which was introduced in [230]. ROP relies on finding a Turing-complete set of small instruction stubs in already existing code that end with a return instruction (or an indirect jump [59]). By carefully setting up the stack with correct return addresses, an attacker is then able to chain these *gadgets* to get the expected behaviour.

Defending against these code-reuse attacks has proved to be much more difficult than preventing direct code injection. Address Space Layout Randomization makes it harder for an attacker to reliably locate gadgets but memory leaks are pretty common and cancel the effect of this protection.

**Control flow integrity**

Control Flow Integrity (CFI) is one of the most studied countermeasures to code-reuse exploits. Abadi et al. introduced it in [1] and CFI has since become a very active research area. CFI relies on an off-line analysis of the software code in order to create a Control Flow Graph (CFG)—that is, a graph in which vertices are instructions of the program and edges between two instructions mean that these instructions can be chained in a valid execution of the program. Once the CFG is constructed, CFI attempts to prevent the running program to follow paths that are not valid paths in the CFG. To do this, [1] proposes to rewrite binaries to add labels on instructions targeted by a control flow instruction and software runtime checks that verify the target label before using a control flow instruction. There are two kinds of control flow instructions: forward-edges (`call`, `jmp`) and backward-edges (`ret`).

This approach however raises some issues. First, extracting the CFG from the program may be harder than it seems. Indeed, a good code-reuse defense would need to be applicable to compiled binaries, even when they are stripped of their symbols. And mainly, the performance

overhead induced by software-based control flow integrity is prohibitive for most usage. Typically, the worst-case overhead reported by Abadi et al. in [1] was over 50% on SPEC benchmark. Further works have reduced it (worst-case overhead of around 25% for [268], of 15% for [83]) by optimizing the original scheme.

In order to further reduce the overhead, some works have presented a *coarse-grained* version of CFI. Instead of guaranteeing that the execution paths strictly conform to existing paths in the CFG, these solutions typically identify call and jump targets in the binary and ensure that the control flow instructions only target valid destinations [39, 255, 269, 270]. Other solutions [64, 202, 261] have proposed to leverage existing hardware features (typically provided by CPUs for performance monitoring purpose) to periodically check that the program conforms to some CFI policies: that return targets are preceded by a call instruction, that call targets are valid destinations in the CFG or that there is at least one long sequence of instructions among the last executed stubs (before reaching a return).

However, these coarse-grained CFI policies have proved to be insufficient to prevent ROP attacks: [55, 87, 123]. These papers have shown that even if targets of jumps and calls are restricted to valid targets in the CFG and if return targets are limited to instructions that follow a call site, it is still possible to find a Turing-complete set of gadgets in common programs like Internet Explorer or Acrobat Reader.

#### Hardware-assisted control flow integrity

In regard of this, some solutions have recently been proposed to provide fine-grained CFI with little overhead. All of them [67, 84–86] rely on custom hardware modifications to speed up the control flow verification (which had already been proposed before: [50, 271]).

In this section, we propose to give a brief overview of HCFI as presented in [67]. At the time of writing this report, HCFI is one of the latest hardware-based solution aiming at providing CFI with reduced overhead. It will serve all along this presentation as an example of how hardware and software intricacy may complicate formal analysis of the security of a system.

HCFI (which stands for Hardware-enforced Control-Flow Integrity) implements forward-edge and backward-edge CFI by relying on a set of custom architectural features:

- 4 CFI-related instructions,
- an unmapped memory area used to store a *shadow stack* and
- an unmapped memory area used as a *shadow register*.

The authors of [67] have implemented a prototype HCFI by modifying the Leon3 SPARC V8 Softcore and synthesizing it on a Virtex 6 FPGA board. They choose to allocate $128 * 32$ bits for the shadow stack and 32 bits for the shadow register. The 4 instructions that were added to the SparcV8 instruction set are `SetPC`, `SetPCLabel`, `CheckLabel` and `CheckPC`.

As proposed in the original implementation [1], forward-edges control is enforced by a system of labels. An off-line analysis of a binary reconstructs its CFG and adds a `CheckLabel` instruction at each point of the program that can be the target of an indirect jump. A label is provided with the `CheckLabel` instruction. All the instructions that may jump to this location need to specify this label with a `SetPCLabel` instruction before jumping to this location. The `SetPCLabel` is placed in the *delay slot* of an indirect jump location. The delay slot is a common RISC feature that helps in reducing the cost of branch shadows. On some architectures, the instruction placed right after a branch is always executed, no matter whether the branch is taken or not. `SetPCLabel` stores the label passed to it in the shadow register and makes the core transition to a state where the only accepted instruction is a `CheckLabel` with the label stored in the shadow

register. If another instruction is met, core transitions to a *Control Flow Violation* state which corresponds to the Leon3 Error Mode of the Integer Unit. Likewise, if a `CheckLabel` is met and it was not preceded by a `SetPCLabel`, the core is put in the Control Flow Violation state. To enable locations to be the target of both indirect and direct jumps, the `SetPC` instruction is used in the delay slot of direct jumps to negate the fault detection if the next instruction were a `CheckLabel`.

Backward-edges CFI is enforced by using a shadow stack, as other—software or hardware-based—CFI solutions do. Each time a `SetPC` or `SetPCLabel` instruction is executed, the current Program Counter (PC) is pushed on the shadow stack. During the off-line analysis, a `CheckPC` instruction is added in the delay slot of every return instruction. This instruction compares the value stored at the top of the shadow stack and the PC corresponding to the following instruction (after the return). If they don't match, the system is put in Control Flow Violation state.

#### 2.1.3.2 Software integrity verification and software attestation

Once a software vulnerability has been successfully exploited (due to a failure or the absence of CFI), the attacker often injects malicious code that may reside in the target machine over time or even across reboots. Complementary protection solutions have thus be envisioned to enable to measure the content of part of the memory. Measuring part of the memory can be done:

- when the system boots to allow only signed software to take control of the computer. This is called *secure boot*.
- when the system boots to measure the software stack which is executed. This is called *static root of trust for measurement*.
- when the system has already booted to create an isolated section of memory. This is called *dynamic root of trust for measurement*.
- when the system has already booted to produce a trusted measurement of part of the system. This is called *software attestation*.

The eventual purpose of these solutions is to provide a tangible proof that the software code that is (or is going to be) executed is in a correct expected state.

The solutions that have been proposed (and implemented in COTS architectures for some) can be split into two main categories: solutions that require a reset of the software stack to build an environment that is not compromised and those which do not.

##### Trusted measurement of software after reset

Secure boot mechanisms have now been widely adopted in the industry [15]. They rely on a trusted initial procedure (the *root of trust*) which is often a small bootloader, usually in a mask ROM and which computes a hash of the next software part to be executed and compares it to a reference hash value (whose integrity is also trusted). In turn, the next link of this *chain of trust* will measure another part of the memory before transferring control to it, and so on. It is important to note now that the *trusted* part of the chain—the root of trust here—is called "trusted" because it **needs** to be trusted, without judgement over whether this trust is misplaced or not. One of the reasons it is trusted could for instance be that the root of trust procedure and the first hash are stored on ROM. However in this case, it will prevent even non-malicious agents (as the owner) from updating any part of the software stack that belongs to the chain of trust.

To overcome this issue, it is possible to allow rewriting the root of trust through a controlled mechanism (like a hardware switch) or to delay verification: the user (let us call her the *verifier*) is in charge of checking that the list of hashes measuring the different software parts is indeed valid. This mechanism is called static root of trust for measurement. To perform this measurement chain, the Trusted Computing Group introduced a specification for a hardware module named Trusted Platform Module (TPM) that is in charge of computing hashes on memory ranges (actually on measurements of memory ranges), providing a signed value (quote) representing the combination of the successive measurements since reset and securely storing the key used to sign the measurements. Each software element of the boot chain measures the next element and extends the value stored in the TPM with the new measurement. At the end of the boot process, the value stored in the TPM is bound to the TPM through the key used to sign the measurements and to the measurements that were computed since boot (and the order they were done).

These solutions enable to create an environment that is not compromised (or attest that an environment has not been compromised) on a machine that has just been reset. However, rebooting an embedded system to start from a clean environment may often be impractical or impossible. Some mechanisms have thus been presented in order to allow a machine (traditionally called the *prover*) to prove to the verifier that it is running an untampered software.

Among those mechanisms, some rely on specific hardware (such as the TPM) and some attempt to achieve software attestation with software-only schemes (software-based attestation). We will first discuss the latter and then argue that the recent focus on hardware-based solutions acts the defiance of the community toward reliable pure software-based attestation.

#### Software-based attestation

The specificity of pure software-based software attestation—which originated with the presentation of the Genuinity scheme by Kennell et al. [143]—is that the verifier needs to recognize a malicious prover, even when the whole software stack is compromised and without relying on specific hardware. Thus, no secret shared between the prover and the verifier can be relied on, which rules out cryptographic schemes. To achieve software attestation under these conditions, the verifier has to rely on generic hardware side-effects. The prover is asked to perform an attestation algorithm that is carefully crafted so that feigning the attestation would provoke side-effects that would be noted by the external verifier (which only has a black box access to the prover). Most commonly, software-based attestation schemes observe the time taken by the prover to execute the attestation routine and compare it to a threshold computed based on the physical properties of the prover. Time is an interesting side-effect for software-based attestation as it does not require direct access to the prover. Most of the schemes [156,159,169,226–229,231] rely on a challenge-response protocol where the verifier requires the prover to compute a hash on (parts of) its memory and the verifier measures the time elapsed between the moment the challenge was sent and the moment the response was received. The verifier can compare the hash to a measurement of the expected memory content and she is able to detect that the attestation routine was modified if the measured time is higher than a fixed threshold.

Note that all these schemes require that the verifier has a precise idea of the hardware characteristics of the prover. In particular, the verifier should be convinced that the attestation scheme has really been executed by the prover and not by a third-party device with different hardware characteristics (a faster processor typically). Such an attack is called a *proxy attack*. Some of the schemes thus require the attestation scheme to be executed in a Faraday cage to prevent external communication [159], use physically unclonable functions to distinguish the genuine prover [154, 222] or attest all faster nodes in the network before attesting the targeted

device [169]. Another requirement to software-based attestation is that the verifier is not mistaken in her understanding of the observed side-effect. Indeed, it should be hard for an attacker to execute a modified version of the attestation scheme with no noticeable side-effect. Thus, targets of software-based attestation are mostly simple architectures (limited instruction set, no DMA, interrupts can be easily disabled, relatively simple cache architecture), often sensor network devices.

Apart from proxy attacks (which are often ignored in the proposed schemes [226–228]), basic attacks on naive implementations of software-based attestation consist of:

- Pre-computing the hash before the attestation is requested. This is commonly defeated by including a nonce in the verifier request.
- Copying the original data to another location and modifying the attestation routine so that this location is queried instead when hashing the modified data. This attack can be detected if the checks added to the attestation routine incur a sufficient slowdown. To prevent the attacker from guessing when the modified data will be hashed, pseudo-random traversal is often used. It is also possible to limit the available free memory space (by adding random data to the memory attested for instance).
- Using a faster implementation of the attestation routine so that the added instructions are not detected. Attestation routines must thus be carefully designed so that they cannot be optimized (or be parallelized if the prover is helped by another device).

Since software-based attestation schemes were proposed, many researchers [57, 232] have expressed doubts about the possibility to prevent these attacks. In particular, guaranteeing that the attestation routine cannot be optimized and that the attested memory cannot be compressed (or that doing so would incur a noticeable time overhead) is difficult, especially when the timing measurements are noisy (for instance when the prover and verifier interact through a multi-hop network). These doubts have motivated researchers to study hardware-assisted software attestation to provide a dynamic root of trust.

#### Hardware-based attestation

As mentioned earlier, the TPM is a hardware module that was initially used to provide measurements of software elements of the boot chain in order to establish a cryptographically provable static root of trust. In the specification 1.2 of the TPM, new Platform Configuration Registers (PCRs) were added to the existing 16 PCRs used to store static (boot-time) root of trust measurements. Contrary to the first 16 PCRs, these ones can be reset without requiring to reboot the hardware by using a special instruction. This specification enables to build a scheme relying on the TPM to provide dynamic attestation capabilities. Such a scheme was implemented by Intel and named Intel Trusted Execution Technology (Intel TXT). Intel TXT introduces new instructions to create an environment were an isolated and Measured Launch Environment (MLE) can run. The `SENTER` instruction resets the PCR dedicated to the Dynamic Root of Trust for Measurement (DRTM) and stores in the TPM the measurement of a software module (named SINIT ACM) which is then called. This module disables Direct Memory Access (DMA) targeting the memory of the MLE and measures it. The MLE can then require a quote from the TPM to guarantee both the SRTM and DRTM.

A class of attacks particularly dangerous for software attestation schemes is Time-Of-Check-Time-Of-Use (TOCTOU) attacks: guaranteeing the integrity of a piece of software is only interesting as far as it is possible to *atomically* verify and run the measured procedure. By atomically, we mean that control cannot be passed to another (potentially corrupted) software that would

be able to modify the procedure after it was measured but before it is executed. As a matter of fact, Intel TXT has been shown to suffer from a kind of TOCTOU attack. Indeed, Wojtczuk et al. showed in [260] that an attacker may be able to corrupt the MLE after it is measured by using a corrupted System Management Mode (SMM) as SMM is not measured during the DRTM establishment and SMM can be called once MLE is launched if a System Management Interrupt is received. Intel published a new specification for a SMM Transfer Monitor in charge of controlling the interactions of the SMM [266] and presented a new generation mechanism, Intel SGX [183], to overcome the limitations of Intel TXT (security, performance).

TPM-based solutions may not be suitable for low-power embedded devices such as sensors. Since software-based attestation relies on questionable assumptions, some solutions were presented to guarantee strong software attestation schemes at the cost of few hardware modifications [78,100,151,183,198]. All of them rely on hardware-enforced isolation to protect a software or microcode procedure that serves as a dynamic root of trust and initiate the measurement chain that allows to measure the software to attest.

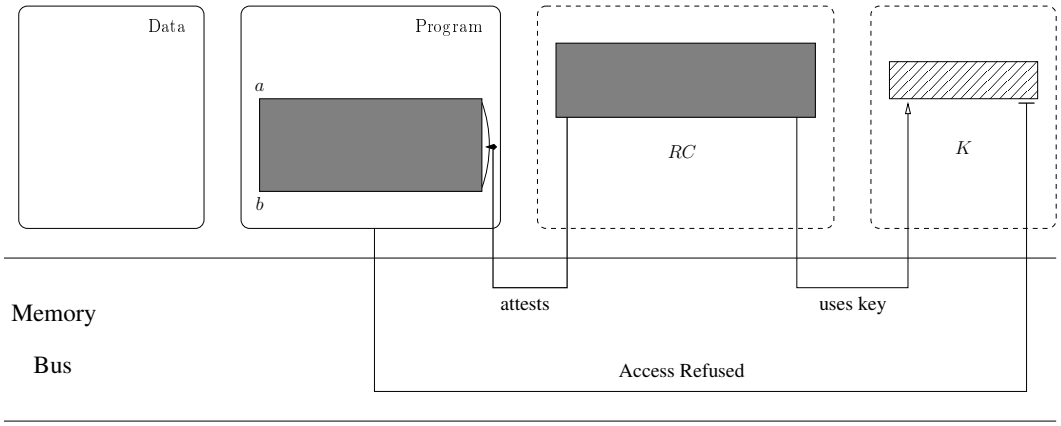



Figure 2.2 – SMART overview.

As another example design where hardware and software components interact to provide security properties on a piece of software, we propose here a short description of one of these designs: SMART, which stands for Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust and has been presented in [100]. This primitive tackles the problem of remote attestation by relying on a slightly customized micro-controller unit and a critical routine stored in ROM.

In SMART, the memory layout is augmented by adding two read-only sections (as presented by dashed lines in Fig. 2.2). The first one contains a procedure referred to as $\mathcal{RC}$ and the second one contains a key $\mathcal{K}$. $\mathcal{RC}$ can be called in order to compute a keyed-hash message authentication code (HMAC) on a memory range $[a, b]$ passed as an argument. This HMAC relies on a key $\mathcal{K}$ to prevent a compromised software from computing the hash itself. The modified processor enforces that $\mathcal{RC}$ procedure cannot be entered in the middle nor exited from the middle of the code. Moreover, $\mathcal{RC}$ is designed to be constant-time to prevent side-channel analysis and it disables interrupts to prevent the key from leaking if the procedure was interrupted. Likewise, memory is sanitized at reset to prevent an attacker with physical access to the prover from learning the key by rebooting the device while $\mathcal{RC}$ is running. Once the measurement procedure has been run, the measurement is stored in available RAM and control is passed to the measured memory region. Note that the measurement also includes:

- a nonce to prevent replaying previous attestation,
- the addresses of the measured memory region,
- the address of the memory location where $\mathcal{RC}$ should jump when it ends,
- the parameters to be passed to the measured code.

Software attestation and processor-memory bus protection may be part of a more holistic security solution to protect a piece of software running on an untrusted machine where the whole software stack may be corrupted and an attacker may have physical access to the device and launch relatively simple hardware attacks. We present some of these solutions which are known as *trusted execution environment* (TEE) and discuss the problem of accessing peripherals under such a powerful attacker model.

### 2.1.4 Isolated execution

As mentioned in the introduction, the development of complex architectures with large privileged software stacks justifies the need to protect applications even when their—privileged—software environment is potentially corrupted. Moreover, the success of Infrastructure-as-a-Service (IaaS) providers (such as Microsoft Azure or Amazon EC2) and the challenges raised by Digital Right Management (DRM) schemes have shed a new light on a special attacker model that used to specifically target systems on chip: the owner of the hardware is untrusted. The protection schemes need to consider that an attacker has access to the physical hardware and can mount sophisticated attacks against it. However, it may be argued that an attacker would only invest as much money and time as the asset she is targeting is worth. This trade-off between the value of the data stored on the device and the investment to protect it has progressively depicted a powerful, but limited, attacker model. The attacker typically has a physical access to the device, controls most of the privileged software stack and can reboot the device or plug new peripherals but she does not invest enough money and time to be able to probe communication channels internal to the processor die.

#### 2.1.4.1 Trusted execution environment

TEE are architectural solutions that aim at providing isolated execution on an untrusted platform. They may enforce various security properties that show variations in the assumed attacker model:

- reporting on the value of a memory region that has been loaded,
- controlled enter to and exit from a protected memory region,
- protection of a memory region against modifications by other (potentially privileged) software,
- protection of a memory region against software access originating from another (potentially privileged) software,
- protection of a memory region against leakage and against modification by an attacker with physical access to the DRAM (and more generally, to all the peripherals outside the processor die),
- protection against addressing pattern leakage in presence of an attacker with physical access to the processor-memory bus and

- protection against side-channel attacks (cache timing attacks particularly).

In the rest of this section, we will discuss how these properties are handled by the different existing schemes (in particular: XOM [170], Aegis [243], Bastion [58], Intel SGX [183], Sanctum [78], Overshadow [63], PodArch [236], InkTag [128] and TrustLite [151]). As all these schemes refer to the protected memory region with different names, we will use a common denomination here: *trusted application*. Note that this comparison was greatly facilitated by the previous survey work of Costan et al. in [77].

### Isolating from untrusted software

Arguably, the main purpose of trusted execution environment is to isolate the trusted application from other (potentially privileged) applications. In a traditional setup, the Memory Management Unit (MMU) or Memory Protection Unit (MPU) is responsible for isolating memory regions. However, this protection mechanism is controlled by the Operating System (OS) or hypervisor and thus is not adapted in a scenario where they are untrusted. Overshadow [63], InkTag [128] and Bastion [58] propose to use a trusted hypervisor so few (or no) modifications on the MMU/MPU are needed. The trusted hypervisor ensures that the page accessed (typically on Translation Lookaside Buffer (TLB) misses) belongs to the trusted application currently running. Aegis [243] uses a trusted security kernel and includes the virtual page management in this trusted kernel which is enough to protect from software accesses in this weakened attacker model. Intel SGX [183], PodArch [236], TrustLite [151], XOM [170] and Sanctum [78] do not trust the OS nor the hypervisor so the MMU/MPU needs to be modified to verify whether an access is authorized or not. This verification may be implemented in hardware (for PodArch, TrustLite and XOM), in microcode instruction (for Intel SGX) or by a trusted piece of software (the security monitor in Sanctum).

Note that disallowing the untrusted OS to read pages of the trusted application would prevent it from doing standard management tasks such as paging out these memory pages. This issue is either ignored (TrustLite uses a MPU which does not provide page management, XOM disables paging and Sanctum lets each trusted application manage its own page table) or the untrusted OS is given the possibility to access an encrypted version of the page in order to page it out (Intel SGX, Overshadow, PodArch, InkTag, Bastion).

### Measuring the protected region

In these schemes, the untrusted OS is in charge of loading the trusted application into memory so that it can execute. It is therefore mandatory to either prevent the OS from modifying the application before it is loaded or to give the application the ability to prove its integrity to an external verifier. Aegis, Bastion, Intel SGX, Sanctum and TrustLite offer primitives (either implemented in software or hardware) to issue a measurement of the current trusted application (signed with a private key bound to the device). On the other hand, Overshadow, PodArch and InkTag accept encrypted applications and the trusted hardware of VMM checks the integrity of the data once it is loaded. In these cases, the application provider can embed a secret into the application which will guarantee the integrity of the loaded data. XOM also accepts encrypted applications but does not verify the integrity of the data when it is loaded in memory.

The main advantage of providing trusted applications encrypted is that it natively provides a way to securely transfer secret data (either resources or code) to the trusted application. On the other hand, trusted applications distributed in plaintext need to contact the external verifier and establish a secure channel through which the trusted application measurement may be sent and secrets may be provisioned.

### Controlling entry and exit

To guarantee isolated execution, it is important to enforce strict policies about how control flows both from the untrusted environment to the trusted application and from the trusted application to the untrusted environment. Controlling entries in the trusted application is important in order to prevent simple control-flow hijacking by jumping in the middle of the trusted application's code. Controlling exits from trusted application is necessary to prevent private information from leaking to the untrusted environment through the processor's registers. That is why all of the mentioned TEEs include in their TCB (hardware, trusted hypervisor or trusted kernel) a mechanism to control where the trusted application can be entered and to save the context of the trusted application when it is interrupted. The notable exception is XOM which stops the trusted application when an interrupt occurs during its execution.

### Protecting against physical attacks on the processor-memory bus

We have already mentioned these protections in a previous section. More generally, protecting the processor-memory bus against physical probing is orthogonal to software isolation provided by TEEs: without memory access policies enforced on software modules, encrypting the transactions in the memory controller does not prevent a software attacker from accessing the memory of the trusted application (since the data in memory is fetched as it would be by the trusted application). On the other hand, legal memory accesses from the trusted application would leak to a physical attacker if memory accesses are not encrypted. Likewise, protecting the application against addressing pattern leakage could be addressed separately. As a matter of fact, none of the TEEs described in this section natively implements a kind of ORAM protection.

### Protecting against side-channel attacks

As all these schemes attempt to provide software isolation at the cost of minimal hardware modifications, protecting against side-channel attacks is often discarded as outside of the attacker model or because protections are orthogonal (like designing an encryption algorithm in a way that it is not vulnerable to cache timing attacks). There are two classes of side-channel attacks that are of particular interest in the context of TEEs: cache timing attacks and addressing pattern leakage through page faults (called controlled-channel attack [264]). The threats they pose are serious as they are efficient and do not require physical access to the device. The notable exception is Sanctum, which provides protection against both of these attacks by partitioning the cache so that the trusted application never competes with another application for cache lines and let the trusted application manage its page tables so that it does not need the untrusted OS to load back a page on page faults.

### ARM TrustZone

The TrustZone technology [10] from ARM introduces a partitioning of the processor into two modes (which are orthogonal to the traditional user/kernel mode): normal world and secure world. Transitioning from one world to the other is controlled by a secure monitor running in the secure world. When issuing transactions on the AMBA AXI system bus, the processor forwards the world the processor is running in to other hardware modules by using a special bit added to the AMBA AXI bus. This bit enables to partition other resources (like DRAM) and to give different views depending on the world the processor is running in. Typically, the secure world has access to the memory of the normal world but not the other way around. TrustZone's guarantees depend on how these hardware modules handle the secure bit. In particular, it does not necessarily provide memory encryption, software attestation or side-channel protection.

Most of the industrial schemes branded as TEEs based on TrustZone [1] run a secure OS in the secure world which is used to isolate software in the normal world. They differ from the other solutions presented here (sometimes called hardware-enforced isolated execution environment) as the hardware support (TrustZone) provides isolation for only one trusted application which is used to manage the applications running in the normal world.

Table 2.1 – Comparison of various TEEs.

| TEE | SW Attest. | Controlled jump | Trusted OS | Trusted VMM | Protection mem. bus | Protection addr. pattern | Protection cache timing |
|---|---|---|---|---|---|---|---|
| XOM | ✗ | ✓ but int. kill app. | ✗ | N/A | ✗ | N/A (Paging not supported) | ✗ |
| Aegis | ✓ | ✓ | ✓ | N/A | ✗ | ✗ | ✗ |
| Bastion | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Intel SGX | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ |
| Sanctum | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Overshadow | Encrypted App | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| PodArch | Encrypted App | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| InkTag | Encrypted App | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| TrustLite | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| ARM-based | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |

#### 2.1.4.2 Trusted path

Isolated execution environments as they were presented may be seen as architectural solutions that link a state of the processor (depending on the program counter, the privilege level) to a restricted view of the memory. These two hardware components—the processor and the external memory—are the fundamental elements that compose a traditional computer system in an overly simplistic presentation. However, there are many applications where other hardware components (that we will call *peripherals* to simplify) are used and need to exchange valuable information with the trusted application. Three potential use cases would be:

- The peripheral is an input device used to authenticate the user to the trusted application.
- The peripheral is an output device (monitor, speakers) that is used to (dis)play protected content.
- The peripheral is used as an accelerator to offload the processor or increase the throughput of an algorithm transforming private data.

In these cases, the trusted application needs to extend the range of the protection offered by the TEE to the peripheral. This issue of establishing a secure channel between a peripheral and a trusted application in an untrusted environment has been intensively studied under the name of *trusted path*.

[1] Kinibi from Trustonic, QSEE from Qualcomm, T6 from TrustKernel, SecuriTEE from Hansol Secure, CoreTEE from Sequitur Labs, ProvenCore from Prove&Run, OP-TEE from Linaro, SierraTEE from Sierraware

Conceptually, one could argue that offering different views of memory depending on the execution context of the processor is a solution to a sub-case of the problem of establishing trusted paths to peripherals. The main difference as far as physical attacks are concerned is that, in the case of memory, the peripheral does not need to know the actual value of the data to provide the service it is supposed to. Other peripherals share this peculiarity: other storage media like hard disks or hardware accelerators that work on data encrypted with homomorphic functions for instance. For peripherals that handle cleartext data (like an input device), transactions with this peripheral need to be encrypted to protect the confidentiality of the data against physical probes on the communicating bus (LPC, PCI, PCIe, SPI, SATA, USB, etc.).

This consideration leads to a classification of studies on trusted path into two groups: the ones that use cryptographic primitives to protect the path from end to end and the ones that weaken the attacker model in order to limit modifications of the peripheral (hardware or firmware). Note that in both cases, an ideal trusted path solution would guarantee:

- confidentiality of the data (e.g., to prevent keyloggers),
- authentication of the application to the peripheral (e.g., to prevent phishing attacks [101]),
- authentication of the peripheral to the application (e.g., to prevent malware from submitting transactions to the trusted application) and
- integrity of the data to prevent an attacker from altering a genuine transaction.

#### Supporting legacy devices

In a scenario where the peripheral does not provide cryptographic primitives to build an end-to-end trusted path protection, physical attacks on the peripheral bus need to be put out of scope. However even with this weakened attacker model, providing a trusted input path or a trusted display remains a challenge, especially with limited TCB. As the peripheral device does not enable to establish an encrypted or authenticated channel, the communication to the device needs to be isolated from the untrusted software stack. Communication with this device would normally occur through direct access, interrupts or DMA. All of these are typically under control of the OS or hypervisor which can, if it is compromised, intercept the interrupts generated by the peripheral, initiate DMA to this device or configure another PCIe device so that its address range overlaps with the one of the trusted peripheral.

As such, a trusted anchor is needed to isolate the trusted path. It may be the OS [102, 248], a trusted hypervisor [257, 267, 274, 275] or a hardware feature providing software isolation (AMD SVM, Intel TXT, ARM TrustZone) [103,168,206]. [103,206] are based on Flicker [182], a solution which uses AMD SVM or Intel TXT to launch a measured, isolated environment which has exclusive access to the peripherals and can be attested by an external entity.

#### End-to-end protection

The downside of these mechanisms, apart from the weakened attacker model, is that they either require to stop all other concurrently executing software or need to considerably increase the TCB. To provide end-to-end protection, a cryptographic channel between the peripheral and the trusted application—or an external communicant—is needed. In an untrusted environment with no TEE in place, the application would not be able to keep cryptographic material secret from other privileged and untrusted software. Thus, solutions that offer end-to-end protection either:

- trust the software stack (e.g., a set-top box encrypting a video stream to send it through a HDMI link protected with High-bandwidth Digital Content Protection),

- use an isolated execution environment [19] or
- distrust the application and establish an encrypted channel between an external trusted entity and the peripheral [256].

Note that if the OS or hypervisor is trusted—like in some of the aforementioned schemes—cryptography could still be added as an orthogonal protection against physical probes on the peripheral bus.

### 2.1.5 Conclusion

In this section, we have explored why the distinction between hardware and software may make sense from a security point of view. Naturally, there exist practical, methodological differences between hardware and software concerning their development (implementation languages are generally different), deployment (hardware needs to be synthesized), verification (as will be presented in the next section) and security concerns (physical damage or logical exploits). However, this section has shown a more theoretical approach to this difference: we sometimes need to think about systems in their hardware/software duplicity to understand their behaviour. As far as security is concerned, this behaviour may be

- unintentional and unwanted in case of hardware-based attacks; or
- intentional when the hardware implementation exports high-level features that provide primitives on which secure software solutions can be built.

In this thesis we were especially focusing on the second case. More generally, we were interested in how security can be analyzed for systems described as hardware and software components. In the next section we will discuss how software or hardware behaviours can be described—modeled—and verified and how these models can be linked to prove high level properties on whole systems.

## 2.2 Hardware and software models

Modeling is fundamentally linked to the process of realisation (be it the realisation of a process, of a physical object or of a language). Models are—as etymology would have it—measures to which an implementation must compare and eventually conform to. Models first and foremost exist as an intermediate step between the intellectual conception of an object and its realization. All of the large-scale projects go through a modeling stage whose purpose is to provide the designer with a reduced—but hopefully similar—version of the object that is being realized. Its purpose is to enable the designer to predict how the final object will behave under different environments so that she can change her conception if the object does not meet some required criteria. In the field of computer systems design, these criteria may be about functionalities, safety, performance, security, reliability, availability, ergonomics, etc.

Formal verification designates the activity of evaluating these criteria (expressed as *properties*) on the model using mathematical tools. As mathematical tools require the model to take a certain form that is not necessarily suitable for evaluation by human eyes, we split this presentation into two parts: first we will present how models are used during the design of embedded systems with a particular attention to security; then, we survey different verification methods and mathematical models commonly used and study how they differ with respect to their relation to the software abstraction.

### 2.2.1 Designing secure embedded systems

As mentioned earlier, models are intermediate steps between specification and implementation in *Model-Driven Engineering* (MDE). We will first present the fundamental concepts of MDE as described in [82].

#### 2.2.1.1 Model-Driven Engineering

Modeling languages used in MDE can be classified according to their domain and their viewpoint. The domain of a modeling language corresponds to the application domain of the systems that can be described with this modeling language. General-purpose modeling languages (such as UML or SysML) can describe a wide variety of systems while domain-specific languages define artifacts specific to the application domain and thus allow easier modeling and model transformation in their specific domain.

For each system being designed, multiple criteria can be considered in a model. The viewpoint of a modeling language defines which kind of aspect of a system is meant to be described in this modeling language. First, a modeling language can specifically target the description of static or dynamic properties of the system. A static description shows how a system is structured while a dynamic description shows how it behaves. Also, it is possible to classify modeling languages according to their abstraction level. This classification was proposed by the Object Management Group (OMG) in their approach to MDE (called Model Driven Architecture) [189]. According to this classification, models can be:

- Computation Independent Models (CIM), which *"focus on the environment of the system, and the requirements for the system"*.
- Platform Independent Models (PIM), which detail the operation of a system but do not depend on the specific platform that will be used to implement this system.
- Platform Specific Models (PSM), which show how a specific platform will be used by the system.

These viewpoints enable to unambiguously detail a specific *view*—behaviour, graphics or communications for instance—of the system under conception.

The specificity of embedded system design lies in the concurrent conception of software components and of a hardware architecture—or even hardware components. This raises new issues compared to software engineering in other domains: which components will be present in the embedded system? How do they communicate? How are behavioural tasks mapped to these components? Different methodologies have been explored to help embedded system designers answer these questions.

#### 2.2.1.2 Embedded system design methodologies

While agile methodologies have been widely accepted in the traditional software development industry for their ability to cope with specification changes and unpredicted development issues [215], they have more trouble permeating the hardware and embedded system design market—despite some interest [119, 258]. According to these two presentations, the difficulties to adapt agile methodologies to hardware development—and thus, to some extent, to embedded system design—come from three fundamental differences:

- Building a hardware prototype architecture usually takes a lot more time than compiling software. In particular, procuring parts requires much more time than fetching software libraries.

- Hardware components are expensive. This makes trial and error costly for companies, in particular when hardware custom components need to be synthesized.
- Hardware and embedded system designs involve diverse competencies (software design, hardware design, formal verification, etc.) that are often implemented in different teams so communication and synchronization is difficult.

These reasons explain why the *waterfall*—or *V-Model*—paradigm is still widely used in the embedded system development industry. In such a paradigm, system design follows different successive steps: requirement, analysis, design and implementation.

During the design step, embedded system design methodologies typically propose to model the system according to two different questions. The first one, more abstract, requires the designer to specify a functional representation (PIM) and an architectural representation (PSM) of the design and to map functional elements onto architectural nodes. This approach [145], named Y-Chart has been widely adopted [14,20,146,253] in order to help designers choose which function to map on software or hardware components (this process is called *design space exploration*). At this step, the designer is typically able to get feedback about the latencies, bus usage or hardware module solicitation induced by the mapping she has realized. The second question the designer needs to address is how the global system composed of various components behaves in terms of computations and in terms of communications.

The models created during requirement, analysis and design phases can conform to different languages which describe the possible diagrams available to the designer and how they should be used. Examples of such languages are SysML (a UML profile) or Arcadia [216]. Modeling languages can be implemented in various dedicated or general purpose modeling tools (e.g., Capella, Sirius, Papyrus).

During all these steps, formal methods can be employed to verify safety and security properties on the design. Some methodologies provide a semi-formal description of their modeling language in order to formally verify properties directly on analysis or design models [204].

#### 2.2.1.3 Security in design methodologies

As embedded systems become more connected, their security becomes a growing concern and many—even critical—embedded systems have been shown to present various vulnerabilities [13, 74, 79, 136]. To address these issues, some works have attempted to integrate security concerns into model-based engineering methodologies.

Security may be integrated early in the design process through security requirements and security analysis. Security requirements are often associated to an analysis of the possible threats targeting the system which are rigorously evaluated [8, 199] and can be expressed textually (informally), mathematically or by specific models such as attack trees [47].

Different models have been presented to capture system security features in analysis or design models. They usually rely on an existing modeling language or technique to enable capturing functional and non-functional properties on the same model. They can target specific models (such as Petri nets [8]) or be integrated in a broader design methodology such as UML [141, 174, 234] or SysML [204, 217]. These solutions also differ in terms of specificity of the security properties they target. Indeed, some may provide *low-level* modeling capabilities that enable to model a large range of security features or properties (such as Linear Temporal Logic properties in [8]) while some require to model security properties in a more specific semantics (confidentiality and integrity of messages in [141, 204, 217, 234] or access control in [174]).

These security modeling capabilities may or may not provide integration with formal verification tools to guarantee properties on the system (as it is described in the model). We will now

survey how formal verification can help designers in assessing safety and security properties on their design.

### 2.2.2 System verification

Formal system verification has benefited from constant interest since the dawn of computers. In its simplest form, formal verification may target satisfiability of first-order logic formulas (an NP-complete problem known as *SAT*). Various algorithms (*SAT-solvers*) have been proposed to automatically answer the SAT problem. These verification algorithms take their roots in the Davis-Putnam–Logemann–Loveland (DPLL) algorithm [89]. This algorithm takes as input a logic formula in Conjunctive Normal Form (CNF) and recursively finds a valuation of the literals that would satisfy the formula or fails if the formula is unsatisfiable. Each time a literal is picked and a value is affected to it, the formula is simplified and recursively checked to see if it is satisfiable. If it is, a valuation has been found: the picked literal is added to the valuation found satisfying the sub-formula and the valuation is returned. If the sub-formula is not satisfiable, the other value is affected to the literal and the formula is simplified and checked again. The algorithm was further extended [107, 191] to interface with domain-specific solvers (such as real numbers, bit vectors or linear inequalities). The DPLL algorithm (or another SAT-solving algorithm) queries the domain-specific solver to check if a clause if satisfiable. If it is not, the domain-specific solver returns a core conflict clause that the SAT-solver adds to the original formula. The SAT problem with background theories is known as Satisfiability Modulo Theories (SMT).

It would theoretically be possible to verify any property on any system by finding an adequate logic formula and solving it with an SMT-solver. However, even for small real-world systems, such a logic formula may be far too large to be solved by an SMT-solver. In order to achieve formal system verification, additional semantics needs to be provided to decompose the problem. To this regard, some verification methods require that a human interact with the prover and some use an algorithm on more abstract models—compared to first-order logic formulas—to provide totally automatic verification. Both of these methods may use SMT-solvers (such as Z3 [91]) as building blocs.

In this work, we are interested in how formal verification techniques can provide a precise evaluation of security to designers in the specific context of systems consisting of hardware and software components. We will first present verification methods that expect human interaction and argue that they require advanced training and significant manual work. Those are thus not well suited to our problem. Then, we will discuss automated verification methods and show how some are more suitable to software or hardware verification.

#### 2.2.2.1 Semi-automated verification

In this first section, we propose to describe verification methods that do not completely infer a property automatically but instead rely on a human interaction to help it. Such methods define a language used to model the system and the property to verify and a set of allowed inference rules. The application enables the user to derive properties using inference rules. Moreover, the application can automatically apply simple inference rules when possible. As the human user and the application appear as interacting entities where the former require that some inference rule be applied and the latter checks that the required rule is valid, these solutions are grouped under the name of *proof assistants* (or equivalently *interactive theorem prover*). Some of the most well-known proof assistants are Isabelle/HOL [197], Coq [31], EasyCrypt [25] and the B-Method [3].

Proving non-obvious properties on whole systems by using proof assistants may involve a lot of human work (11 person years were needed to write the 200,000 lines of Isabelle script to verify the

14,000 lines of Haskell and C code of the seL4 microkernel [148]). Yet, automated verification algorithms quickly suffer from the fact that increasing the system size usually exponentially impacts the time needed by the algorithm to complete (traditionally called the *state explosion* issue). In contrast, while human verifiers are slower to apply and combine inference rules, their intuition to select which rule to apply is hard to implement in automated verification algorithm and helps in directing the proof to counter state explosion.

This partly explains the choice of many academic and industrial solutions that use proof assistants to formally verify their system. To name but a few: seL4 [148] is a microkernel of the L4 family which has been formally proved to guarantee high-level functional properties on its C implementation. An Haskell prototype of the kernel was first implemented and automatically translated into Isabelle/HOL code. The resulting code (named executable specification) is proved to be a refinement of a more abstract specification in Isabelle. Concurrently, a C implementation is generated based on the Haskell prototype. The authors of [148] propose a formalization of part of the C language in Isabelle/HOL language and show that the implementation of seL4 is a refinement of the executable specification. It results that properties proved on the abstract specification (expressed as Hoare logic) are verified on the C implementation (more precisely, on the model described by the C implementation and the formalization of the C language). Another project involving formal verification through interactive theorem proving is CompCert. CompCert [163] is a C compiler that handles a large subset of the C language and was initially targeting the PowerPC architecture. The compiler is developed in the Coq language and the proof is also described in Coq. The proof targets observational equivalence between the source and compiled code: the authors of this tool prove that the compiler does not introduce a behaviour that could not be observed in the source code and it does not introduce cases where the compiled code could *go wrong* (such as accessing an array out of bounds). The implementation of the compiler takes around 6,000 lines of Coq code and the proof 36,000. Another famous application of semi-automatic verification in an industrial project is the B-Method which was used to design the pilot of the automated Metro Line 14 in Paris and the pilot of the automated shuttle in Paris Charles de Gaulle airport [18].

Formal verification of these designs has involved a lot of manual work. As mentioned earlier, state explosion prevents automated verification of large systems. However, performance increase of these algorithms may enable them to be applied to relatively small systems. Apart from the amount of work it would spare, there is another important advantage to fast automated verification of systems. Indeed, skills required to manually verify a system are usually complex and quite different from implementation skills. As a result, a distinct team often needs to be dedicated to formal verification which complicates the feedback that it would provide to the designer if it were applied on implementation models.

#### 2.2.2.2 Automated verification

To allow designers with little formal verification knowledge to prove properties on models, totally automated verification methods can be used. These automated formal verification methods may differ in terms of:

- the modeling language used to describe the system,
- the language used to describe the property to be verified,
- the environment under which the system is expected to run, and
- the algorithm used to evaluate the property.

### Representing simple systems as finite state machines

A—more abstract than boolean formulas but still simple—representation of systems is to depict them as Finite State Machines (FSMs): a finite set of states and a set of transitions linking these states together. Various other state-based models (such as Petri nets or Kripke structures) are based on FSMs and extend them with specific semantics. Systems may be modeled directly by FSMs or they may use another modeling language which is pre-processed to be transformed to FSMs. This language can for instance be a synchronous language such as Esterel [30], Signal [28] or Lustre [124]. FSMs are interesting to model systems as they capture an *evolution* of the system and so enable to verify temporal properties on the model. These properties are typically expressed as temporal logic formulas (e.g., using Linear Temporal Logic or Computation Tree Logic).

Tools that check such properties on finite state machines are grouped under the generic name of *model checkers* [72]. Even though finite state machines have a finite number of states and transitions, this number may be quite large, even for simple systems due to the state explosion problem. Various strategies have been proposed to improve the verification algorithm by arbitrary limiting the states that are checked by the algorithm (*bounded model checking* [33]) or by simplifying the model before the algorithm runs through abstracting part of the state of the system (*symbolic model checking* [184]). This abstraction could generate a mismatch between the abstract and original model. This would result in a property being verified in one model and not in another. To solve this issue, abstraction is usually iteratively refined through a process known as CounterExample Guided Abstraction Refinement (CEGAR) presented in [70]. In CEGAR, the abstraction is chosen so that it is a *sound* approximation: if the property holds for the abstract model, then it also holds for the original one. If the property is proved false on the abstract model, a counterexample on the abstract model is generated and checked against the original model. If it is valid for the original model, then the property is false on the original model. If it is not, the counterexample is *spurious* (introduced by the abstraction step) and abstraction is refined so that the counterexample is not valid on the abstract model anymore. These improvements enable to verify properties on models of real-life systems [51].

### Protocol Verification

Modeling concurrently executing actors which take part in a protocol (or game) with FSMs is difficult as the number of states is greatly increased due to concurrency. Other modeling languages were thus proposed to model these systems. Many of them derive from the family of *process calculus*. Process calculus and its derivations enable to model a system by giving a high-level formal description of the actors and communications of the model. It focuses on how processes are duplicated, how they communicate and how they synchronize rather than on a low-level imperative description of their implementation.

Formal verification of cryptographic protocols has motivated the presentation of numerous and various methods since the 1990s. Blanchet gives in [37] a survey of these methods and of tools that implement the methods. We will summarize this survey and we refer the interested reader to this paper for a more complete overview.

A first distinction which allows to classify protocol verification approaches lies in how cryptographic primitives are modeled. Some methods assume a *computational model* in which the concrete representation of data (as bitstring) is taken into account so that the protocol can be analyzed in terms of probability to violate the security properties. This model is relatively close to the eventual implementation of the cryptographic protocol and has been traditionally assumed by cryptographers to analyze the security of cryptographic schemes. Automated formal verification of computational models is however difficult. A representation more suitable for automated

verification is achieved by using *symbolic models*. In a symbolic model, cryptographic primitives are assumed to be perfect: hash functions can not collide, encryption can only be reversed by decrypting with the corresponding key, etc. Moreover, all messages exchanged in the protocol, crafter by a genuine participant or by the attacker, can only use these well-defined primitives.

While automated verification is easier for symbolic models than for computational models, the problem is undecidable in the general case [98]. In particular, the set of messages that the attacker is able to learn in the protocol is infinite since the attacker can apply primitives an infinite number of times and since an infinite number of sessions of the protocol can be executed in parallel. To cope with this problem, diverse solutions have been proposed:

- The number of sessions and the length of messages can be limited. In such case, traditional model-checking techniques can be applied. This approach is taken by FDR [176] or SATMC [17] for instance. It is efficient at finding attacks against a protocol but can not guarantee that a protocol is free of vulnerability since part of the state space has not been explored.

- Another possibility is to limit only the number of sessions and assume reasonable properties on the cryptographic properties in which case the problem may be NP-complete. The protocol can then be analyzed with constraint solving or model checking based approaches for instance (as in the Cl-AtSe [65] or OFMC [27] tools).

- A third possibility is to simply allow the verification tool to not terminate in some cases as it is done in Maude-NPA [186].

- It is also possible to restrict the class of protocols on which the verifier will work (tagged protocols in [211] for instance).

- A manual intervention of the user may be requested to direct the proof. This intervention may for example be an interaction with a proof assistant (Isabelle/HOL in [203]), the input of type annotations for Cryptyc [116] or the specification of intermediate lemmas for Tamarin [220].

- Finally, the verification tool can rely on an abstractions to terminate at the price of completeness: the tool may provide an "I don't know" answer. This method was pioneered by Bolignano in [41]. A tool that falls in this category and that we have used during this thesis is ProVerif [35]. We will discuss in Chapter 4 why we chose this tool over the other mentioned here.

Note that some platforms have been implemented that provide a protocol description language and allow this language to be translated to other languages to use the aforementioned tools. This is the case of the Common Authentication Protocol Specification Language (CAPSL) [188] which can be translated to the input language of Maude-NPA or of the High-Level Protocol Specification Language (HLPSL) used by AVISPA [16] which can use multiple back-ends like SATMC, Cl-AtSe or OFMC. It is also notable for this thesis that some of these tools have been integrated to design modeling methodologies using modeling languages such as UML or SysML [204, 234].

Verifying protocols in the computational model can be achieved in two ways. The first possibility relies on a result Abadi and Rogaway [2] which states that under some assumptions, if a protocol is secure in the symbolic model, then it is also secure in the computational model. This enables to exploit the existing automated verification tools for symbolic models. The second possibility is to use one of the fewer dedicated tools (mostly CryptoVerif [36] and EasyCrypt [26]).

#### 2.2.2.3 Hardware and software verification

Formal verification tools usually expect systems that need to be verified to be modeled in a dedicated language. As mentioned in the previous section, some translation algorithms were proposed to convert design models into models suitable for formal verification. However, this translation may be tedious to automate and error-prone if the original modeling language and the mathematical language used for formal verification are too different. This explains that some formal verification methods are more suited to verify software or hardware even though a transformation of the model can often close the gap. We will discuss these methods in this section. Note that we are interested in solutions bringing mathematical guarantees and thus do not discuss solutions that execute the targeted system within a specific environment (including input): testing and dynamic analysis.

**Hardware verification and software verification**

Model checking methods on FSMs—as detailed earlier—are of particular importance for hardware designs. While they are not specific to hardware models, the state explosion problem heavily limits the size of models that can be checked through—even symbolic—model checking. Yet, model checking is perfectly suitable for small parts of hardware designs and has been implemented in various commercial and academic tools (Incisive from Cadence, Magellan from Synopsys, VIS [45], NuSMV [69], UPPAAL [162], CADP [108], etc.).

Combinational equivalence is another property that is particularly important for hardware designers. It is used to prove that two boolean functions are equivalent (which is a coNP-complete problem). There are traditionally two ways of proving such a property: either by using SAT-solving algorithms as DPLL mentioned previously [113] or by using a canonical representation of boolean functions (as reduced ordered binary decision diagrams). Recent research in this domains focuses on increasing performance of combinational equivalence checking algorithms for specific circuits [12, 190, 192] or extending equivalence checking to non-combinational circuits (sequential equivalence checking) [71, 219].

Software-specific verification methods have typically been conceived with two properties in mind that differ from other verification methods (apart from their expected input modeling language). First, they usually need to verify large systems since most of the software vulnerabilities do not come from the core controlling logic but from implementation details. Secondly when software verification is applied to a system described in an implementation language, it may need to handle parts of the system in different forms: machine code for various architectures, source code in various languages. These constraints shape the various automated verification methods that will be presented in this section. This presentation is built upon various recent surveys [53, 95, 223].

As for hardware models, model checking has been implemented in many tools targeting software descriptions (SPIN [130], JAVA PathFinder [126], SLAM [21], BLAST [32], etc.). As mentioned earlier, model checking suffers from the state explosion problem and most of the research effort concerning software model checking has focused on improving performance of model checking algorithms by approximating them. Approximation is typically done either through abstraction in symbolic model checking and predicate abstraction or through limiting the depth of the search in bounded model checking. Note that limiting the depth of the search would normally affect the completeness of the proof but simple static analysis of software (as estimating loop counts) may help in approximating the completeness threshold through trial and errors (the depth of a model that needs to be explored so that further exploration would only go through already explored states).

An automated static analysis method close to symbolic model checking that has been raising

great expectations from the community due to the increasing performance of SAT and SMT solvers is symbolic execution [53]. Symbolic model checking is based on constructing an FSM of the system according to abstractions chosen before the FSM construction. Symbolic execution delays abstraction until a branch is found. In symbolic execution, state variables have symbolic values and the current explored path is conditioned to a constraint formula. When reaching a branch, a constraint solver (typically SMT solver) is used to check that the two branches can be taken. If they can, symbolic execution splits and the constraints are added to the respective path constraints. Various tools exist to symbolically execute software: KLEE [52], S2E [66], Symbolic PathFinder [208], etc.). As model checking, symbolic execution suffers from a kind of state explosion (called *path explosion* for this method), so recent research works have focused on improving its performance by selectively merging states [161] or approximating the exploration with concrete values [225] when the size of the path constraint exceeds the capabilities of the constraint solver or when the source code for part of the system is not available (e.g., libraries).

Another static analysis technique widely applied to analyze software programs is abstract interpretation [80]. Instead of computing a FSM of the program as concrete and symbolic model checking do, abstract interpretation iteratively builds an approximation of the program by using monotonic functions to over-approximate the set of values the state variables may have at any point in the program. Operations on state variables are extended to operate on the abstract domains used to represent the variables and the state variables abstract representation is iteratively enlarged to fit new values computed by these operations until a fixed point is reached. The approximation needs to be carefully chosen to be sound—as defined earlier—and to balance the precision of the proof with the time needed to reach the fixed point. Tools such as Astrée [81] or IKOS [44] enable to verify programs using abstract interpretation. The inabilities of abstract interpretation to generate counterexamples and to deal with subtle, path-sensitive properties explains that it is mainly used for verifying simple generic properties or for compiler optimizations [95]. Other formal verification methods exist that cope with the state space explosion problem by trading accuracy, like flow sensitiveness: pointer analysis [127] for instance.

Another area of research concerning formal verification of software concerns the prover environment:

- providing a—preferably general-purpose—modeling language to describe software programs in order to present a common and practical input language to formal verifiers [48,52,97,122] or
- formally describing the effect of software instructions on real hardware [106].

### Combined HW/SW verification

Most commonly, industrial verification of hardware and software co-designs relies on simulation and testing rather than on formal verification due to the state explosion problem limiting the size of designs that can be formally verified. Due to potentially tight interaction between hardware and software, new verification methods focus on simulating both hardware and software at the same time instead of successively verifying hardware against a hardware specification and software against a software specification. In order to test a HW/SW co-design, the hardware components need to be either implemented in a prototype [194], simulated in software [131, 224, 242] or hardware and software need to be described in an abstract modeling language [51] (this case was discussed in the previous section).

An important step in co-design testing is to define how software and hardware components interact (e.g., software procedures triggered by hardware interrupts, software variables mapped to

hardware signals). The question of how software and hardware components interact is central to all formal verification methods targeting hardware/software co-designs [6,40,71,157,193,245,254, 262,263,273]. Indeed, most of the verification methods proposed rely on leveraging a common formal language to model both hardware and software components. This formal description may be directly formulas for SAT or SMT solvers [71,193], (extended) petri nets [245], Kripke structures [157] or other automata [164,273]. The formal model that is verified may also be derived from another common specification such as C [6] or SystemC [157].

The main difficulties that these works address are the composition of hardware and software components and the performance of the verification algorithm. As previously mentioned, composing hardware and software components requires to define the mechanisms used to interact (the interface) but also the possible concurrency of hardware and software components to model that software and hardware execution is not sequential (the problem is for instance raised in [131]). It is notable that, in order to simplify the combination of hardware and software models, most of the verification techniques target rather loosely coupled hardware and software components. Some solutions as presented in [133] propose to verify a full emulation of the hardware but this greatly limits the size of the design that can be verified.

### 2.2.3 Conclusion

In this section, we have presented various formal verification methods targeting generic or specific system models. Testing was not part of this presentation as it does not provide a mathematical guarantee that a property is satisfied on the system (except if the model is exhaustively tested, which relates more to model checking than to testing). Nonetheless, testing is by far the preferred software and hardware quality attestation technique used by the industry in most—if not all—of the large system design project. Testing is faster and requires less specific competencies than formal verification. The process of implementing then testing is also more intuitive for developers than starting by formally modeling the design to apply mathematical verification. Moreover, the guarantees that testing may bring can be greatly improved by automated test-case generation techniques that take metrics such as coverage into account, only leaving unproved details. Yet, the devil is in the detail.

Table 2.2 – Comparison of various verification methods.

| Verification methods | Manual work for | | Design size | Suitable for verifying | | | |
|---|---|---|---|---|---|---|---|
| | modeling | verifying | | HL models | HW | SW | HW/SW |
| Proof assistants | Low | Very high | Very large | ✓ | ✓ | ✓ | ✓ |
| Model checking | Medium | Low | Medium | ✓ | ✓ | ✓ | Loosely coupled |
| SAT/SMT solving | High | Low | Small | ✓ | ✓ | ✗ | Loosely coupled |
| Abstract interpretation | Low | Low | Large | ✗ | ✗ | ✓ | ✗ |
| Symbolic execution | Low | Low | Medium | ✗ | ✗ | ✓ | ✗ |

## 2.3 Conclusion

As far as this thesis is concerned, analyzing the security of embedded systems draws the attention on three different points:

- Many studies have proposed isolated execution environments. Trusted path has also been a subject of research. Yet, the combination of these techniques has not been extensively studied and the existing approaches [257] considerably weaken the attacker model proposed by trusted execution environments.
- Security analysis during embedded system development has recently been an important subject of research. These works have focused both on providing modeling languages to perform security analysis and design early in the design and on formally verifying security properties on system models. However, the combination of both in a design methodology that would enable easy modeling of security features, fast formal verification algorithms accessible to designing teams and clear feedback to the designers is still an unsolved problem.
- Formal verification of hardware/software co-designs has primarily focused on loosely interacting software and hardware components. In the case where hardware modifies the semantics of software, formal verification of relatively large systems is still an open problem.

We will propose answers to these three questions in the next sections. In chapter 3, we show how Intel SGX can be extended to provide hardware support to establish a trusted path between a peripheral and an application. Verifying some parts of this system (such as the key exchange protocol) can be done from high-level design diagrams. We thus show in chapter 4 how it is possible to perform verification from SysML diagrams during the partitioning and software design phases. However, modeling a system with loosely interacting hardware and software may not be suitable (or even doable) for some systems where hardware modifications deeply impact how software executes. We study this problem in chapter 5.

# Chapter 3

# Hardware and software from a security point of view

Hardware may be an essential part of a software security solution where it is sometimes used to improve performance (e.g., to enforce control flow integrity) or to anchor trust in a secure physical core (e.g., a TPM used to measure part of the memory). In order to better understand how software and hardware components can cooperatively provide security guarantees, we propose in this chapter to present a design—based on an existing architecture—that aims at providing a secure path between a peripheral and an application. The proposed architecture protects the communications with an external peripheral against a powerful attacker that controls the whole software stack (except the application) and is able to physically probe the bus between the processor and the peripheral.

We chose this scenario as a case study to illustrate the presentation in Chapter 4 as it relies on a recent technology to enforce strong security guarantees that could not be achieved in software only. Formally analyzing the interacting hardware and software components to prove security properties of the whole design is a valuable but difficult problem.

## 3.1 Intel SGX architecture

While the first infrastructure providers go a long way back, the advantages of virtualization in terms of machine management (reliability, availability, dynamic resource allocation, machine migrations, etc.) still makes it an active research field. This interest stresses out the need to realize the abstraction that has always motivated virtualization: behave as if the customer application were alone in an isolated, high-availability and high-performance physical machine. Economic value of the data managed by such a customer application may—and actually frequently does—motivate other co-located applications to abuse this belief of clear isolation.

There are two different ways an application can spy on another co-located Virtual Machine (VM): either by performing actions or measurements that are—legitimately or not—available to it, or by elevating its privileges to attack the targeted VM from a more privileged context. Both have proved to be efficient in breaking the isolation abstraction and recovering confidential information [214]. The discovery of a vulnerability in a hypervisor is unfortunately not so uncommon [205] and it motivated academic and industrial research targeting VM isolation from the privileged hypervisor. In particular, technologies have been presented to build an untampered isolated software *enclave* upon a corrupted environment. One of the latest industrial technologies

providing such software isolation capabilities is Intel Software Guard eXtension (SGX).

Intel SGX relies on limited hardware modifications and a cooperation between hardware and software components to provide a Trusted Execution Environment. In this first section, we propose to give a brief overview of Intel SGX: its attacker model, its security guarantees, its mechanisms and its architecture. For a more thorough presentation of SGX, we refer the reader to the detailed paper from Costan et al. [77]. We will then argue that extending Intel SGX protection to trusted peripherals is possible at the cost of some minor hardware modifications. This design is a typical example of a hardware/software co-design that could greatly benefit from formal security verification.

### 3.1.1 Overview of the protection model

Intel SGX provides strong security guarantees in terms of software isolation and software attestation in presence of a powerful adversary.

#### 3.1.1.1 Attacker Model

The main idea behind SGX is to reduce the Trusted Computing Base (TCB) to the processor package only. All hardware components outside of the package can be seen as elements providing services to the software running in the enclave protected by the processor. While the enclave—which runs in user mode—relies on the OS to perform actions such as accessing peripherals, it should not believe the OS to actually be honest. Likewise, the hypervisor should not be trusted. In the rest of this chapter, we use the term OS to denote the privileged software stack, including the hypervisor.

**Physical attacks**

An enclave would be very limited in what it could do without trusting something as basic as data coming from DRAM. So including—at least parts of—the DRAM into the TCB would first appear to be mandatory. Once Direct Memory Access (DMA) to the protected DRAM area are forbidden, and assuming that an attacker cannot fake a DRAM or spy on the communication inside the processor die (which contains the processor cores and the uncore: the QuickPath Interconnect controller, the DDR3L/DDR4 controller, the shared L3 cache mainly) and on the memory bus, the only entity able to read from and write to DRAM would be the processor attached to this DRAM (see Section 3.1.2 for a discussion about multi-processor architectures).

While one can argue that spying on the communications inside the processor die requires relatively costly equipment, connecting a fake DRAM to the memory bus could be an affordable attack for realistic attackers. Since the Memory Controller (MC) in recent Intel architectures is integrated to the processor die, the TCB can be limited to the processor die by transparently encrypting and integrity checking memory reads and writes to DRAM in the memory controller. This is implemented by SGX in the Memory Encryption Engine.

**Software attacks**

In Intel SGX, the whole software stack is untrusted: the OS, hypervisor and SMM are considered as potentially corrupted. The only trusted software applications are the application enclave and a few other management enclaves. All of them are measured to guarantee their integrity.

#### Side-channel attacks

Side-channel attacks are not considered by Intel SGX. In particular, cache-timing attacks have been proved to be effective in retrieving secret data handled by an enclave [117]. While hardware-based protections exist to protect against side-channel attacks [78], some software-only countermeasures (as presented in section 2.1.1.1) can be used to protect against these attacks and the problem can be considered orthogonal.

### 3.1.1.2 Guarantees

Intel SGX provides strong program isolation: memory pages of protected enclaves are only readable and writable by their owner. Control flow from and to the enclave is protected so that an enclave can only be entered on defined locations and such that the processor registers are cleared before control is passed to the untrusted software when the enclave is interrupted. Moreover, the enclave can be measured to provide a cryptographic proof of the content of the memory pages of the enclave and of its configuration (such as entry points).

## 3.1.2 Architecture

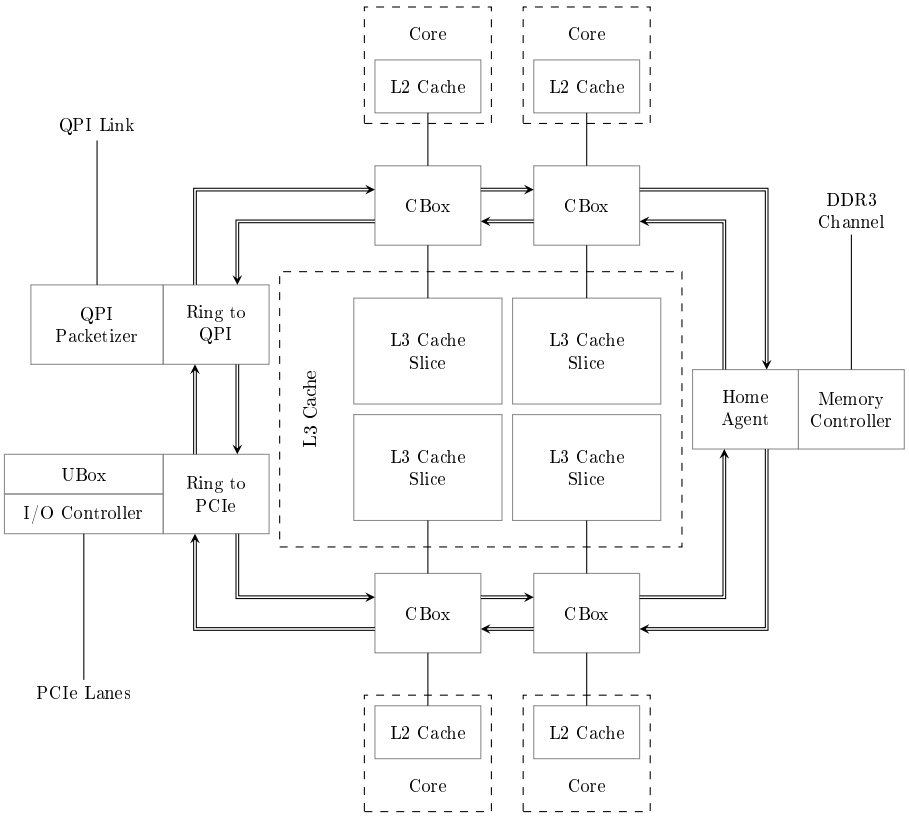



Figure 3.1 – Intel inter-core and core-uncore communication overview (borrowed from [77]).

Figure 3.1 presents an overview of the architecture of a Skylake processor die. This shows the hardware TCB considered by Intel SGX. Data fetched from memory is decrypted by the memory

controller and stored unencrypted in the L3 caches. All links outside of the processor die (QPI, DDR) are untrusted and it is assumed that an attacker can eavesdrop and tamper with these buses.

From a logical point of view, part of the memory is reserved for the processor to store enclave pages and metadata to manage the enclaves. Accesses to this part of the memory from a non-enclave application are forbidden. This Processor Reserved Memory (PRM) is used to store:

- SGX specific pages in an area called the Enclave Page Cache (EPC).
- Metadata about these pages (one entry per page in the EPC). This area is called the Enclave Page Cache Map (EPCM). The address of an EPCM entry is computed by the processor based on the physical address of the corresponding EPC page.

An EPC page can be either:

- a regular page of an enclave
- an SGX Enclave Control Structure (SECS) page which contains metadata about an enclave or
- other types of management pages that we will ignore for the clarity of the presentation (used to store hashes of swapped out pages for instance).

**Security of multi-processor, SGX-enabled architectures**

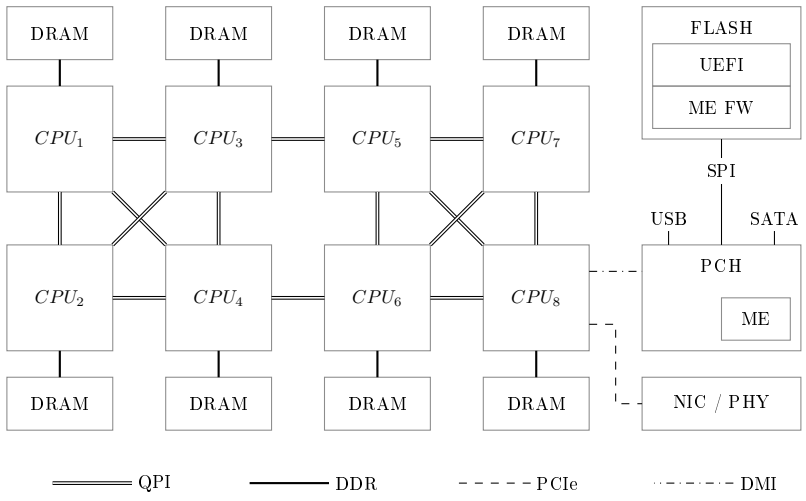



Figure 3.2 – Intel motherboard overview (borrowed from [77]).

As mentioned previously, the HW TCB in SGX design is reduced to the processor die. This means that the communication links between processors in a multi-processors architecture are considered to be untrusted. In each processor, multiple cores are connected to the ring (see Figure 3.1), which is also connected to a memory controller and a Quick Path Interconnect packetizer that enables the processor to communicate with other processors. When a core on a processor needs to access protected memory attached to another processor, data is decrypted on the second processor and sent back to the first through the QPI link. For instance, if $CPU_1$ in

figure 3.2 accesses the memory of an enclave with an address that belongs to the DRAM attached to $CPU_2$, the data stored at this address is first decrypted by the memory controller of $CPU_2$, transferred through the ring interconnect (see Figure 3.1) to the QPI packetizer of $CPU_2$ where it is encrypted. Then it is sent encrypted to the QPI packetizer of $CPU_1$ which decrypts it and sends it to one of the CBox on $CPU_1$ which will store the unencrypted value in cache.

Considering SGX TCB, securing these QPI links is thus important, and yet only briefly mentionned in an Intel patent [140]: *"The CMA* [Intel denomination for MEE] *fully integrates into the Intel QuickPath Interconnect (QPI) protocol, and scales to multi-package platforms, with security extensions to the QPI protocol. In a multi-package platform configuration, the CMA protects memory transfers between Intel CPUs using a link-level security (Link-Sec) engine in the externally facing QPI link layers."*. However, this *Link-Sec* engine does not seem to be mentionned anywhere else.

As these QPI links are interfaces between the core trusted by Intel SGX and the outside world which is considered to be controlled by an attacker, the implementation of the cryptographic protections of these links is critical to the overall security of the architecture. Moreover, the implementation needs to be efficient to deal with the high throughput of this high-performance link. Reversing Intel cryptographic protection of the QPI links would be interesting, but difficult. Indeed, these links are high-speed (initial implementation was 3.2GHz, which yields 6.4GT/s as QPI is double-rate) and each transfer carries 16 bits of data.

### 3.1.3 Enclaves

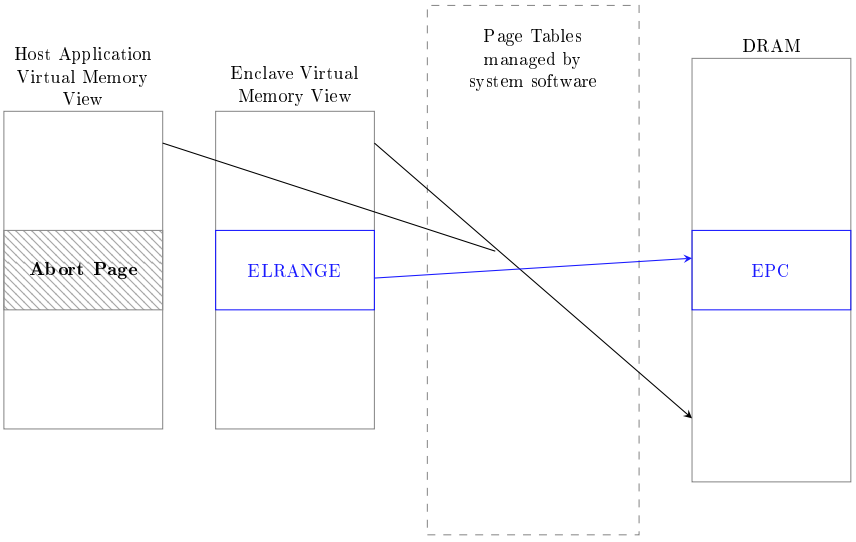



Figure 3.3 – Enclave and host application memory view (adapted from [77]).

As illustrated by Figure 3.3, an enclave is a subpart of an application—a collection of memory pages—that needs to be protected. All the memory pages of an application, no matter whether they are in an enclave or not, share the same virtual address space. However, when a program running from outside an enclave tries to access the enclave page, the memory access is denied and the processor returns zeroed out data.

To implement this access control policy, the Memory Management Unit (MMU) has been

modified to check each time a new entry is added to the Translation Look-aside Buffer (TLB) that the targeted enclave page belongs to the same enclave that is currently running. TLBs are flushed at each context switch to guarantee that the enclave pages cannot be accessed in the new context.

The management of memory pages (page table management, swapping) is done by the system software. Since system software is untrusted, address translations are checked by the processor on TLB misses.

### 3.1.4 Memory protections

As presented in section 2.1.2.1, to prevent a physical attacker from spying on the processor-memory bus, the content of the PRM is encrypted by the memory controller and a Merkle tree is maintained to guarantee integrity of the memory.

## 3.2 Integrating security-aware peripherals

In an SGX setup, it is interesting to wonder what kind of operation could be performed by an enclave without weakening the attacker model. The software TCB is well defined as everything outside the enclave code is not trusted. When the enclave needs to communicate with other software, it thus has to assume that the other software may be corrupted. This other software can for instance be the OS. Indeed, enclaves execute with user privileges and the OS is not trusted. As a result, an enclave should not assume system calls to be correct or it would be vulnerable to a corrupted OS returning malicious values from system calls (this type of attacks is known as Iago attacks [60]). The hardware TCB is reduced to the processor die. As for software communications, the enclave should thus not trust data coming from outside the processor die: from a peripheral for instance.

In this section, we introduce the problem of peripheral communication on an untrusted machine. We present possible scenarios with and without Intel SGX technology. Some of them are commonly found in commercial setup or research projects and some are theoretical architectures that we propose in order to combine the guarantees provided by different security solutions. We then discuss their security guarantees and their performance.

### 3.2.1 The problem of peripheral communication on an untrusted machine

Communication with a peripheral introduces a new mean for the OS to spy on data handled by the enclave. More generally even if the OS is not malicious, an application could manage to modify the configuration space of the PCIe endpoint of the host, or one of the PCIe switches along the path from the host to the peripheral (targeting the Base Address Register for instance) in order to route Transaction Layer Packets to an attacker-controlled peripheral. This is even more realistic in modern setups where a lot of devices may be plugged on the PCIe bus—in order to provide concurrent access to different NIC for different VMs for instance. While the reality of large chunks of data transiting on the bus between the processor and a peripheral may have been arguable years ago, the increasing amount of data handled by algorithms in the cloud nowadays calls for more specific devices to unload the CPU. This trend is best illustrated by the ongoing adoption of FPGAs in modern cloud architectures.[1]

[1] `https://azure.microsoft.com/en-us/resources/videos/build-2017-inside-the-microsoft-fpga-based-configurable-cloud/`

To the potential threat of a malicious OS and of a misconfigured PCIe network should be added the less obvious possibility of the infrastructure provider tapping the PCIe link to spy on the communication between the enclave and the peripheral. This means that, even if the OS (or at least the PCIe configuration space) were trusted, this would not prevent a motivated physical attacker from spying on the communication. For all these reasons, it may be interesting to think that an attacker can control the communication on the PCIe link.

**Example scenario**

In the following scenarios, we consider that an application is running on a particular core in user mode and that it is relying on an underlying OS to communicate with the peripheral connected through the PCIe link. Some example cases of such a setup could be:

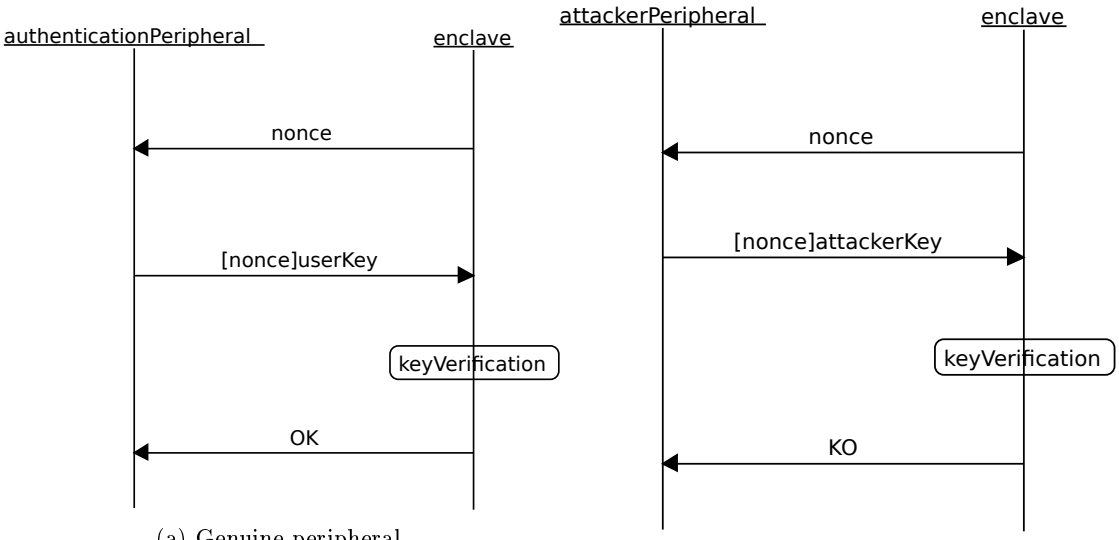
- The application needs the user to input some private data.
- The application needs to output private data to the user.
- The application needs to read values coming from sensors, in a way that the owner of the platform cannot fake them.
- The application wishes to rely on an accelerator to run some algorithm on private data.
- The application needs to rely on a peripheral to run some algorithm on private data that it is not able to (because keys are stored in a HSM for instance).

The first case has received a lot of attention from the community since it corresponds to the case where a user wishes to connect to an application on a platform that she owns, with some guarantees that the inputed credential could not be stolen by an eventual all-powerful malware that would have control over the whole SW stack except the application (to which the user is trying to authenticate). This attacker model is however far less powerful than the one used to design SGX since the attacker has no HW access (we could argue that it only assumes that the attacker cannot replace the peripheral by another one, but it is somehow difficult to depict a scenario where the attacker would be able to spy on the internal buses, and yet could not replace a peripheral).

In most of the exposed cases, an attacker should not be able to fake being the expected peripheral. This is obvious when the application must send private data to the peripheral, and when the application expects data from one particular sensor. In the first case however, some scenario may not require the peripheral to be authenticated. We will present two short examples to illustrate why it may or may not be useful:

- The application unlocks personal data when the user is identified by plugging some personal device that sends a key to the application. In this case, the ownership of the device is the property that needs to be verified. An attacker would gain nothing by using a different peripheral, except if she manages to get the key beforehand. This scenario is illustrated by Figure 3.4.
- The application unlocks personal data when the user is authenticated by using a secure fingerprint sensor. In this case, the authentication of the sensor is mandatory to guarantee the security of the scheme since, otherwise, an attacker could replace the rightful sensor with a fake device that will intentionally leak the key of the user. This scenario is illustrated by Figure 3.5.

To simplify, we have used the following notations in Figures 3.4 and 3.5:




(a) Genuine peripheral.

(b) Attacker peripheral.

Figure 3.4 – Authentication based on ownership of the peripheral.

- $[m]actor$ means the signature of $m$ with the private key of $actor$.
- $[*]actor$ means the signature of everything preceding this in the current message with the private key of $actor$.

We will reuse this notation for other sequence diagrams presented later.

### 3.2.2 Architectural description of the setup

We give in this section an abstract description of a typical setup of a cloud provider machine hosting a customer application that uses data stored in RAM and communicates with a peripheral connected through the PCIe bus.

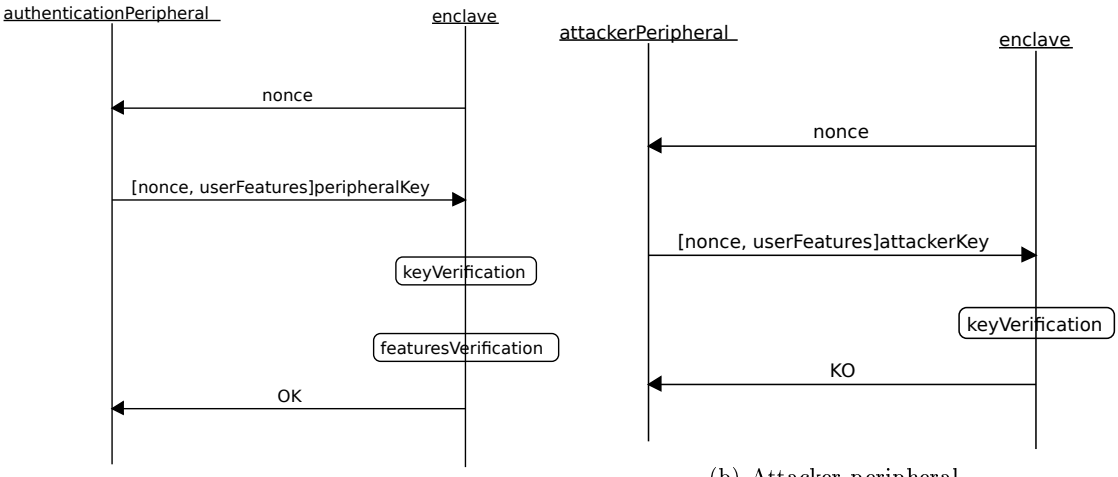
The host machine represented in Figure 3.6 is inspired by latest Intel architectures (Skylake) but could mainly represent most of the recent non uniform memory architectures used in today cloud servers (except for technical subtleties that are not relevant here). It consists of interconnected processors, each composed of multiple cores and a Last-Level Cache (LLC) and connected to a memory module, the southbridge (called Platform Controller Hub in Intel architectures) and high-speed peripherals through PCIe (or AGP).

### 3.2.3 Threat Model

We classify the different attackers that will be considered in this section according to their abilities.

#### Bus Manipulation

The memory bus may be encrypted and the memory controller may or may not be integrated to the processor die. As a result, the attacker may or may not be able to spy on the memory bus.




(a) Genuine peripheral.

(b) Attacker peripheral.

Figure 3.5 – Authentication based on user features.

When the attacker can spy on the bus, we will consider that it is also able to replay a previous message and send its own messages.

An attacker may or may not be able to spy on the PCIe link, either by putting hardware probes on it, or by abusing PCIe configuration. When the attacker can spy on the link, we will consider that it is also able to replay a previous message and send its own messages.

#### Accessing Application Memory from another Software Application

A software attacker can either be another application running in user mode on the same or on another core, or it can be the OS itself. We are interested in its ability to access the memory of the application.

#### Side-Channel Attacks

As we said earlier, Intel SGX considers that side-channels are an orthogonal problem. We will thus also ignore them in our work.

### 3.2.4 Security Properties of Interest

Ultimately, we want to evaluate if the peripheral and the application can rightfully trust their communication to be secure. This security is evaluated according to the following criteria:

- Identification of the Other Party: the application identifies the peripheral or the other way around.
- Confidentiality of the Data Exchanged.
- Integrity of the Data Exchanged.
- Non-Repudiation Property.

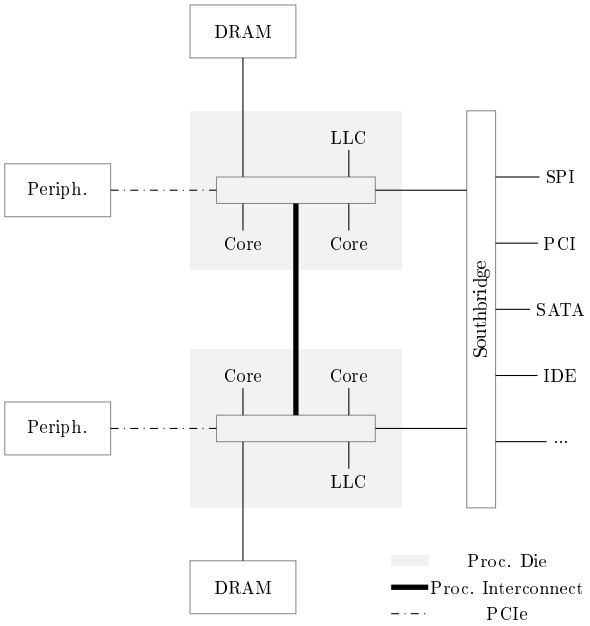



Figure 3.6 – Hardware Description.

### 3.2.5 Various protection scenarios

We give in this section an iterative process toward a secure cloud infrastructure. Each step improves security or performance by leveraging technologies (mostly cryptographic-based) that may be available in the industry or may require more or less substantial modifications of existing ones.

We focus here on a situation where a peripheral is connected through a PCIe link. Some of the results given should be valid for a peripheral connected through the southbridge (with some technical modifications) but performance is evaluated for a PCIe device. Note that some low-frequency buses connected on the southbridge are shared (and not point-to-point as PCIe) and spying on those buses would be much easier than spying on the PCIe or memory bus (this is the case for SPI, PCI, I2C, etc.). Also for simplicity, we will consider that the host only consists of one processor (with potentially multiple cores).

#### 3.2.5.1 No protection

The first scenario, where no particular attention is given to security, will help in understanding potential threats and in comparing performance with other designs.

**Presentation**

In this scenario, an application hosted on the CPU exchanges data with a PCIe device. At some point, this data is stored in the DRAM of the host and travels along the PCIe channel. No special precaution is taken as to where this data is stored and how it is sent.

Obviously, the OS is able to read and modify this data in memory. If the machine is a desktop or server computer, we can safely assume that the OS could configure an IOMMU to forbid DMA

transfers originating from non-authorized devices connected through PCIe. However, an attacker able to spy on the PCIe link would be able to recover data exchanged with the trusted peripheral.

#### Security Discussion

Although this scenario may seem too naive with respect to the other ones presented in the paper, it is nonetheless important to present as it is the model that best represent a typical cloud infrastructure as of today.

#### 3.2.5.2 Memory encryption

A first attack vector that can be addressed is the processor-memory bus.

#### Presentation

As presented in section 2.1.2.1, various solutions [49, 58, 78, 90, 96, 99, 120, 170, 243] propose to protect the content of the memory by encryption and hash trees.

#### Security Discussion

These solutions guarantee the confidentiality and integrity of the data stored in memory against a physical attacker who can spy on the processor-memory bus. However, they do not protect against a logical attacker which controls either the OS or an application on the processor.

#### Performance

Some of these solutions can be found in commercial off-the-shelf processors. In particular, Intel SGX uses an extension of the memory controller called Memory Encryption Engine (MEE) to dynamically encrypt and hash data read and written to memory. In [120], Intel advertises performance measurements on the MEE: *"We see that the MEE imposes performance degradation that varies from 2.2% to 14%, with an average of 5.5%."*

#### 3.2.5.3 Enclaves and memory isolation

Protecting the processor-memory bus prevents *external* attackers from tampering with data managed by an application. However, it does prevent another application running on the same core from accessing the data of another application. The next step toward a secure communication between an application and a peripheral is thus to isolate the memory of an application from other software.

#### Presentation

Software isolation has also been previously discussed in section 2.1.4. Many architectures have been proposed to protect software enclaves against software-based attacks from other—potentially privileged—applications [58, 63, 78, 128, 151, 170, 183, 236, 243].

#### Security Discussion

These solutions provide isolation guarantees against an attacker which controls the OS or hypervisor. Their protection is however purely logical in most cases (although some like Intel SGX are combined with memory-bus protections described in the previous section).

#### 3.2.5.4 Driver-enforced trusted path

While these protections enable an application to securely handle data in an isolated environment, the application cannot trust communication with an external peripheral. The problem of establishing a trusted path isolated from other software applications thus needs to be addressed.

**Presentation**

We described solutions that aim at providing trusted path in section 2.1.4.2. They also aim at protecting against logical attackers but instead of protecting data in memory, they target the communication with a peripheral.

**Security Discussion**

To provide such a protection, most of these solutions rely on a trusted logical core (driver, microkernel, hypervisor) to establish a secure link between the application and the peripheral. Their attacker model is thus weaker than Intel SGX as they require support from a trusted piece of software and do not protect against physical attacker spying on the channel to the peripheral.

**Performance**

As these solutions do not assume that there is a strict memory isolation they usually cannot rely on DMA to transfer data to the peripheral. While DMA may not be suitable for all types of peripheral, their use may greatly improve performance in case large chunks of memory need to be transferred. This would typically be the case in a setup where the peripheral is a GPU or hardware accelerator.

#### 3.2.5.5 Software-based channel encryption

As traditional trusted path requires to rely on a trusted piece of software to communicate with a peripheral, it is not directly applicable to an Intel SGX setup. Indeed, Intel SGX only enables to protect user-mode applications which cannot directly access peripherals without trusting the OS (the page tables are managed by the OS).

**Presentation**

An enclave protected with SGX can nonetheless securely transfer data to a peripheral by using cryptographic primitives. The encrypted data could then be transmitted directly to the peripheral (through port-mapped I/O or memory-mapped I/O) or stored in memory outside of the PRM and a peripheral could access it through DMA.

**Security Discussion**

In this scenario, communication with the peripheral is protected both from a logical attacker with control over the OS and against a physical attacker who could spy on the PCIe link.

**Performance**

Although the Skylake architecture provides special instructions (AES-NI) that implement AES encryption in hardware, this method would require either to directly transfer data to the peripheral (through memory-mapped I/O for instance) which would use CPU cycles and incur a non negligible performance cost if large memory chunks need to be transferred; or to write them

back to memory to trigger a DMA transfer. Arguably, this method would results in poorer performance that the preceding one. Indeed, for each data unit to be transferred, one memory read and one memory write would be issued by the CPU, and one memory read would eventually be issued by the peripheral. If the encrypted cache line needs to be written to memory, this would result in two memory accesses for single-use, disposable data. This is even more problematic since the memory bus is nowadays the bottleneck for most architectures.

#### 3.2.5.6 Unified memory and I/O encryption

**Presentation**

It could also be possible to leverage the fact that the data is already present in encrypted form in memory to avoid decryption and re-encryption. We could thus modify the MEE to let the enclave decide which key should be used to encrypt some particular memory page. Another device knowing this key could thus access the memory through a DMA transfer. Confidentiality would thus be guaranteed by the fact that only the enclave and the device know the key that was used to encrypt the requested page.

The main downfalls of this method are:

- When the MC receives a transaction directed toward an address in the DRAM, it does not know which enclave it belongs to, and thus which key should be used. Knowing such thing would involve either a lot more logic on the MC side to fetch the EPCM corresponding to the requested page and fetch the required key, or the active participation of the CPU that would provide with each request an identifier of the enclave, or directly the key to be used.
- An enclave may wish to allow part of its memory to be shared, but not all of it. MC would thus need to match keys to memory ranges instead of enclaves, which would require more memory reserved for the controller to remember these policies.
- Data in cleartext in the LLC could not be used directly to be transferred to a peripheral. This would require that the cache data is written back (and encrypted) to the memory and read again to be transferred to the peripheral.
- Even with such an implementation, replay attacks would still be possible. To prevent these, we would need to add sequence numbers (or similar mechanism) to mark transactions between the memory controller and the peripheral. However, this would require a second layer of encryption / hashing.

**Performance**

This last comment is important since it negates to some extent the effect of reducing the number of encryptions to perform data transfer to the peripheral. However, moving the cryptographic algorithms from the CPU to an external entity (be it the MC or another one) would still be beneficial to unload the CPU.

#### 3.2.5.7 Hardware-based channel encryption

To this regard, we could also find a compromise between HW modification of the MC and the number of encryptions performed by placing the encryption logic in an exterior entity (e.g., I/O controller). The CPU would configure this entity with the correct key and enable DMA transfers to the enclave memory range. No HW modification on the MC core logic would be needed.

This architecture is described in more details in the next section.

## 3.3 A Secure and efficient design

In this section, we propose architectural modifications to the Intel architecture that try to balance the three following points:

- **Hardware modification** is kept as lightweight as possible. This is achieved by leveraging already existing functionalities implemented as part of Intel VT-d technology. (See [77] Section 2.11.3)
- **Security** guarantees granted by Intel SGX are extended to the communication with a peripheral without substantially increasing the TCB.
- **Performance** is taken into account which rules out decrypting and re-encrypting directly in the CPU.

### 3.3.1 Example scenario

Using cryptographic algorithms to secure a path between a peripheral and an enclave however comes with a price, in terms of execution time and delay particularly. This price should be weighted against the guarantees that are brought by the encryption and against the purpose of the peripheral (e.g., when it is a HW accelerator). We first limit our study to cases where the price is negligible. This means that we focus on cases where data is exchanged through burst instead of frequent interrupts. This rules out controllers and sensors.

An interesting example case is when an enclave must perform heavy computations and wishes to unload the CPU by running them on a HW accelerator (such as on a GPU). This HW accelerator could be shared by all the enclaves / VMs on the host platform (a form of isolation could be guaranteed, for instance by self-virtualizing the device as in [238]). In this case, the enclave needs some guarantees that it is sending private data to the expected peripheral, and not to a host-controlled device (either because it is impersonating a rightful accelerator, or because the host OS changed the MMIO configurations to redirect traffic to another device). Since the enclave does not know, a priori, which specific accelerator will be present, there should be a trusted third-party the enclave would depend on to authenticate the peripheral. Such third-party could for instance be present as a root certificate issued by the vendor of the HW accelerator.

### 3.3.2 Architecture of the design

A representation of the proposed architecture is depicted in Figure 3.7. Gray blocks represent the physical modifications that would have to be performed on the original architecture of Intel SGX.

This architecture would require some modifications in two different components: the CPU and the DMA remapping unit.

#### 3.3.2.1 CPU modifications

**Adding key and counter to the Enclave Page Cache Map**

In order to authorize DMA to a protected page, the Enclave Page Cache Map (EPCM) entry corresponding to this page should define a key and a counter that will be used to encrypt and decrypt transactions targeting this page. Intel System Programming Guide [137] does not describe in details the fields of each EPCM entry but adding a new field should not be really difficult (in the worst case, EPCM entry size would need to be increased). If we use the same

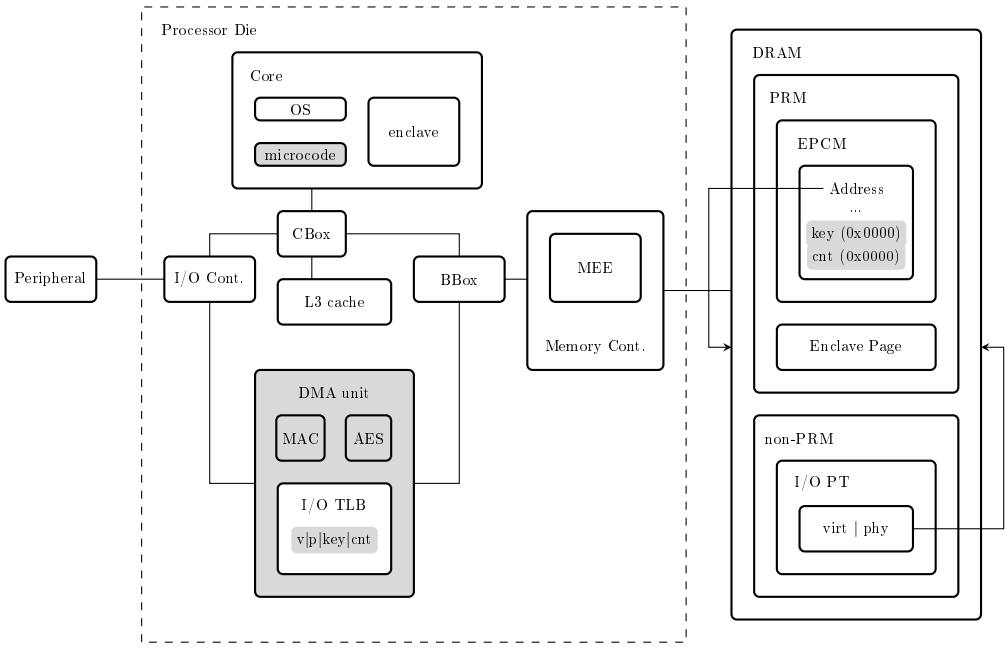



Figure 3.7 – Proposed Architecture.

cryptographic algorithms than the ones used in MEE, we would need 56 bits for the counter and 128 bits for the key, which adds up to a total of 184 bits (23 bytes). When an enclave page is created (with EADD instruction), the key field should be set to zero to forbid DMA to this page until the enclave explicitly sets it. The key field would also need to be reset when the page is swapped out. This would prevent the OS from swapping the physical addresses of two protected pages and replay a transaction (that embeds the virtual address of the targeted page) on a different physical page. The enclave should thus pick a new key each time the page is swapped out.

It would also be possible to flush I/O TLB each time an enclave page is swapped out as it is done for standard TLB.

#### Adding a new instruction

The main logic that would need to be added on the processor side is an instruction to set the key (and reset the counter). This means that setting the same key multiple times on the same page would open up possibilities of replay attacks. This could be mitigated by letting the instruction randomly pick a key. However, it would thus be impossible to share a unique key between different pages so a lot of cryptographic material would need to be exchanged with the peripheral, which probably rules out this possibility. Still, due to the swapping problem, enclaves should not assume that they will only need one key.

#### 3.3.2.2 DMA remapping unit

DMA domains are described in Intel VT-d spec [76] as *"an isolated environment in the platform, to which a subset of the host physical memory is allocated"*. This abstract definition can be

implemented in different ways but Intel *"envisions DMA-remapping hardware to be implemented in Root-Complex components, such as the memory controller hub (MCH) or I/O hub (IOH)"*. I/O page tables stored in system memory are used to translate DMA-Virtual Addresses to Host-Physical Addresses.

Encrypting and decrypting memory transactions to enclaves would be implemented by the DMA remapping hardware unit. DMA remapping hardware unit receiving a transaction targeting PRM would first use unmodified IO page tables to translate the DMA-virtual address to a host-physical address. If this host-physical address falls into PRM, EPCM would be queried to know if the page is accessible through DMA. If so, key and sequence number would be used by the remapping unit to encrypt / decrypt data. Note that this key and sequence number could also be cached in the IOTLB.

A DMA read by a trusted device would result in a transaction containing the encrypted cache line—we suppose that the granularity of memory accesses are of one cache line, as the MEE—, the realization of a counter and a MAC over the counter and virtual address of the page.

Depending on how the remapping units are implemented, they could either reuse existing cryptographic hardware (present in the MEE for instance) or use their own. Note that a more in-depth investigation should be performed to see if reusing MEE encryption hardware would not negatively affect performance of memory accesses from CPU cores. Since on-die silicium is scarce, properly selecting minimal-sized hardware modules is necessary. Since the required cryptographic primitives are the same as those needed by the MEE, hardware for incrementing the counter, computing the MAC and performing AES could be the same as the one described in [120, 121].

#### AES module

An AES module would need to be added to the DMA remapping unit to encrypt and decrypt transactions targeting PRM range. The same module as the one used by the MEE (and described in [121]) could be used. According to this paper, the unified encrypt/decrypt module computes an AES round in one clock cycle and only occupies an area of $7995\mu m^2$ (round keys are precomputed). In our design where we need to compute round keys each time (since keys are different depending on the page accessed), the silicium area occupied should be around $10000\mu m^2$.

#### MAC module

We also need a MAC module to guarantee integrity of the transactions. The same MAC module as MEE (Carter-Wegman MAC) could be used.

#### I/O TLB

I/O TLB lines would need to be increased to contain the 184 bits of the key and the counter.

#### Additional logic

The main logic implemented by the DMA remapping unit would be to fetch key and counter when a DMA transfer targets a protected page and to discard the memory access when the key is not set.

Incrementing the counter could also use the hardware module described for MEE which relies on a "56-bit Galois Shift Register with taps in positions 34, 35, 55".

For write transactions, the DMA remapping unit would have to check that the MAC is valid before forwarding write to the memory controller.

#### 3.3.2.3 Key management

Figure 3.8 shows a sequence diagram of the protocol used to share a secret key between a peripheral and the enclave (called *driver* here). This protocol relies on an external entity to verify that the peripheral is genuine (we call it *verifier*).

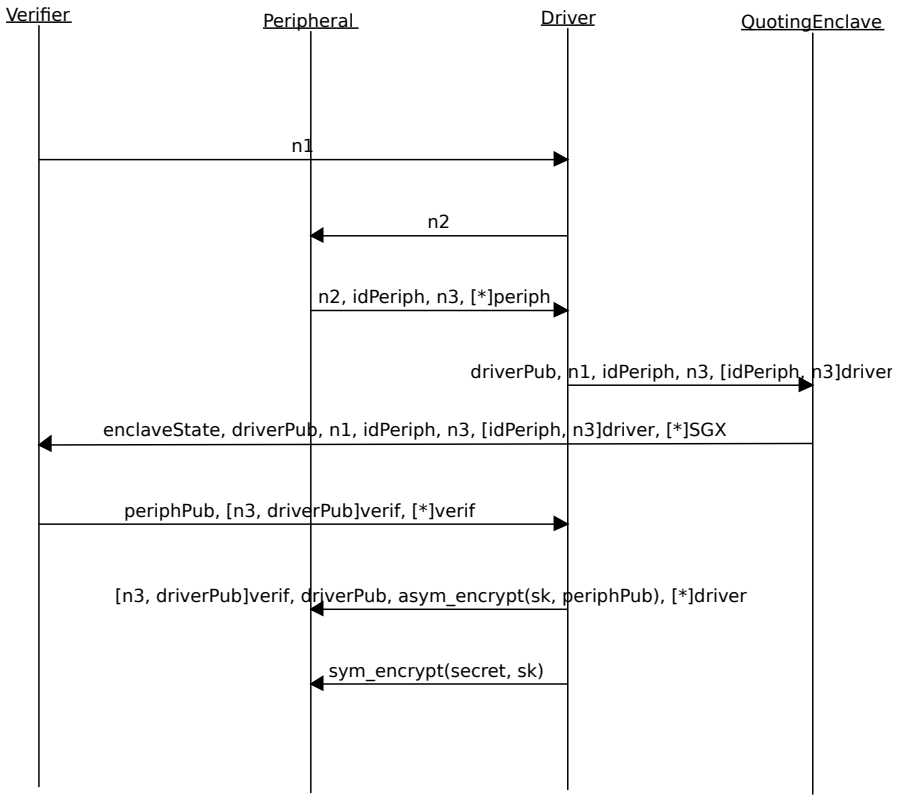



Figure 3.8 – Key establishment.

To simplify, we modeled the SGX attestation process as a simple actor named *QuotingEnclave* which outputs a signed measurement of the state of the enclave. This measurement also contains the data passed to it by the measured enclave. This summarizes a more complex attestation protocol described in [77] for instance.

The steps of this protocol are:

1. The verifier sends a nonce $n_1$ to the driver.

2. The driver sends a nonce $n_2$ to the peripheral.

3. The peripheral returns $n_2$, its *id* and a new nonce $n_3$ to the driver. This message is also signed by the peripheral with its private key.

4. The driver creates a new public and private key pair.

5. The driver requires a quote that embeds the measurement of the enclave, the public key of the driver, $n_1$, the $id$ of the peripheral, $n_3$ and a signature of the driver over the $id$ of the peripheral and $n_3$.

6. The resulting quote is forwarded to the external verifier.

7. The verifier verifies that the quote originates from an authorized enclave and contains $n_1$.

8. The verifier fetches the public key of the peripheral according to its $id$.

9. The verifier sends a signed message to the driver containing the public key of the peripheral and a signature of the verifier over the public key of the driver and $n_3$.

10. The driver verifies the signature of the driver.

11. The driver verifies the signature of the peripheral on the message received in step 3.

12. The driver creates a new symmetric key $sk$.

13. The driver sends to the peripheral a signed message composed of

    - the signature of the verifier over $n_3$ and the public key of the driver,
    - the public key of the driver and
    - $sk$ encrypted with the public key of the peripheral.

14. The peripheral verifies the signature of the driver.

15. The peripheral verifies the signature of the verifier over $n_3$ and the public key of the driver.

16. The peripheral decrypts $sk$.

At the end of the protocol the driver and peripheral share the symmetric key $sk$. Note that:

- The driver does not contain any private material initially. This is needed since the OS is untrusted but it has to load the driver into memory.

- The driver does not know which peripheral it is going to interact with. The authentication is done by the external third party.

- The peripheral does not know which driver it is going to interact with. It relies on the external third party to verify that the correct driver is running in a genuine SGX-enabled system.

- The verifier can selectively forbid communication depending on the $id$ of the peripheral or on the content of the driver.

The security of this protocol is formally verified and the results are presented in section 4.3.2.

#### 3.3.2.4 Steps to allow DMA

In order for an enclave to authorize DMA to a subset of its memory and for the peripheral to access it, the following steps would happen:

1. The enclave issues an instruction to set the key for a particular enclave page. This instruction checks that the enclave page belongs to the enclave calling this instruction and sets the key and resets the counter in the EPCM.
2. The OS adds the mapping for the targeted enclave page to the I/O page table.
3. The peripheral tries to access the enclave page (addressed with the virtual address) through DMA. The request is handled by the DMA remapping unit that fetches the corresponding entry in the I/O page table.
4. The DMA remapping unit fetches the EPCM entry corresponding to the targeted page. Since the address of this entry can be computed from the physical address of the page, only one memory access is needed. If the key is set, the DMA remapping unit adds the mapping, key and counter to its I/O TLB and forwards the memory access to the memory controller.
5. When the DMA remapping unit gets the result from memory, it stores the content in the L3 cache (part of the L3 cache of Intel architectures is reserved for DMA)
6. The DMA remapping unit concatenates data and counter, encrypts it and appends the MAC of the counter and virtual address to it. The result is sent to the peripheral. The counter is then incremented.
7. If the peripheral reads data from the same page again, the key and counter can be used directly from the I/O TLB so only one access to memory is needed. If this particular data is already cached, the CBox queries it from cache instead of forwarding it to memory so no memory access needs to take place.

This protocol is illustrated by the sequence diagram presented in Figure 3.9.

Only steps 5 and 6 would change if the peripheral were to issue a DMA write. these steps would be:

5. The data received from the peripheral is composed of two parts: the encrypted data and counter and a MAC over the counter and virtual address. The DMA remapping unit thus decrypts the first part and checks that the counter is correct with respect with the value fetched from memory. Then the MAC is verified against the counter and the required virtual address. The counter is then incremented.
6. The data is then sent to the CBox so that it can be written either to cache or to memory depending on the cache policy implemented.

### 3.3.3 Performance evaluation

One of the advantages of this architecture is that it is transparent for anything behind the I/O controller. As such, if data is present in cache, it would be possible to fetch it directly instead of going through the memory controller to query the DRAM. Indeed, since Xeon E5 family, Intel introduced Data Direct I/O Technology [75] which consists of directing DMA transfers to the last level cache instead of main memory. This must be evaluated in different scenario since in some scenario (such as GPU communication), memory to transfer may be tagged as non-cachable.

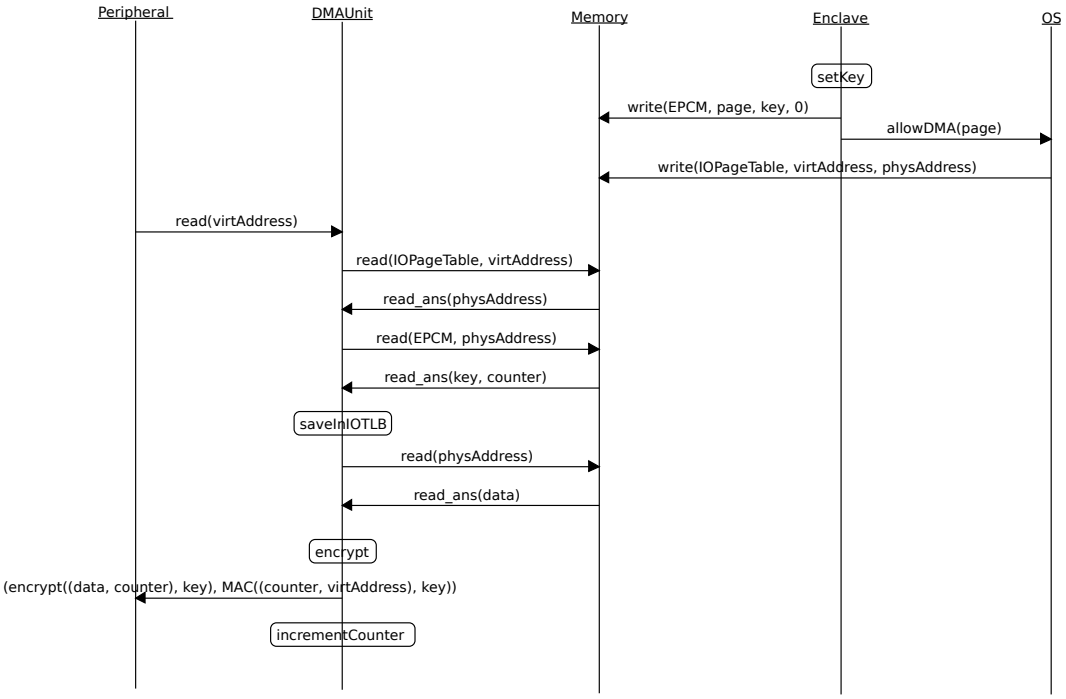



Figure 3.9 – Secure DMA read.

On the one hand, the OS controls the I/O page table needed by the DMA remapping unit to translate the virtual address to a physical one. On the other hand, each time a peripheral accesses an enclave page, the key corresponding to this enclave page is used to encrypt data exchanged. Since the address of the EPCM entry is computed from the enclave page physical address, an enclave (or the OS) is able to access memory from an enclave page if and only if it knows the key stored in the corresponding EPCM entry. However, the CPU guarantees that only the enclave owning a particular enclave page can set the key in the corresponding EPCM entry. If this key is shared with the peripheral in a secure way (using the previously presented protocol) and not leaked from the peripheral and the enclave, this guarantees that only the peripheral is able to access the content of the enclave page.

### Theoretical analysis

When a peripheral needs to access a protected page for the first time, the virtual address used by the peripheral needs to first be translated to a physical address by the DMA remapping unit based on the I/O page table. This process is the same as when non protected memory is targeted. However, once the address has been translated, the key and counter need to be fetched which results in one additional memory access (with MEE decryption). Compared to non-enclave DMA, following DMA transfers would need to be decrypted by the MEE (if data is not cached), then encrypted once again by the DMA remapping unit and decrypted by the peripheral. Since we envision that the DMA remapping unit would use the same cryptographic algorithm than the MEE, we can evaluate that the encryption and MAC will introduce a delay corresponding to one MEE round. If the peripheral uses the same hardware support to decrypt DMA reads, we can approximate the mean overhead per memory read in a same page introduced by protecting

the path from peripheral to memory as:

$$\tau = \frac{1-q}{n} * (\delta_{mem} + \delta_{mee}) + (3-p) * \delta_{mee}$$

where

- $q$ is the probability that the accessed page has a corresponding entry in the I/O TLB
- $p$ is the probability that the accessed data has been cached
- $\delta_{mem}$ is the delay introduced by one memory access
- $\delta_{mee}$ is the delay introduced by one MEE round
- $n$ is the number of memory accesses done on the same page

$q$ depends on the number of peripheral issuing DMA requests and on the range of memory accessed by the considered peripheral. $p$ depends on the space / time locality of memory accesses from the peripheral and CPU core. If the enclave and the peripheral exchange a lot of data from a localized area, the overhead will roughly be

$$\tau = 2 * \delta_{mee}$$

In [120], the author states that the average overhead induced by the MEE is 5.5% (the result is obtained by running experiments on the SPECINT2006 benchmark). We can thus expect an overhead of around 11% in our case for applications with localized memory accesses. While the impact on performance is not negligible, it should be weighted against the strong security guarantees provided by the cryptographic primitives.

## 3.4 Conclusion

In this chapter, we have proposed a design that allows to build a secure trusted path between a peripheral and an application against an attacker model stronger than the ones that have been considered by the previous works until now (as presented in section 2.1.4.2). Indeed, the communication between the peripheral (it could be an authentication device or a hardware accelerator for instance) and the application is protected against an attacker who controls the entire software stack except the application (including OS, hypervisor) and can physically probe all buses external to the processor die or the peripheral.

To provide such guarantees, we had to rely on cryptographic primitives, which explains the overhead incurred by this secure design. However, we have made it so other communication links and application are almost not affected (the only verification added to every DMA transfer is to check whether it is targeting the PRM or not). The hardware TCB is only increased to contain the peripheral in order to guarantee that the key used by the cryptographic primitives do not leak.

In order to assess the security of this design, formal verification may be used to check properties on different features of the design. It is for instance possible to verify that the key exchange protocol does not leak the key. It would also be possible to verify that the DMA transfer protocol is correct from a high-level: meaning that only the peripheral which has just shared the key with the application is able to access the enclave pages. It could also be desirable to prove that a low-level implementation in the DMA remapping unit complies to the high-level description of its behaviour given here.

In the following chapters, we propose to discuss how formal verification of such a design could be performed. In particular, we first focus on how formal security verification can be integrated to high-level models used during the conception of an embedded system (and we provide a model and formal proof of the key establishment protocol presented in this chapter). Then we will discuss how formal verification can be performed when hardware and software components are tightly coupled.

# Chapter 4

# Formal security verification of embedded system from design models

One objective of this thesis was to integrate modeling artifacts for security features to a design methodology targeting embedded system design and to provide fast, simple and informative formal verification of these models. I personally focused on system verification from software design models (presented in section 4.3) but I have also worked on integrating security verification to the partitioning stage (presented in section 4.4).

We will first discuss how security requirements may be integrated to embedded system design methodologies and explain why we focused on the SysML-Sec approach [217]. We will then present our contribution and how they integrate to SysML-Sec.

## 4.1 Introduction

### 4.1.1 Motivation

While the security of cryptographic protocols benefits from regular, accurate and thorough analysis (and in particular, formal analysis), strict security development processes still have trouble finding their ways to industrial applications. To non-specialists, security is often seen as *the right way to use the right tools*. This however leads to subtle bugs when out-of-the-box cryptographic solutions are not suitable, and in particular when the importance of an asset or communication is misunderstood.

Such a security issue can be minor when the number of affected devices is small and when the vulnerability can be easily fixed (e.g., with a software patch). However, this is typically not the case for embedded systems where design flaws can be impossible to fix and can affect a whole range of products. Even when a security vulnerability is discovered before the product is released, the amount of work needed to rethink the whole architecture may be prohibitive.

This calls for an accurate embedded system design methodology—and, in particular a methodology supported by a language to model both hardware and software components—that enables early security verification. In order to provide reliable and fast feedback to the designer, such a methodology should enable to build formal proofs on high-level models.

### 4.1.2 Integrating security considerations during the conception of embedded systems

In [195], Nhlabatsi et al. propose to classify approaches targeting security modeling during system conception into four classes:

- Goal-based approaches [181,207,212,250] focus on progressively refining goals into a set of sub-goals linked by logical relationship. These methods enable to reason about high-level goal relationship but are not directly concerned about capturing behaviours.
- Model-based approaches [129, 141, 174, 217, 218] aim at integrating security elements to existing model-driven software development methodologies.
- Problem-oriented approaches [155,171,237] focus on attack scenarios and integrate security as countermeasures to these attacks. As goal-based approaches, functional behaviours are described to support higher-level security assertions (when they are described at all).
- Process-oriented approaches [185, 244] focus on how security analysis is performed in the global process of system design. They are not concerned with the design at hand but with the process the conception goes through.

In this thesis, we were interested in formally verifying behaviours (described either by high-level models or low-level implementations) against security properties. In order to take into account the behaviour of a system and its security, we thus require to base our verification method on a modeling language that would enable describing functional and non-functional behaviours. Our choice was thus drawn to model-based approaches.

Among various model-based approaches, we chose SysML-Sec since:

- It specifically targets embedded system design. Thus, it allows to model HW/SW partitioning and supports modeling of low-level properties (computational complexity, clock frequency, FIFO policies, etc.).
- It builds upon a recognized language. SysML is widely used both in the industrial (e.g., Thales) and academic world.
- It proposes to integrate security features to multiple phases of the design process. In particular, its model-based approach to system design is suitable for formal verification.
- It is supported by a free and open-source toolkit and thus enables to easily develop algorithms that leverage existing modeling capabilities to perform formal verification with the same tool that is used for modeling.
- This toolkit already implements various other formal verification algorithms targeting performance or safety properties. Performing safety and security verification from the same model can greatly increase the reliability of the proofs as mechanisms implemented to guarantee security may impact safety or performance results and modifications targeting safety or performance may endanger the security of a design.

## 4.2 SysML-Sec

SysML-Sec is a model-driven approach to design embedded systems with safety, security and performance constraints.

### 4.2.1 Methodology and diagrams

#### 4.2.1.1 Overview

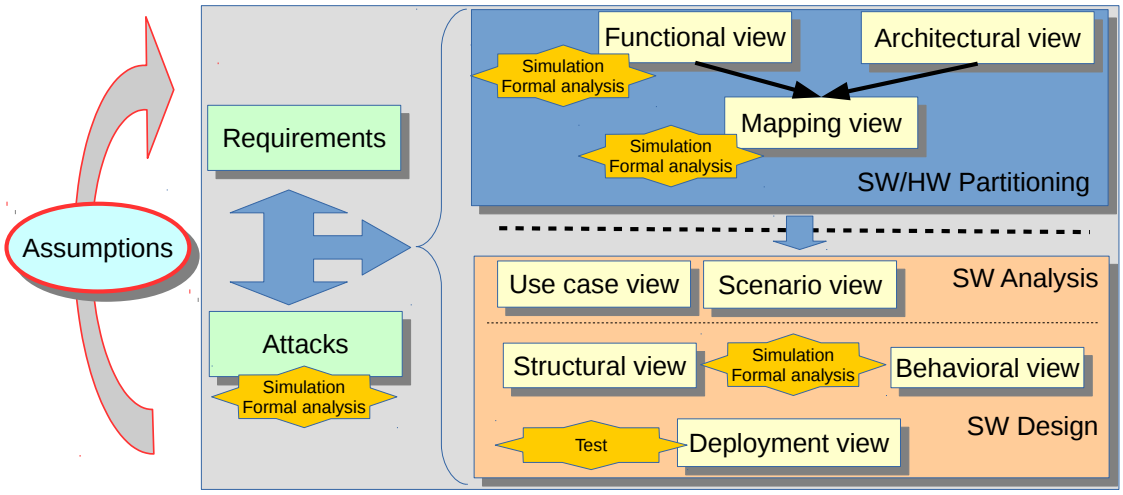



Figure 4.1 – Overall SysML-Sec Methodology.[1]

Basically, SysML-Sec supports three main modeling phases (see Figure 4.1):

1. The **requirement analysis** phase targets the analysis of desirable properties or possible attack scenarios on the system. As this phase is not concerned with the question of *how* the system will be implemented but in *what* should be implemented, we will not detail it in this work.

2. The **system-level HW/SW partitioning** phase includes capturing functional elements of the target application, modeling candidate architectures and finally mapping functional elements—including communications between functions—to candidate architectures. Then follows a verification sub-phase in which safety, security and performance constraints are evaluated in order to select the "best" HW/SW partition.

3. A **software design** phase follows a successful partitioning phase. Software components are first built from high-level functions mapped onto processor nodes at the previous phase. Then, they are progressively refined. Refinement typically concerns the accurate description of algorithms and protocols, including security protocols.

Design elements of the last two phases are built from (safety and security) requirements. Verification is supported in all modeling stages in order to assess the security and safety requirements. Attack trees also help capturing potential attacks that are feasible in the considered mapping models.

TTool is a free and open-source tool that supports the different phases and models of SysML-Sec. TTool also offers a press-button approach for safety, security and performance verification, and can backtrace verification results to modeling views.

[1] Figure available at `http://sysml-sec.telecom-paristech.fr/`.

#### 4.2.1.2 Modeling partitioning in SysML-Sec

A SysML-Sec partitioning follows the Y-chart approach [146] comprising three modeling stages: application modeling, architectural modeling, and mapping modeling.

The application model comprises of a set of communicating tasks, modeled in a unified diagram corresponding to the SysML internal block and block definition diagrams (we will call it *SysML block diagram* for simplification). The behavior of each task is described abstractly in an Activity Diagram. Functional abstraction allows to ignore the exact calculations and data processing of algorithms, and consider only their relative execution time. Data abstraction allows to consider only the size of data sent or received, and ignore details such as type, values, or other details.

The architectural model displays the underlying architecture as a network of abstract execution nodes, communication nodes, and storage nodes. Execution nodes consist of CPUs and Hardware Accelerators. All execution nodes must be described by data size, instruction execution time, and clock ratio. CPUs can further be customized with scheduling policy, task switching time, cache-miss percentage, etc. Communication nodes include bridges and buses. Buses connect execution and storage nodes for task communication and data storage or exchange, and bridges connect buses. Buses are characterized by their arbitration policy, data size, clock ratio, etc., and bridges are characterized by data size and clock ratio. Storage nodes are Memories, which are defined by data size and clock ratio.

Mapping partitions the application into software and hardware by specifying the location of the implementation of functional tasks on the architectural model. A task mapped onto a processor will be implemented in software, and a task mapped onto a hardware accelerator will be implemented in hardware. The exact physical path of a data/event write may also be specified by mapping channels to buses, bridges and memories.

#### 4.2.1.3 SysML-Sec software design

A SysML-Sec software design is composed of SysML block and state machine diagrams. The block diagram describes the architecture and components of the system, and state diagrams describe their behavior. In a SysML block diagram, each component of the system is modelled by a block and the blocks are linked by channels to model communication links. The blocks can define attributes to model state conditions, methods to model internal procedures and signals to model external communications. Each block is associated with a state machine diagram which describes the high-level behaviour of the corresponding component. The state machine models attributes modifications, procedure calls, sending and receiving signals and the generic control flow of the component.

### 4.2.2 Formal verification and security

Verification takes place at different engineering phases. Simulation is mostly used at the partitioning stage in order to evaluate the impact of security mechanisms in terms of performance. Formal verification on design diagrams intends to prove safety or liveness properties of the system and its resilience to threats.

#### 4.2.2.1 During partitioning

Before we added formal verification of security during the partitioning stage in [165], mathematical analysis during the partitioning stage only concerned performance and safety evaluation. The

performance of the design is evaluated according to multiple criteria: bus and cpu load, latencies between an input and an output or schedulability of bus transfers.

#### 4.2.2.2 During software design

**Safety**

Safety verification can evaluate the absence of deadlocks, and the reachability and liveness of state machine states. More complex properties can be modeled either with a subset of CTL, or with the use of observers in the model that are expressed with state machine diagrams. TTool includes its own verifier that can also generate a reachability graph, and minimize it according to a set of user-selected observable properties. TTool can also rely on UPPAAL [162] for safety proofs.

Safety proofs take into account all design elements besides the ones specifically defined for security purposes: security-oriented pragmas and cryptographic methods that have no impact on safety properties (liveness, reachability).

**Security**

The software design stage includes the definition of security mechanisms, and the refinement of security requirements in security properties to be proved in the design.

Security verification is based on model transformation to a formal specification than can be handled by the ProVerif security prover [35]. A first version of model transformation was already published in [204] but it had strong limitations on the usage of choices and loops (as described in the next section). We will show in the next section how we have resolved these issues and we will give a formal description of the transformation algorithm.

In both of these versions, SysML-Sec blocks can define a set of methods corresponding to cryptographic algorithms, such as *encrypt*(), to enable to describe security mechanisms built upon these algorithms, e.g., cryptographic protocols. Blocks can also pre-share values, a feature commonly needed to set up cryptographic protocols.

The behaviours described with these elements are evaluated to verify confidentiality, authenticity and reachability properties. Authenticity appears in two forms: weak authenticity (also known as integrity) and strong authenticity (integrity and protection against replay). Strong authenticity is verified by taking into account multiple executions of the system (called *sessions*). This concept is similar to the one described in [35]. Note also that while reachability is traditionally qualified as a safety property, it is also useful for security as security properties can be expressed with observers. The reachability results cannot be reused from safety verification because some of the features of the models are ignored by security and safety verification.

Indeed, like safety verification, the proof of security properties abstracts away irrelevant or non supported system details. For example, in the attacker model we consider, an attacker is able to delay a message so temporal information provided on design diagrams are discarded. This results in a sound approximation where all attacks possible on a timed model exist on the un-timed approximation. The temporal information discarded is both the temporal operators (as the *after* operator), or the semantics of the channel. Indeed, while SysML-Sec allows to model blocking channels, the attacker could accept the message and send it later. Likewise, the attacker could play the role of an infinite queue and rearrange messages as she wishes. Other modeling elements, like loops, were ignored by the translation process.

## 4.3 Verification of security properties on SysML-Sec software design diagrams

We now propose an algorithm to translate a SysML-Sec software design into an applied pi-calculus specification to be handled by the ProVerif automated cryptographic protocol verifier. Modeling security behaviours during the software design phase is easier than during the partitioning phase as functional behaviour is not abstracted. We thus present first verification of security during software design before discussing partitioning.

In this section, we will first present a brief and informal description of the translation algorithm, then we will give a detailed formalization of this algorithm and finally discuss how ProVerif results are backtraced to the SysML-Sec models.

### 4.3.1 Overview

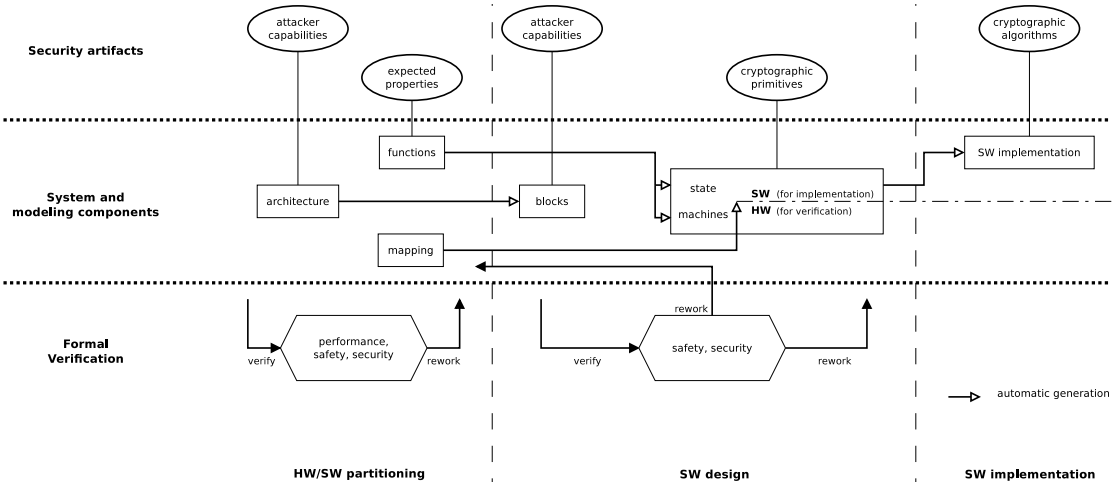



Figure 4.2 – Security and formal verification during embedded system design.

Figure 4.2 shows the integration of security modeling and formal verification during the different stages of SysML-Sec. Conception starts with the design-space exploration phase (discussed in the next section) and advances to a software design phase where software and hardware components are modelled with state machine diagrams. Software models are then refined to their concrete implementation. Hardware models are present during the software design phase to describe the behaviour of communicating agents and enable verification of whole systems. Security artifacts are added to the diagrams of the different phases to model high-level properties during partitioning and lower-level cryptographic primitives during software design.

#### 4.3.1.1 ProVerif

Formal verification of SysML-Sec software design diagrams was done by using the ProVerif prover.

**Description**

ProVerif is a toolkit that relies on Horn clauses resolution for the automated analysis of security properties over cryptographic protocols, under the Dolev-Yao model. ProVerif takes an input

as a set of Horn Clauses, or a specification in a restricted version of the applied pi-calculus together with a set of queries. ProVerif then outputs a result for each query, telling whether it is satisfied or not. In the latter case, ProVerif tries to identify a trace explaining how it came to the conclusion that a query is not satisfied.

When translating from SysML-Sec to ProVerif, we generate an applied pi-calculus specification. The generation takes advantage of the Horn clauses semantic to model concepts, such as loops in the state machine diagram, that are not directly supported by ProVerif. Note that relying on the semantic of Horn clauses to help ProVerif verification algorithm has proved to be effective by some other works [93].

Pi-calculus enables description of protocols in term of processes executing in parallel and exchanging messages over channels. In the applied pi-calculus language, messages can be terms over literals. Processes can split to create concurrently executing processes, and replicate to model multiple executions (called sessions) of a given protocol. Cryptographic primitives, such as symmetric and asymmetric encryption or hash, can be modeled through functions that are either constructors, which create new values, or destructors, which reduce the number of applied constructors in an expression.

An example of some ProVerif code is presented in Listing 4.1.

```
(* Functions *)
fun sencrypt (bitstring, bitstring): bitstring.
reduc forall x: bitstring, k: bitstring;
        sdecrypt (sencrypt (x, k), k) = x.
...

(* Variables *)
free token___QuotingEnclave___0: bitstring [private].
free token___ExternalVerifier___0: bitstring [private].
...

(* Queries *)
query event(enteringState___ExternalVerifier___Success()).
...

(* Sub-processes *)
let QuotingEnclave___0 =
    new strong___QuotingEnclave___02: bitstring;
    out (chControl, strong___QuotingEnclave___02);
    ...

(* Main process *)
process
    new QuotingEnclave___SGXPrivKey__data: bitstring;
    ...
```

Listing 4.1 – Global structure of the ProVerif file.

Starting from this specification, reachability, correspondence and confidentiality properties can be queried and ProVerif will present a result to the user that is either *true* if the property is verified, *false* if a trace has been found that falsifies the property, or *cannot be proved* if ProVerif failed in asserting or refuting the queried property. Such failures in proving properties over an unbounded number of sessions and unbounded message space are unfortunately unavoidable since this problem is undecidable [98]. Thus, sound approximations are made by ProVerif when translating the applied pi-calculus specification into Horn clauses.

### Reasons to use ProVerif

The choice of ProVerif as a formal verification tool seemed logical for our purpose as it is a state-of-the-art prover specifically targeting security properties on communicating components against an attacker conforming to the Dolev-Yao [94] attacker model. This attacker model describes a

powerful adversary which is able to read all messages sent between the components, create new messages, replay previously learned messages and apply cryptographic primitives. The attacker model was suitable for the verification of embedded systems as accessing communication buses between components is much easier than spying on their internal states. Moreover, ProVerif does not limit the states to be explored and does not require manual interaction.

Another possibility would have been to use a common language as provided by the AVISPA tool [16]. However, the description of the behaviours of the blocks in SysML-Sec state machine diagrams is closer to the imperative style of ProVerif than on the event and transition-based description used by AVISPA. We were also interested in the ability of ProVerif to define custom low-level elements (predicates, constructors, destructors, clauses) and in the possibility to add pointers to modify the verification algorithm (through `nounif` instructions for instance).

#### 4.3.1.2 Missing features in SysML-Sec software design diagrams

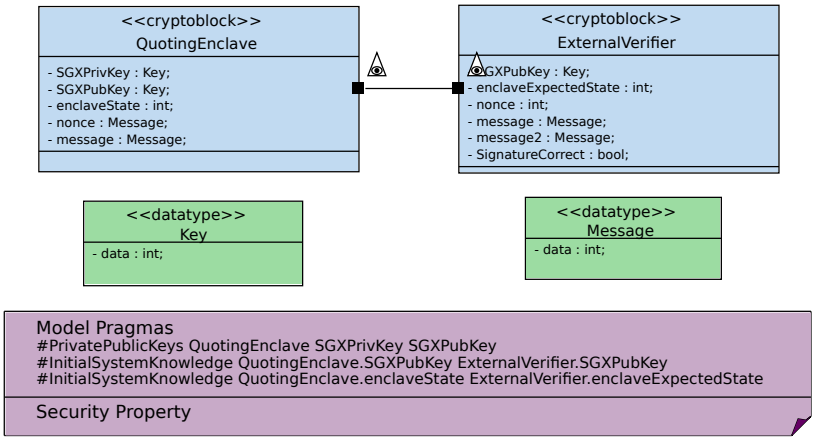


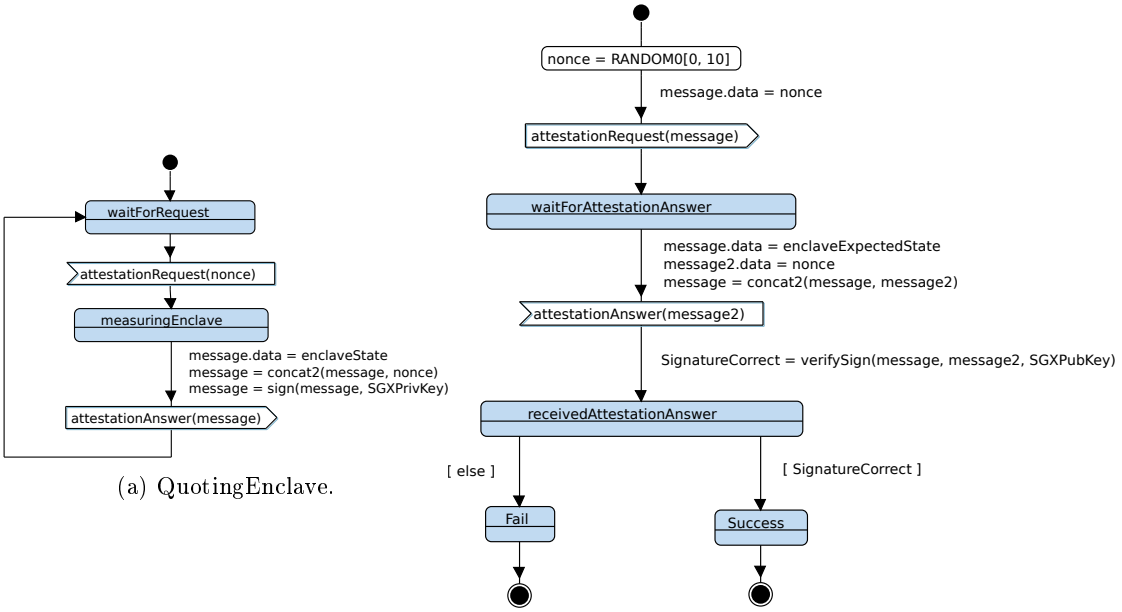

Figure 4.3 – Block diagram of Intel SGX model.

To illustrate the models and the transformation described in these sections, we provide a simplified model of the SGX attestation scheme using the SysML-Sec modeling language. As mentioned previously, a SysML-Sec software design is composed of a block diagram (Figure 4.3) and a state machine diagram for each block of the block diagram (Figure 4.4).

In the years since a first translation from a SysML-Sec model to ProVerif was proposed [204], valuable feedback was gathered from industrial and academic testers. Among easily corrected corner cases or usability bugs, some robustness issues indicated more fundamental problems. These issues were often related to a mismatch between the semantics of features in SysML-Sec and their counterparts in ProVerif. We present some of the most important issues that motivated us to completely rework the translation process:

**Loops**

Often, users take advantage of the verification feature embedded in TTool in order to assess the security of a system that provides a service. For some of these systems, modeling their state machines using loops seems natural. Unfortunately, ProVerif tries to flatten the given specification as a first step. By using a straightforward translation as it was implemented, verification on a design containing loops would not terminate. We will show how we address this significant drawback in sections 4.3.1.4 and 4.3.2.




(a) QuotingEnclave.

(b) ExternalVerifier.

Figure 4.4 – State machine diagrams of the blocks in the SGX model.

### Private Channels

As in ProVerif, SysML-Sec enables the user to model two types of channels between blocks: private and public. While their semantics should be the same for ProVerif, their behaviors with respect to reachability queries differ. ProVerif considers that a message sent should always be received. On private channels, only explicitly specified entities can read the sent message, while on public channels, the message can also be read by the attacker. When assessing the reachability of an event, ProVerif tries to reconstruct a trace and stop at the exact moment when the event is triggered. Therefore, if the event occurs just after a message was sent on a private channel, the message would not yet be read. ProVerif would thus consider the trace invalid, returning a *cannot be proved* result. As we wished, in our context, to provide the ability to prove reachability properties, modifying the private channel representation was also an important issue we addressed in our work.

### Trace reconstruction

If ProVerif manages to find a counterexample to prove that a property is not guaranteed by the model, it outputs a trace. In the original implementation of the transformation, the trace was directly provided to the designer to help her debug the models. However, ProVerif traces are tedious to read, in particular due to the artifacts added by the translation process.

#### 4.3.1.3 New modeling artifacts

The specificities of SysML-Sec block and state machine diagrams compared to their counterpart in SysML are manifold: first, a type for cryptographic material (keys) is added. This type enables to define cryptographic primitives in SysML-Sec blocks that basics operations such as encrypting, decrypting, computing hashes, verifying certificates, etc. A graphical artifact is added

to channels in SysML-Sec blocks diagrams when they are modelled as public (where an attacker can spy). Also, we extended SysML with pragmas, i.e., formal text notes regarding the attributes of the system (purple box on Figure 4.3). This enables for instance to declare two attributes equal at the start of a session. Pragmas are classified into *Model Pragmas* or *Property Pragmas*. Model Pragmas provide more security-oriented semantics to attributes of blocks, while Property Pragmas describe security properties that need to be verified on the system. The pragmas that we defined are described in Table 4.1.

| Pragma | Description |
|---|---|
| **Model Pragmas** | |
| PrivatePublicKeys | Two attributes of a block are set as Private Key and Public Key, respectively |
| InitialSessionKnowledge | Listed attributes have the same value at the start of a session |
| InitialSystemKnowledge | Listed attributes have the same value when the system starts |
| SecrecyAssumption | Listed attributes are assumed secret. ProVerif verifies this afterwards |
| Constant | Declares a string as a possible constant value |
| **Security Properties** | |
| Confidentiality/Secret | Query the confidentiality of attributes listed |
| Authenticity | Query the weak and strong authenticity of the two attributes at given states |

Table 4.1 – Pragma Descriptions.

#### 4.3.1.4 The transformation algorithm

In order to verify a SysML-Sec software design with ProVerif, the diagrams need to be first translated to a ProVerif specification. We will first give an informal description of the translation process in this section and then detail a formal description of the algorithm in the next section.

Some components of the SysML-Sec diagram can be mapped to their ProVerif counterparts quite straightforwardly since we borrow the attacker model and the channel semantic from ProVerif. However, for other components, like loops on the state machine diagram or private channels, we had to carefully consider ProVerif reasoning in order to avoid as much as possible cases where the proof would fail.

A first transformation, already implemented, translates timer-related capabilities into a set of blocks and signals. The latter are inherently taken into account in our formalization.

The transformation then can be described as happening in three steps: basic blocks creation, basic blocks translation and basic blocks linking (see Figure 4.5).

**Basic blocks creation**

Since a SysML-Sec state machine diagram may contain loops, we need either to unroll these loops up to a certain limit, or to reuse the parts of the translated specification that could be executed multiple times. Both of these choices have been considered in static analysis works [139, 161] and are strategically important, since allowing jumps in the specification often incurs a loss in terms of completeness, while flattening the graph by limiting the number of repetitions affects the correctness of the proof.

In order to maintain the soundness of the verification guaranteed by ProVerif, we chose not to limit the loops. As such, the first step in translating a SysML-Sec Diagram composed of

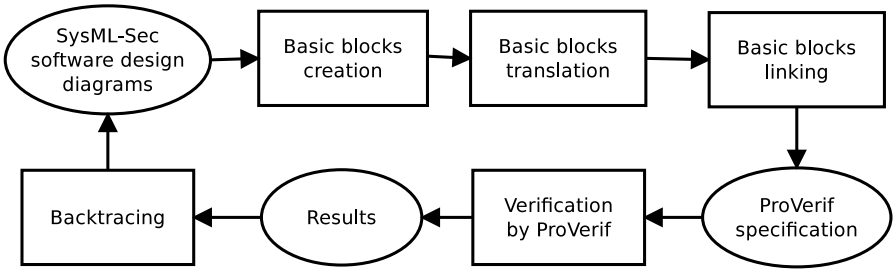



Figure 4.5 – Overview of the translation algorithm.

well-formed blocks to a ProVerif specification is to partition the state machines of the different blocks into atomic *basic blocks* that cannot be entered from two different locations. A basic block correspond to a maximum connected sub-graph of a state machine where no state may have multiple incoming edges. A connected state machine can thus be partitioned by a set of basic blocks (see Figure 4.6). Note that these basic blocks are trees and thus have a root state (initial state, $s_6$ and $s_7$ in Figure 4.6).

#### Basic blocks translation

The algorithm then translates each basic block into a process of the ProVerif specification. Most of the operators in the SysML-Sec state machines are translated to their counterparts in applied pi-calculus.

When a state is followed by multiple outgoing transitions, any of them can be taken as long as the boolean condition that guards the transition is verified. From a security point of view, any trace of attacks using one of these transitions is valid. This is conceptually equivalent to letting the attacker choose which transition to take in case of non-deterministic transitions. We model this by creating values corresponding to each transition, making them public, and waiting for the attacker to return a value corresponding to her choice.

#### Basic blocks linking

ProVerif supports simple process calls by flattening them, and as a result, does not terminate for diagrams containing loops. Thus, instead of directly calling basic blocks, we generate tokens at the end of each basic block and make them available to the attacker. The token contains a reference to the next basic block to execute and the current state of the system (the attributes of the blocks). One such token allows the attacker to execute the basic block whose initial state matches the token. When the token is used, the values of the attributes of the block are set according to the values passed as arguments of the token. This token should only be used once, and only in the right session, so we add to the token a session identifier to guarantee that the token is forged and used in the same session, and use a nonce mechanism to prevent its reuse.

### 4.3.2 Formal description of the translation

In order to provide a reliable mathematical base for the proofs performed by ProVerif on a SysML-Sec software design diagram, we decided to formally express the transformations performed on the model as it is implemented in TTool. This formalization will provide understanding on both how we managed to link a general purpose SysML diagram to a ProVerif specification that relies

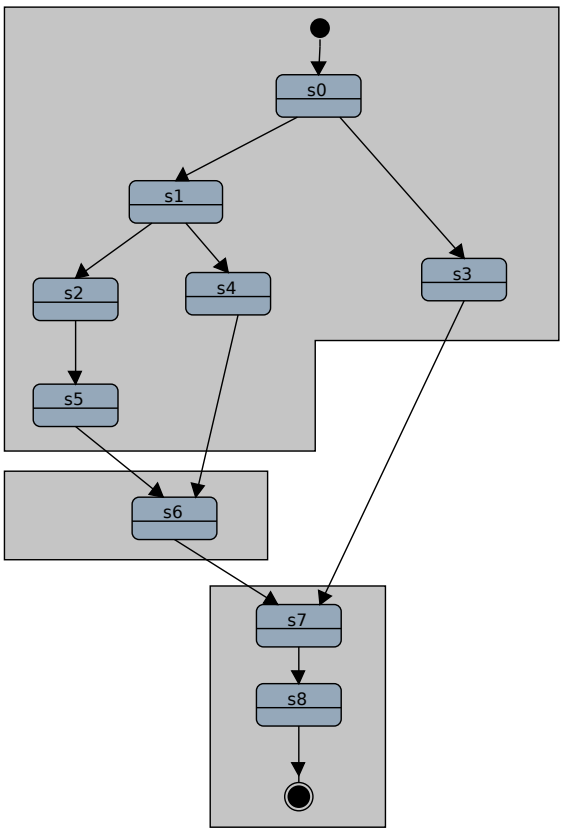



Figure 4.6 – A SysML-Sec state machine partitioned in basic blocks. Each basic block (greyed area) is a rooted acyclic sub-graph of the state machine where only the root of the sub-graph may have multiple incoming transitions in the original state machine.

on Horn clauses, and where sources of incompleteness may come from. This formalization is also a first step in building a mathematical proof of equivalence for a future work.

#### 4.3.2.1 SysML-Sec

In the software design phase, the SysML-Sec diagrams intend to describe a software *design*. This section provides a formal definition to software designs and their inner components.

**Definition 1 Design.** *A design is defined by a network of blocks interconnected by links and a set of pragmas:*

$$\mathcal{D} = \langle \mathcal{B}, \mathcal{C}, \mathcal{P} \rangle$$

Figure 4.3 displays two blocks corresponding to the actors in the SGX attestation protocol. The blocks are linked by a channel which can be private or public—as denoted by the illuminati symbol. In this formal description, we don't mention data types as they only act as syntactic sugar as far as security analysis is concerned.

**Definition 2 Block.** *A block is a tuple*

$$\mathsf{block} = \langle ident, \mathcal{A}, \mathcal{M}, \mathcal{S}, \mathsf{behav} \rangle$$

*where*

- *ident is a block name.*
- $\mathcal{A}$ *is a set of attributes.*
- $\mathcal{M}$ *is a set of methods.*
- $\mathcal{S}$ *is a set of directed signals. For each* $s \in \mathcal{S}$, $\mathsf{type}(s) \in \{in, out\}$.
- $\mathsf{behav}$ *is a state machine diagram.*

We define a function *block* that given a design $\mathcal{D}$ returns the set of its blocks, a function *sig* that given a block returns the set of its signals and a function *att* that returns the set of its attributes.

**Definition 3 Channel** *A channel connects signals between blocks.*

$$\mathsf{channel} = \langle \mathsf{type}, \mathsf{mode}, \mathcal{R} \rangle$$

*where*

- $\mathsf{type}$ *is a physical property which can be either* $\mathsf{private}$ *or* $\mathsf{public}$.
- $\mathcal{R}$ *is one-to-one correspondence between two sets of signals,* $\mathcal{R} \subseteq sig(b_1) \times sig(b_2)$ *where* $b_1, b_2 \in block(\mathcal{D})$ *such that* $\forall (s_1, s_2) \in \mathcal{R}$, $\mathsf{type}(s_1) \neq \mathsf{type}(s_2)$.

Pragmas enable to either describe properties of the system in the initial state, or to query a property of the design that will be checked during verification. To simplify the presentation of this description, we will consider that a pragma can only be of two types:

- either it expresses that two attributes have the same value at the beginning of the execution ($\mathcal{P}_{init}$)
- or it queries the confidentiality of an attribute ($\mathcal{P}_{secret}$).

**Definition 4 Pragma.** *Let* $\mathcal{D}$ *be a design. We define a pragma as a pair*

$$\mathcal{P} = (\mathcal{P}_{init}, \mathcal{P}_{secret})$$

*where*

$$\mathcal{P}_{init} \subset \Big( \bigcup_{\mathsf{b} \in block(\mathcal{D})} att(\mathsf{b}) \Big)^2$$

$$\mathcal{P}_{secret} \subset \bigcup_{\mathsf{b} \in block(\mathcal{D})} att(\mathsf{b})$$

**Definition 5 State Machine Diagram.** *A state machine diagram is a rooted directed graph*

$$\mathsf{behav} = \langle \mathcal{Q}, q_0, q_\perp, \mathcal{E} \rangle$$

*where*

- $\mathcal{Q}$ *is a set of activity state nodes.*
- $q_0 \in \mathcal{Q}$ *is an initial state node.*

- $q_\perp \in \mathcal{Q}$ *is a final state node.*
- $\mathcal{E} \subseteq \mathcal{Q} \times \mathcal{G}uards \times \mathcal{A}ctions \times \mathcal{Q}$.

A name is given by the designer to each state. We thus define a labelling function $\mathbb{L}$ that returns the name associated to a state. Given an edge $e = (q, g, a, q')$, we define functions $source(e) = q$, $gard(e) = g$, $action(e) = a$, and $target(e) = q'$. A trace $\sigma \in \mathcal{A}ctions^*$ in a state machine is a sequence of actions $a_0$, $a_1$,... $a_n$ such that there are $q_0$, $q_1$,... $q_n \in \mathcal{Q}$ with $(q_{i-1}, g, a_i, q_i) \in \mathcal{E}$ for all $i = 1, \ldots, n$.

The set $\mathcal{A}ctions$ of actions used in a state machine is defined as follows:

| | | | |
|---|---|---|---|
| $a \in \mathcal{A}ctions$ | ::= | $f(x_1, \ldots, x_n)$ | function call |
| | | $\mid x := exp$ | assignment expression |
| | | $\mid c\langle v \rangle$ | input action |
| | | $\mid \bar{c}\langle m \rangle$ | output action |
| | | $\mid \nu.x$ | random action |
| | | $\mid \varepsilon$ | empty action |

Expressions in SysML-Sec consist of values, variables and function calls.

$$exp \quad ::= \quad value \mid x \mid f(x_1, \ldots, x_n) \quad \text{Expressions}$$

With no loss of capabilities, we consider that a guard is of the form:

$$g \in \mathcal{G}uards \quad ::= \quad true \mid x \quad \text{Guards}$$

Figures 4.4a and 4.4b show the state machine diagrams of the two blocks in Intel SGX attestation scheme. Note that empty actions and "true" guards are not shown in the diagrams. Note also that multiple actions appear on each transition. This is semantically equivalent to multiple chained transitions, each of which bearing a single action and a `true` guard.

**Syntactic constraints on activity diagram**

TTool enforces some basic properties on the state machine diagrams, namely:

1. The initial state node may only occur in the source of an edge.
2. The final state node may only occur in the target of an edge.
3. For any state node except the final state note, there is at least one path from the initial state node to this node.
4. Any state node different from the final state node has at least an outgoing transition.

#### 4.3.2.2 The resulting ProVerif specification

As described in [38], ProVerif specifications are described in a custom language following a well-defined structure. Our translation, described in section 4.3.2.3, does not make use of the entirety of the ProVerif language. We describe here a subpart of the ProVerif language that is useful for our purpose.

In particular, we consider that a ProVerif specification consists of the following five sections (the ProVerif associated keywords are given inside parentheses):

- **Functions** declaration (*fun* and *reduc* keywords). They are typically used to describe cryptographic primitives such as *hash*, *symmetric encryption*, etc. and don't depend on the particular design we are translating.
- **Variables** declaration (*channel* and *free* keywords). They declare channels and other variables that are shared by every participants and can be either public or private.
- **Queries** (*query* keyword) express the security properties that a user wishes to prove on the design
- **Sub-processes** declarations (*let* keyword). Each sub-process (or macro) declaration contains a behavioral description of part of the state machine diagrams of the design. They may be referenced by other sub-processes or by the main process. If they are not referenced by anyone, they are simply ignored.
- The **main process** (*process* keyword), which is the entry point of the design. It can reference any sub-process.

This global structure can be seen in the ProVerif code presented in Listing 4.1.

In particular, we see a constructor declaration (`sencrypt`), a destructor declaration (`sdecrypt`), two shared variables declarations (for the variables named `token___QuotingEnclave___0` and `token___ExternalVerifier___0`), a reachability query, the declaration of a sub-process (whose name is `QuotingEnclave___0`) and the main process which creates a new private variable (named `QuotingEnclave___SGXPrivKey___data`).

Once the sub-processes have been flattened into the main process, the syntax of the ProVerif language to describe a process is given in Figure 4.7 (inspired by the presentation in [9]).

Figure 4.7 – Syntax of the part of the ProVerif language used in our translation.

| | |
|---|---|
| $M, N ::=$ | terms |
| $x, y, z$ | variable |
| $a, b, c$ | name |
| $f(M_1, ..., M_n)$ | constructor application |
| | |
| $P, Q ::=$ | processes |
| $0$ | nil |
| $P \mid Q$ | parallel composition |
| $!P$ | replication |
| $new\, a;\ P$ | restriction |
| $out(M, N);\ P$ | output |
| $in(M, N);\ P$ | input |
| $let\, x = g(M_1, ..., M_n)\, in\, P\, else\, Q$ | destructor application |
| $if\, M = N\, then\, P\, else\, Q$ | conditional |
| $event(a);\ P$ | event |

In [9], the semantics of the ProVerif language is provided as a set of reduction rules. This is the definition that we use for our translation as well. These reduction rules operate on a configuration represented as a triple $\mathcal{E}, \mathcal{P}, \mathcal{S}$ where *"$\mathcal{P}$ is a multiset of processes, $\mathcal{E}$ is the set of free names of $\mathcal{P}$ and of names created by the adversary, and $\mathcal{S}$ is the set of terms known by the attacker."* The reduction rules are given in Figure 4.8. In these rules, the notation $P\{y/x\}$ means the process $P$ where $x$ has been substituted with $y$. Note that in the semantics, the conditional

construction does not appear. Indeed, as it is explained in [9], this conditional can be represented by a destructor that can only be reduced if the two terms appearing in the conditional are equals: $let\, x = equals(M, N)\, in\, P$ with $equals(M, M) \rightarrow M$. Also note that the rule *Red Event* that we added has no effect on the configuration since events only cause side-effects (not related to the demonstration of confidentiality) that are useful for other properties (reachability and authenticity) which are not presented in this formalisation.

As in [9], we use the notation $fn(P)$ (or $fn(M)$) to designate the set of free names in the process $P$ (or term $M$). We also say that $\mathcal{T}_{P_0}$ is a trace of $P_0$ from $\mathcal{S}_0$ if it is a finite sequence of reductions $fn(P_0) \cup \mathcal{S}_0$, $\{P_0\}$, $\mathcal{S}_0 \rightarrow \ldots \rightarrow \mathcal{E}', \mathcal{P}', \mathcal{S}'$. We also say that the closed term $M$ is secret from $\mathcal{S}_0$ if $fn(M) \subset fn(P_0)$ and there is no trace of $P_0$ from $\mathcal{S}_0$ which contains a state $\mathcal{E}, \mathcal{P}, \mathcal{S}$ where $M \in \mathcal{S}$.

#### 4.3.2.3 Translation

We now give the semantic of a SysML-Sec software design, expressed as a translation from SysML-Sec software designs into ProVerif specifications. For each SysML-Sec software design $\mathcal{D}$, the interpretation function is expressed under the form:

$$[\![\mathcal{D}]\!]_{\mathcal{E}} = F_{\mathcal{E}}(\mathcal{D}) \oplus V_{\mathcal{E}}(\mathcal{D}) \oplus Q_{\mathcal{E}}(\mathcal{D}) \oplus P_{\mathcal{E}}(\mathcal{D}) \oplus \texttt{"process"} \oplus Main_{\mathcal{E}}(\mathcal{D})$$

It relies on several auxiliary functions for expressing the semantics of specific parts of the designs. The core entities of this semantics include: $F_{\mathcal{E}}(\mathcal{D})$ generating functions, $V_{\mathcal{E}}(\mathcal{D})$ generating variables, $Q_{\mathcal{E}}(\mathcal{D})$ generating a set of queries from pragmas, $P_{\mathcal{E}}(\mathcal{D})$ generating a set of processes, and $Main_{\mathcal{E}}(\mathcal{D})$ that generates the process that manages global instantiation of other processes. The construction of these functions relies on an *environment* $\mathcal{E} = (\mathcal{E}_q, \mathcal{E}_v)$ that keeps track of the states that have to be visited ($\mathcal{E}_q$) and those that have already been ($\mathcal{E}_v$).

Before defining the interpretation function, it is helpful to introduce some notations. We use the quote (") character to indicate the beginning and ending of a string (ProVerif instruction). Quoted strings placed next to each other are concatenated (by $\oplus$ operator) to a single string (source code). $\overrightarrow{a}^{a \in \mathcal{S}}$ denotes a list of parameters over the set $\mathcal{S}$.

**Non-processes declarations**

Functions include a list of cryptographic primitives that can be used in the SysML-Sec software design and that are common to all designs: hash, symmetric encryption and decryption, asymmetric encryption and decryption, etc. and the two functions tok and untok (see below). Also, a pair of encryption and decryption functions is added for each private channel.

Variables consist of one channel used for public communication, one channel for control messages (chctrl) and one token_ ... variable (see next paragraph) for each basic block.

Note that the token_ ... variables can only be generated once we know the list of basic blocks. This list is constructed during the generation of sub-processes that thus need to be created before variables can actually be declared.

For each variable v for which the designer would like to check the confidentiality, we generate a query of the form "query attacker(new v)".

Figure 4.8 – Reduction rules of the semantics

$$\mathcal{E}, \mathcal{P} \cup \{0\}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P}, \mathcal{S} \qquad \text{(Red Nil)}$$

$$\mathcal{E}, \mathcal{P} \cup \{event(a);\, P\}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P} \cup \{P\}, \mathcal{S} \qquad \text{(Red Event)}$$

$$\mathcal{E}, \mathcal{P} \cup \{P \,|\, Q\}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P} \cup \{P, Q\}, \mathcal{S} \qquad \text{(Red Par)}$$

$$\mathcal{E}, \mathcal{P} \cup \{!P\}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P} \cup \{P, !P\}, \mathcal{S} \qquad \text{(Red Repl)}$$

$$\frac{a' \notin \mathcal{E}}{\mathcal{E}, \mathcal{P} \cup \{new\, a;\, P\}, \mathcal{S} \;\rightarrow\; \mathcal{E} \cup \{a'\}, \mathcal{P} \cup \{P\{a'/a\}\}, \mathcal{S}} \qquad \text{(Red Restr)}$$

$$\frac{M \notin \mathcal{S}}{\mathcal{E}, \mathcal{P} \cup \{in(M, x);\, P,\, out(M, N);\, Q\}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P} \cup \{P\{N/x\}, Q\}, \mathcal{S}} \qquad \text{(Red I/O)}$$

$$\frac{g \text{ destructor of arity } n,\; g(M_1, ..., M_n) \;\rightarrow\; M}{\mathcal{E}, \mathcal{P} \cup \{let\, x = g(M_1, ..., M_n)\, in\, P\, else\, Q\}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P} \cup \{P\{M/x\}\}, \mathcal{S}} \qquad \text{(Red Let1)}$$

$$\frac{g \text{ destructor of arity } n,\; g(M_1, ..., M_n) \not\rightarrow M \text{ for all terms } M}{\mathcal{E}, \mathcal{P} \cup \{let\, x = g(M_1, ..., M_n)\, in\, P\, else\, Q\}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P} \cup \{Q\}, \mathcal{S}} \qquad \text{(Red Let2)}$$

$$\frac{M \in \mathcal{S}}{\mathcal{E}, \mathcal{P} \cup \{out(M, N);\, P\}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P} \cup \{P\}, \mathcal{S} \cup \{N\}} \qquad \text{(Red Out)}$$

$$\frac{M, N \in \mathcal{S}}{\mathcal{E}, \mathcal{P} \cup \{in(M, x);\, P\}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P} \cup \{P\{N/x\}\}, \mathcal{S}} \qquad \text{(Red In)}$$

$$\frac{f \text{ constructor of arity } n,\; M_1, ..., M_n \in \mathcal{S}}{\mathcal{E}, \mathcal{P}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P}, \mathcal{S} \cup \{f(M_1, ..., M_n)\}} \qquad \text{(Red Constr)}$$

$$\frac{g \text{ destructor of arity } n,\; M_1, ..., M_n \in \mathcal{S},\; g(M_1, ..., M_n) \;\rightarrow\; M}{\mathcal{E}, \mathcal{P}, \mathcal{S} \;\rightarrow\; \mathcal{E}, \mathcal{P}, \mathcal{S} \cup \{M\}} \qquad \text{(Red Destr)}$$

$$\frac{a' \notin \mathcal{E}}{\mathcal{E}, \mathcal{P}, \mathcal{S} \;\rightarrow\; \mathcal{E} \cup \{a'\}, \mathcal{P}, \mathcal{S} \cup \{a'\}} \qquad \text{(Red New)}$$

### Processes generation

Sub-processes are generated by walking through the state machine diagram of every block of the SysML-Sec software design. To do this, the interpretation function relies on a queue of states to be visited $\mathcal{E}_q$ that is initialized to contain the start state of the state machine of each block, and a list $\mathcal{E}_v$ that contains all the states that have already been visited (empty at the beginning). While there still are queued states, one of them (let's call it $s$) is dequeued (it does not matter which one is picked), it is added to $\mathcal{E}_v$ and a new sub-process is created through the function $[\![s]\!]^p_{\mathcal{E}}$ (see Table 4.2). This function will walk through a subpart of the state machine that we call a *basic block*. A basic block is a sub-graph whose root has multiple incoming transitions. The interpretation function of a state is given by the function $[\![.]\!]^s_{\mathcal{E}}$. The interpretation function for single outgoing edge is denoted $[\![.,.]\!]^t_{\mathcal{E}}$ and for multiple outgoing edges is denoted $[\![.]\!]^m_{\mathcal{E}}$. The interpretation of an action is given by the function denoted $[\![.,.]\!]^a_{\mathcal{E}}$. The continuation of the interpretation of following states is completed by $[\![.]\!]^c_{\mathcal{E}}$ function.

In our interpretation, we use the terminology *fresh* variable which means that the variable is a new one and it has no occurrence anywhere in the code except in the instruction that creates it. We use $\mathcal{O}ut$ function that returns the set of transitions $\mathcal{O}ut(q)$ which are outgoing from the state node $q$.

We define a predicate *UniqueOut* that takes a state $q$ and returns true if no two different transitions have $q$ as a source state.

$$UniqueOut(q) \Leftrightarrow \left( \begin{array}{c} \forall (q_1,g_1,a_1,q_1'),(q_2,g_2,a_2,q_2') \in \mathcal{E}. \\ q_1 = q \wedge q_2 = q \Rightarrow g_1 = g_2 \wedge a_1 = a_2 \wedge q_1' = q_2' \end{array} \right)$$

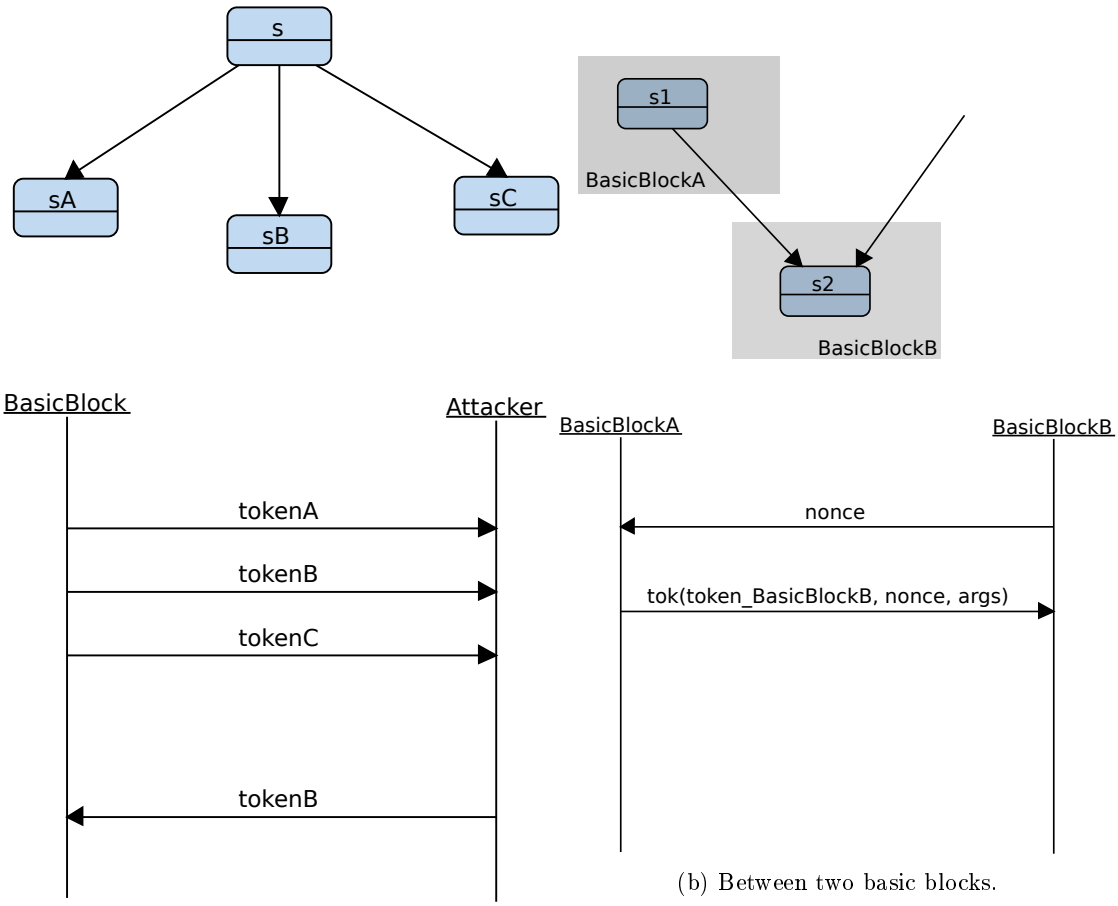
In the same way, we define a predicate *UniqueIn* that takes a state $q$ and evaluates to true if no two different transitions have $q$ as a target state.

$$UniqueIn(q) \Leftrightarrow \left( \begin{array}{c} \forall (q_1,g_1,a_1,q_1'),(q_2,g_2,a_2,q_2') \in \mathcal{E}. \\ q_1' = q \wedge q_2' = q \Rightarrow q_1 = q_2 \wedge g_1 = g_2 \wedge a_1 = a_2 \end{array} \right)$$

The translation of a basic block from a given block $b$, as described in Table 4.2, is mostly straight forward. Transitions are translated by transforming their guards into `if` conditions and their actions into ProVerif instructions. States are translated to a corresponding ProVerif `event` used for reachability queries. Two transformations deserve special care: multiple outgoing transitions and transitions linking states of two different basic blocks (which corresponds to multiple incoming transitions). When there are multiple outgoing transitions, the resulting ProVerif process generates a token for each possible transitions and makes them available to the attacker. Then it triggers the path by asking the attacker to accept one token. This is illustrated by Figure 4.9a. When control should be passed to another basic block, the process also generates a token. This token must contain the current state of the block (as described by its attributes) and the identifier of the basic block to be called (the `token_` variables). Also, in order to prevent the attacker from replaying previous tokens, the token includes a nonce that is issued by the callee. This token is protected from modification and spying by the attacker by encapsulating it into a private function `tok`. This is illustrated by Figure 4.9b.

The main process is then appended to the end of the ProVerif specification. Its purpose is first to create one unique `tok(...)` message for each state machine so that the attacker can *call*[2] the process corresponding to each basic block whose root is the initial state of a state machine. To create each token for a block, the main process needs to instantiate the attributes of the

[2]The term *call* here is abusive. Indeed, the attacker has no control over the execution flow of each process. It is however able to pass a token to a particular process which is blocked waiting for it.




(a) Non-deterministic transitions.

(b) Between two basic blocks.

Figure 4.9 – Special case of translation for transitions.

block, wait for a nonce and send the token. Then, it runs all the created processes in parallel (as denoted by the | operator) infinitely (as denoted by the ! operator).

$$Main_{\mathcal{E}}(\mathcal{D}) = \Big( \bigoplus_{b \in block(\mathcal{D})} \Big( \bigoplus_{a \in att(b)} \text{"new } a\text{;"} \oplus \text{"in(chctrl, nonce);} \\ \text{out(chctrl, tok(token\_}\mathbb{L}(q_0), \text{nonce, } args))\text{"} \Big) \Big) \\ \text{"}\Big|\text{"}_{q \in \mathcal{E}_v} \Big( \text{"!proclabel\_}\mathbb{L}(q)\text{"} \Big)$$

with $args = \overrightarrow{a}^{a \in att(b)}$

Table 4.2 – Interpretation function of State Machine Diagrams.

$$[\![q]\!]^p_{\mathcal{E}} = \begin{cases} \text{"let proclabel\_}\mathbb{L}(q) = \\ \text{new nonce;} \\ \text{out (chctrl, nonce);} \\ \text{in(chctrl, token);let (=token\_}\mathbb{L}(q)\text{, =nonce, } args) = \text{untok(token)"} \oplus [\![q]\!]^s_{\mathcal{E}} \\ \\ \texttt{with } args = \overrightarrow{a}^{a \in att(b)} \end{cases}$$

$$[\![q]\!]^s_{\mathcal{E}} = \begin{cases} \text{"."} & \texttt{if } q = q_\perp \\ \text{"event enteringState\_}\mathbb{L}(q)\text{();"} \oplus [\![q, e]\!]^t_{\mathcal{E}} & \texttt{if } UniqueOut(q) \\ \text{"event enteringState\_}\mathbb{L}(q)\text{();"} \oplus [\![q]\!]^m_{\mathcal{E}} & \texttt{otherwise} \end{cases}$$

$$[\![q, e]\!]^t_{\mathcal{E}} = \begin{cases} \text{"if } guard(e) \text{ then"} \oplus [\![q, e]\!]^a_{\mathcal{E}} & \texttt{if } guard(e) \neq \text{true} \\ [\![q, e]\!]^a_{\mathcal{E}} & \texttt{otherwise} \end{cases}$$

$$[\![q]\!]^m_{\mathcal{E}} = \begin{cases} \bigoplus_{e \in \mathcal{O}ut(q)} \text{"new } x_e\text{; out (chctrl, } x_e\text{); "} \oplus \text{"in (chctrl, } c\text{);"} \bigoplus_{e \in \mathcal{O}ut(q)} \left( \text{"if } c = x_e \text{ then"} \oplus [\![q, e]\!]^t_{\mathcal{E}} \right) \\ \texttt{where } c \texttt{ and } x_e \texttt{ are fresh variables} \end{cases}$$

$$[\![q, e]\!]^a_{\mathcal{E}} = \begin{cases} \text{"let } x = exp \text{ in"} \oplus [\![target(e)]\!]^c_{\mathcal{E}} & \texttt{if } action(e) = x := exp \\ \text{"new } x\text{;"} \oplus [\![target(e)]\!]^c_{\mathcal{E}} & \texttt{if } action(e) = \nu.x \\ \text{"out } (c, m)\text{;"} \oplus [\![target(e)]\!]^c_{\mathcal{E}} & \texttt{if } action(e) = \bar{c}\langle m \rangle \\ \text{"in } (c, v)\text{;"} \oplus [\![target(e)]\!]^c_{\mathcal{E}} & \texttt{if } action(e) = c\langle v \rangle \\ [\![target(e)]\!]^c_{\mathcal{E}} & \texttt{if } action(e) = f(x_1, \ldots, x_2) \mid \varepsilon \end{cases}$$

$$[\![q]\!]^c_{\mathcal{E}} = \begin{cases} [\![q]\!]^s_{\mathcal{E}} & \texttt{if } UniqueIn(q) \\ [\![q]\!]^b_{\mathcal{E}} & \texttt{otherwise} \end{cases}$$

$$[\![q]\!]^b_{\mathcal{E}} = \begin{cases} \text{"in(chctrl, nonce);out(chctrl, tok(token\_}\mathbb{L}(q))\text{, nonce, } args))\text{."} & \texttt{if } q \in \mathcal{E}_v \text{ or } q \in \mathcal{E}_q \\ \text{"in(chctrl, nonce);out(chctrl, tok(token\_}\mathbb{L}(q))\text{, nonce, } args))\text{."} & \texttt{otherwise} \\ \quad \mathcal{E}_q = \mathcal{E}_q \cup \{q\} \\ \\ \texttt{with } args = \overrightarrow{a}^{a \in att(b)} \end{cases}$$

### Correctness and completeness

Thanks to the presented translation, we have given an expression of the studied design in a ProVerif formalism. From the designer point of view, the interesting property to prove on the translation algorithm is that a protocol, as described by the SysML-Sec state machines, guarantees confidentiality. The value of such a proof depends on the validity of the translation algorithm. In this section, we propose to give a formal proof of correctness of the translation with respect to the property of secrecy.

We first need to formally describe the capabilities of the attacker in the SysML-Sec model. These capabilities (which follow the Dolev-Yao model as stated earlier) can be formalized as an implicit state machine described by Definition 6.

**Definition 6** ***Attacker state machine.*** *The attacker state machine is a graph composed of one state and four transitions from this state to the same state (see Figure 4.10). These four transitions are labeled with an action:*

- *in which receives a term on a channel,*
- *out which sends a term on a channel,*
- *new which creates a new name and*
- *function which applies an existing function to terms.*

*Informally, the semantics of the attacker is given by explaining how each step of a trace in its state machine affects a* configuration. *This configuration is composed of*

- *a set of names created by the attacker*
- *and a set of terms known by the attacker.*

*For each in$(M)$ transition, the term $M$ is added to the set of known terms, the out$(M)$ transition can be triggered if $M$ belongs to the set of known terms, new$(x)$ adds $x$ to the set of names created by the attacker and the set of known terms and function$(f(M_1, ..., M_n))$ adds $f(M_1, ..., M_n)$ to the set of known terms if $f$ is a constructor or adds $M$ to the set of known terms if $f$ is a destructor and $f(M_1, ..., M_n) \rightarrow M$.*

*In order to distinguish the actions of the attacker from actions of the other blocks, we will designate these transitions by $atk_{in}$, $atk_{out}$, $atk_{new}$ and $atk_{function}$ in the rest of this chapter.*

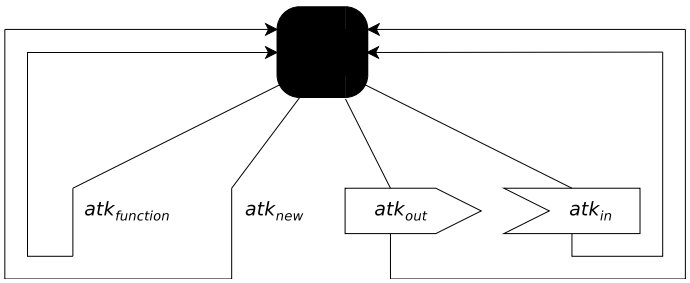



Figure 4.10 – Attacker state machine.

Through this state machine, the attacker is able to interact with the state machines of the SysML-Sec blocks. To give a formal description of this interaction, we define the parallel composition of the state machines of the SysML-Sec software design.

**Definition 7** ***Parallel composition.*** *Given a software design $\mathcal{D} = \langle \mathcal{B}, \mathcal{C}, \mathcal{P} \rangle$, we call parallel composition of $\mathcal{D}$ (noted $\mathcal{C}_\mathcal{D}$ or simply $\mathcal{C}$ in the rest of this formalisation) the automaton whose states are elements of the cartesian product of the states of the state machine of each $b \in \mathcal{B}$ and whose transitions from $(s^0, ..., s^n)$ to $(s'^0, ..., s'^n)$ are either:*

- *corresponding to $atk_{new}$, to $atk_{function}$ or to a transition from the state machine of a block $b \in \mathcal{B}$, i.e., $\exists i \in [\![0, n]\!]$ such that $(s^i, s'^i)$ is a transition of the state machine of $b$. In this case, the transition is guarded by the guard of the transition in the original state machine or by $\epsilon$ in the case of $atk_{new}$ and $atk_{function}$ and the action is either the action of the original state machine, or $atk_{new}$, or $atk_{function}$.*
- *or corresponding to two transitions in the original state machines: one bearing an input action (of a block or $atk_{in}$) and one bearing an output action (of a block or $atk_{out}$). In this case, the guard of the transition is the conjunction of the two guards in the original state machines and the action is a special action which models the exchange between the input and output actions.*

**Definition 8** ***Execution trace.*** *We call an* execution trace *of the design $\mathcal{D}$ composed of $m$ blocks a path $\mathcal{T}_\mathcal{C}$ in the parallel composition $\mathcal{C}$ of $\mathcal{D}$:*

$$\mathcal{T}_\mathcal{C} = \mathcal{N}_1, \Sigma_1, \mathcal{K}_1, \sigma_1 \rightarrow ... \rightarrow \mathcal{N}_n, \Sigma_n, \mathcal{K}_n, \sigma_n$$

*Where $\forall i \in [\![1, n]\!]$:*

- *$\mathcal{N}_i$ contains the names of the attributes of all blocks, the names introduced by random actions and the names created with $atk_{new}$,*
- *$\Sigma_i = (s_i^0, ..., s_i^m)$ is an element of the cartesian product of the states of all state machines,*
- *$\mathcal{K}_i$ is the set of terms over $\mathcal{N}_i$ that are known to the attacker and*
- *$\sigma_i$ is a substitution of all the attributes, i.e., a function from the set of attributes to the set of terms over $\mathcal{N}_i$.*

*Moreover, we say that an execution trace is* valid *if all guards met on the path are satisfied.*

The secrecy property is defined (similarly to [9]) as:

**Definition 9** ***Secrecy in a SysML-Sec design.*** *The SysML-Sec software design preserves the secrecy of an attribute $a$ if and only if for any valid execution trace $\mathcal{T}$, there is no state $(\mathcal{N}, \Sigma, \mathcal{K}, \sigma)$ of $\mathcal{T}$ where $a \in \mathcal{K}$.*

There are two interesting properties to prove concerning the translation algorithm presented in this chapter (with respect to the property of confidentiality): a correctness property and a completeness property. These properties are presented in Theorem 1 and Conjecture 1.

**Theorem 1** ***Correctness.*** *If an attribute $a$ is secret in the resulting applied pi-calculus specification, then it is also secret in the SysML-Sec design (according to definition 9). This means that the translation does not introduce false negatives (a possible attack is not detected).*

**Conjecture 1** ***Completeness.*** *If an attribute $a$ is secret according to definition 9, the presented translation results in a ProVerif specification where the name corresponding to $a$ is confidential.*

Note that, due to the ProVerif approximation discussed in section 4.3.3.1, even if the presented translation is both correct and complete, this would not imply that a term $M$ would be secret according to definition 9 if, and only if, the secrecy of the corresponding name were to be proved by ProVerif in the resulting applied pi-calculus specification. Indeed, $M$ could be secret but the translation could generate a specification where the secrecy of the corresponding attribute could not be proved.

As presented in the previous section, some of the SysML-Sec concepts have quite straightforward counterparts in the resulting ProVerif specification. Basically, when a SysML-Sec state has multiple outgoing transitions, it means that either one of these transition can fire. Proving that no matter which one of these transitions is taken, an attribute remains confidential would be equivalent to proving the same property on a design where the attacker decides which transition the process should take. This is what is modeled in the ProVerif specification. When control is passed from a basic block to another, it should appear as if the instructions of both were literally one after the other. The nonce and `tok` functions ensure that a token cannot be created, modified or replayed by the attacker, ensuring that a basic block can only execute if another one called it.

#### Proof of correctness

A formal proof of correctness is provided in Appendix A. This proof relies on an induction to create a trace of the ProVerif process resulting from the translation algorithm in which the set of terms known by the attacker is a superset of the set of terms known by the attacker in the SysML-Sec model. The proof of completeness would be more difficult and is left for future work.

To conclude, note that in the specific case where ProVerif is used to verify the confidentiality property on the applied pi-calculus specification, it would also be interesting to study when the translation results in a specification where ProVerif can not prove the expected property. Indeed, the algorithm could be correct and complete but could generate specifications that are especially difficult for ProVerif to handle.

### 4.3.3 Backtracing

Providing a clear and visual feedback to the designer is one of the key point in integrating security verification during the modeling phase of a design. Indeed, the purpose of early security verification is twofold: it helps the designer to model parts of the design based on the properties previously proved and it gives a way to better discover and understand design flaws in order to debug them.

#### 4.3.3.1 ProVerif results and ProVerif Output

When queried about a property, ProVerif tries to prove it by finding all possible *execution traces* that would lead to a violation of this property in an *approximated model*. This approximated model—which is needed since proving secrecy property in the Dolev-Yao model has been proved to be undecidable in the general case [11, 98]—is constructed so that each possible trace on the *real* model produces a possible trace in the approximated model. As such, ProVerif can issue three types of results (given for secrecy here):

- Property is **true**. ProVerif did not find any trace leading to a violation of the property in the approximated model. Since the approximation is sound, this means that the property is true also on the real model.

- Property is **false**. ProVerif has found a trace on the approximated design and has managed to construct a corresponding trace on the real model. The trace found is provided with the result by ProVerif.
- Property **cannot be proved**. ProVerif has found a trace on the approximated design but this trace did not match a valid trace on the real model. In this case, ProVerif is not able to conclude but the trace on the approximated model is output so that the designer can decide whether this matches a valid trace or not.

We keep these three possible results and make them available to the designer through the TTool interface.

#### 4.3.3.2 Verification results on models

In order to enable the designer to simultaneously see the results of the previous verification and accordingly continue modeling, verification results are displayed on the diagrams that are build by the designer. Results for the reachability, confidentiality and authenticity properties are displayed on the block and state machine diagrams in the form of green (when property is true) or red (when property is false) locks.

#### 4.3.3.3 Reconstructing traces

Also, in order to ease debugging and when it is available, the designer is provided with a trace that shows why the property is true (for instance how a state is reachable) or false (how a secret can be disclosed). This trace is automatically constructed based on the trace issued by ProVerif and displayed as a sequence diagram. As such, the trace presents the messages exchanged by the participants (all blocks and the attacker) and the states that each block goes through.

### 4.3.4 Model and verification of a key exchange protocol

To illustrate the method proposed in the previous sections, we propose to model and verify the key exchange protocol presented in Section 3.3.2.3. We refer the reader to this section for a detailed description of the protocol. To summarize, the protocol allows a *Peripheral* and an application (named *DriverEnclave*) to share a secret key that will enable them to communicate securely on an untrusted platform (the *Prover*). To do this, they rely on the Intel SGX architecture (modeled by the *QuotingEnclave* component) and on an external trusted third party named *Verifier*.

#### Model

The diagrams used to model this protocol are shown in the following figures. Figure 4.11 shows the block diagram which describes the components of the protocol. Figures 4.12, 4.13, 4.14 and 4.15 show the state machine diagrams of the blocks in the key exchange protocol.

#### Verification results

The result of the verification of the confidentiality of a message exchanged with the established key is shown on Figure 4.16. We can see on the block diagram the green lock which shows that the *secret* attribute has been proved to be confidential. Figure 4.17 shows a trace that leads to the state `nonceVerify` of the state machine of `DriverEnclave`.

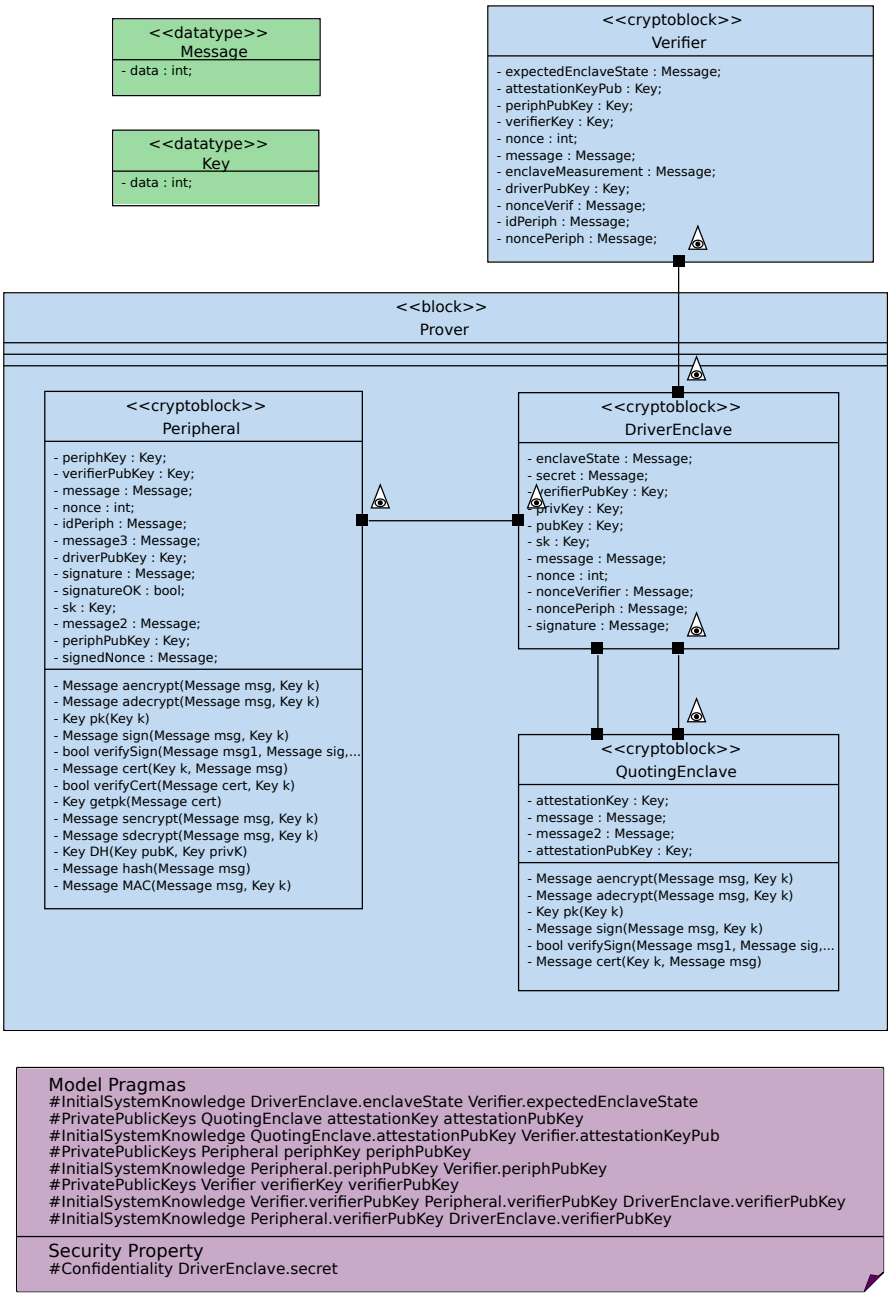



Figure 4.11 – Block diagram describing the actors of the key exchange protocol in the SysML-Sec methodology.

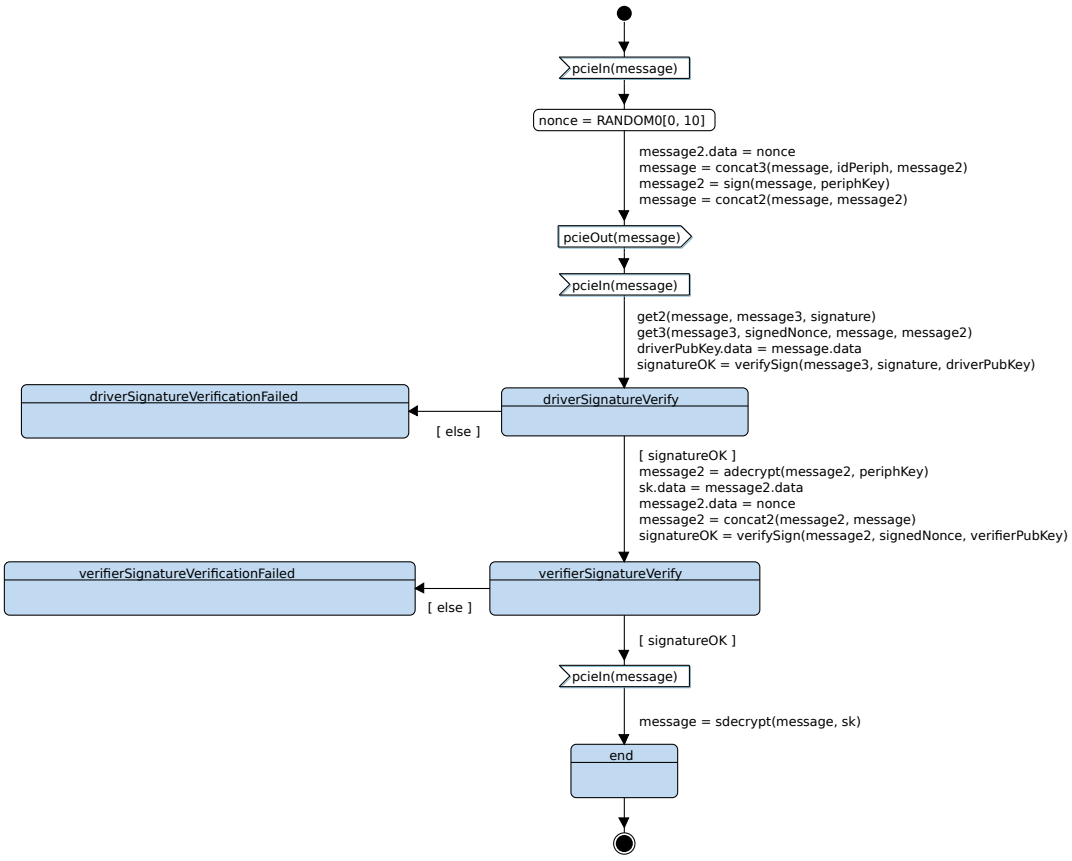



Figure 4.12 – State machine of Peripheral.

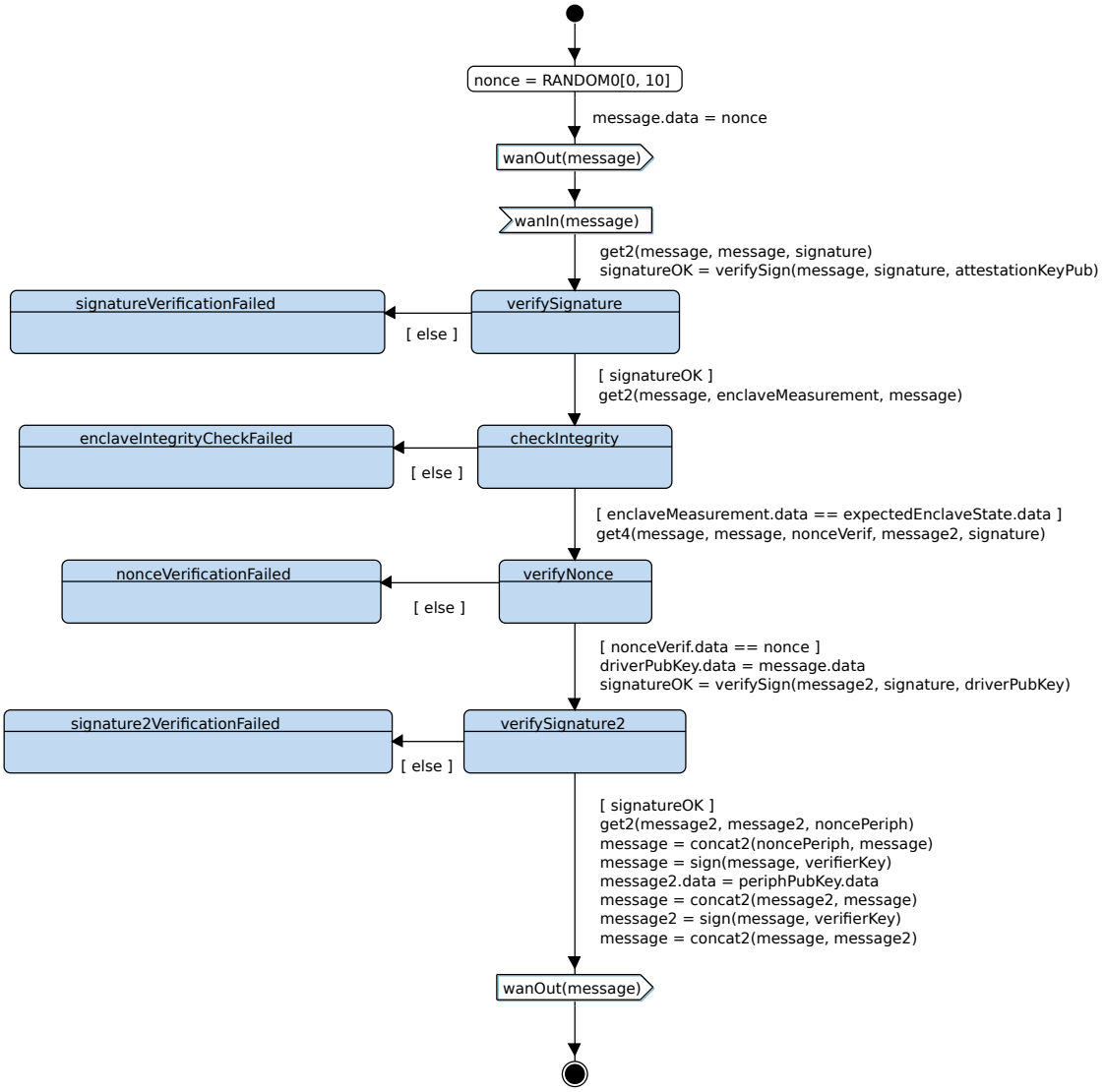



Figure 4.13 – State machine of Verifier.

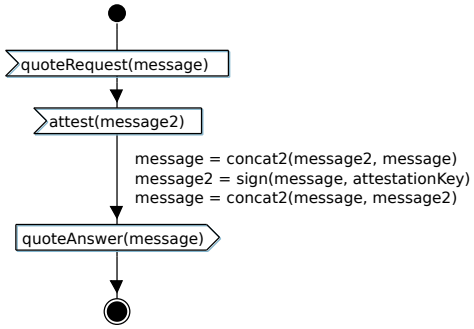



Figure 4.14 – State machine of QuotingEnclave.

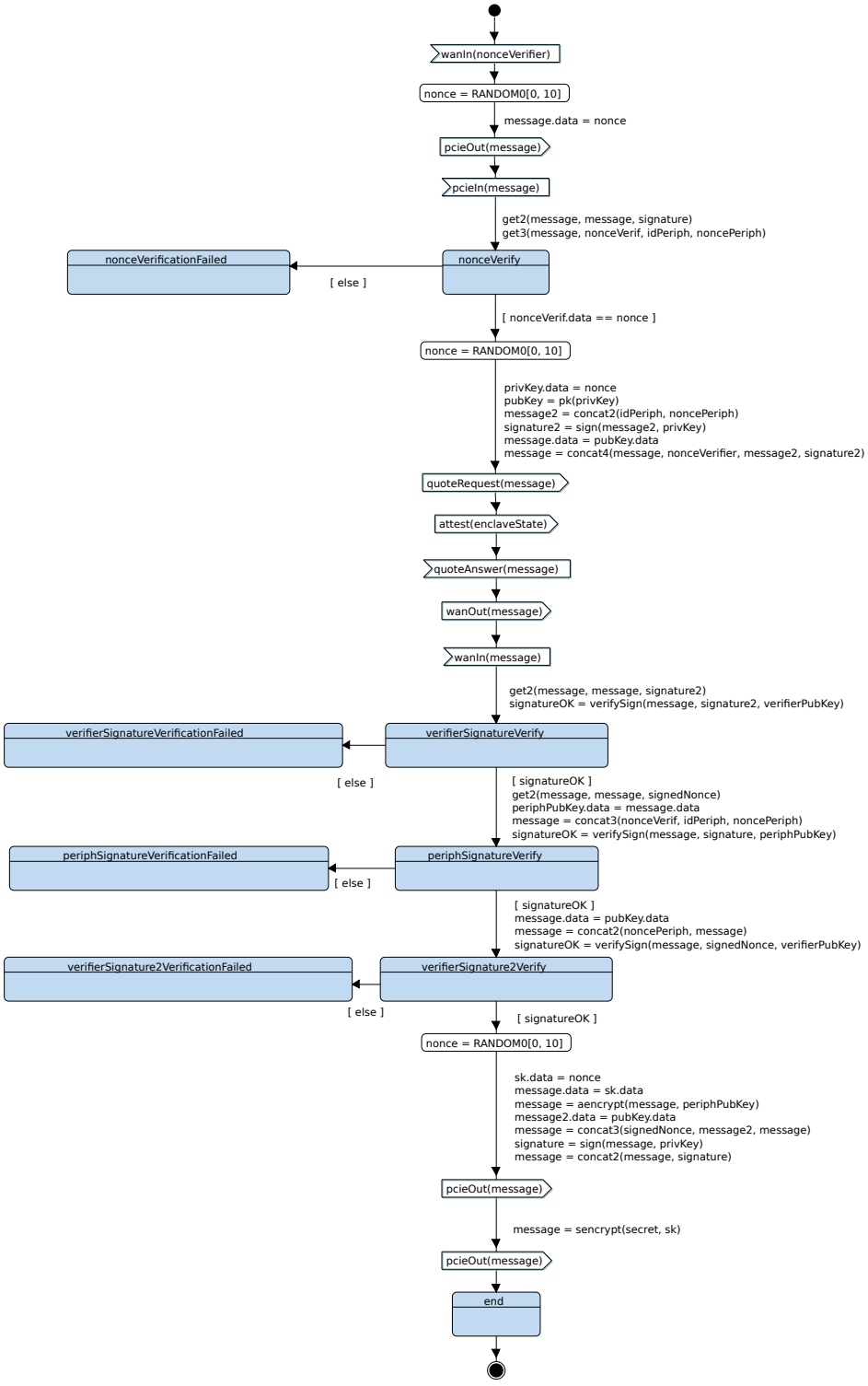



Figure 4.15 – State machine of DriverEnclave.

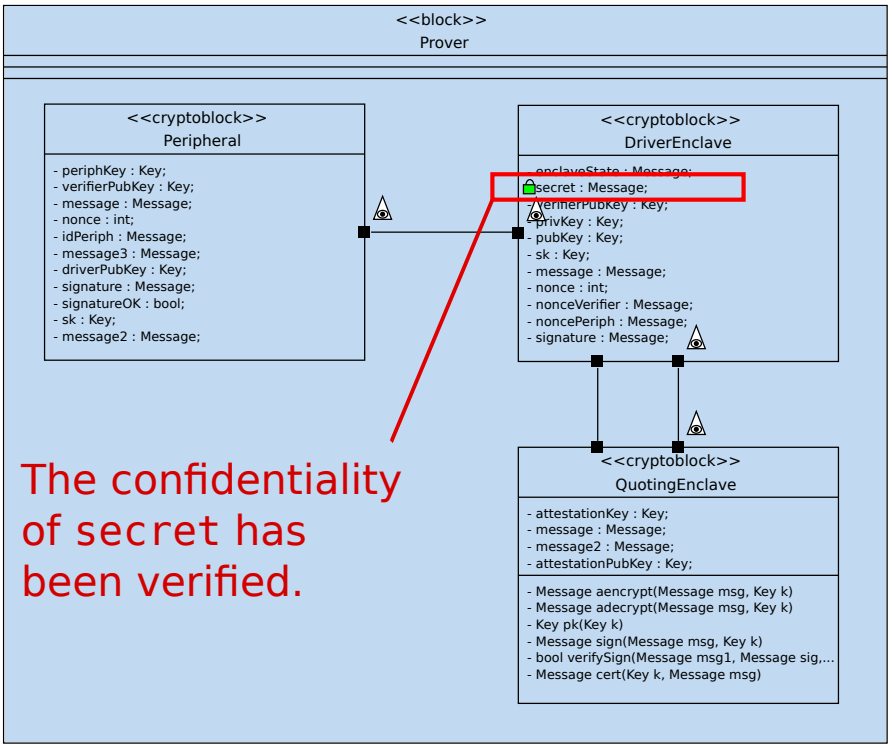



Figure 4.16 – Block diagram with verification result.

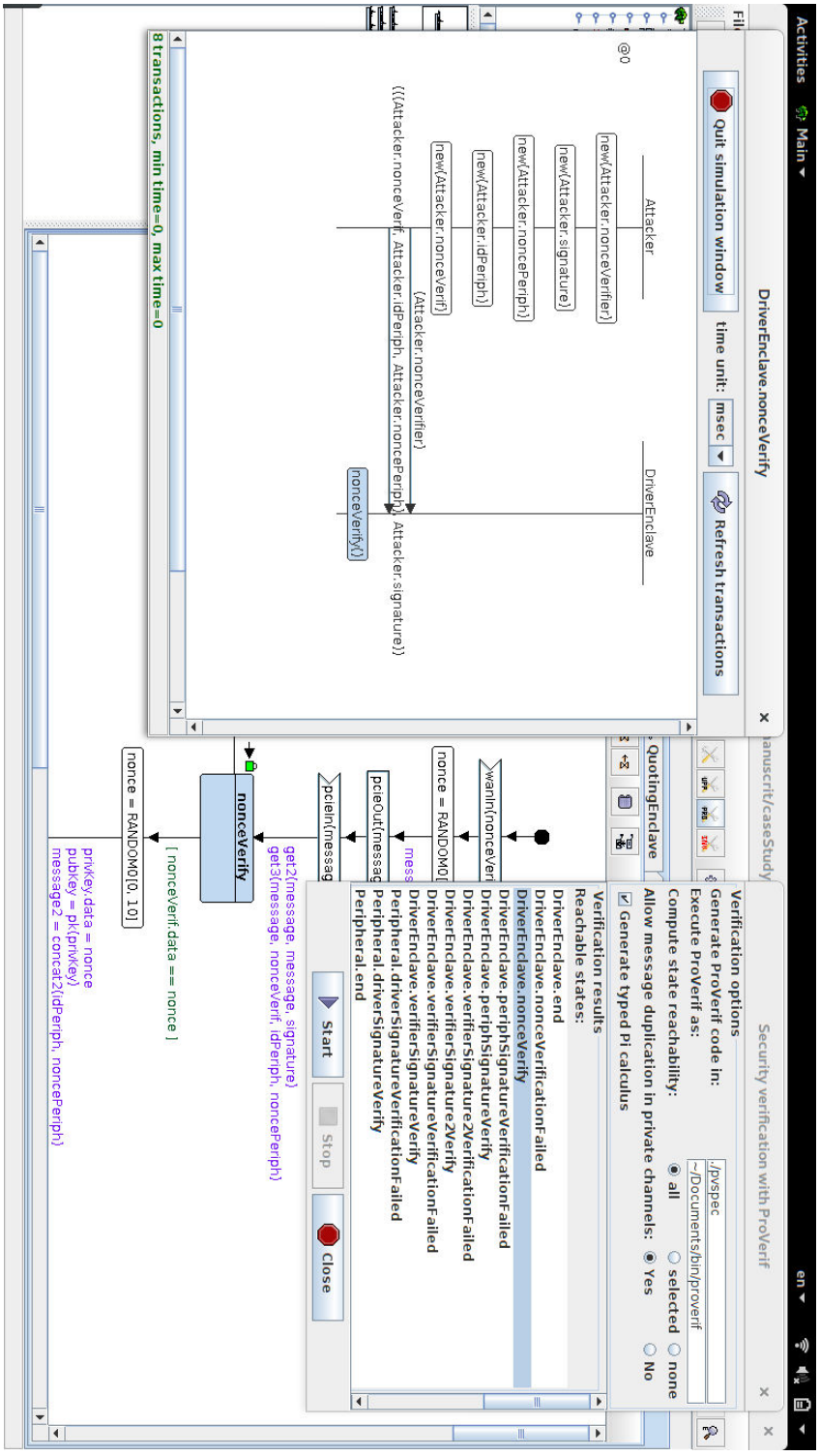



Figure 4.17 – Trace proving the reachability of the state `nonceVerify`.

## 4.4 Security analysis during partitioning

While the implementation details ultimately determine the security of a system, taking security into account in early development phases, without constructing the entire software/system design, would prevent costly late-stage reworking of designs that present vulnerabilities due to early partitioning choices. Even if actual security algorithms will be more complex, basic security primitives and relative computation complexities should be accurate and provide useful feedback to designers when selecting a mapping.

Adding security semantics—and even more, proving security properties—during the partitioning phase is a difficult challenge. On the one hand, the ideal security analysis would take into account every single detail of the system. On the other hand, the designer needs to quickly describe her system for efficient design space exploration, which requires providing an abstract description of the system. In [165], we have provided a security semantic that does not require the designer to deal with implementation details that are irrelevant to partitioning. For example, we are not interested during partitioning in the implementation of encryption algorithms; we only need to consider parameters that will affect the partitioning choice (satisfaction of security properties, execution time, bus load, etc.).

We proposed to add high-level security artifacts to the traditional HW/SW partitioning methodology of the Y-chart approach [146]. Indeed, manually modeling security enables to evaluate its impact in terms of overhead or calculation complexity and security.

### Modeling of security mechanisms

In order to allow the security verifier to track data encryption elements, we introduce *Cryptographic Configurations* (upper left part of Figure 4.2). Cryptographic configurations are graphical elements of activity diagrams (shown on Figure 4.18) that model the fact that a communication is protected by cryptographic primitives. The designer must give a brief description of the cryptographic primitive and define:

- its type (encryption, hash, MAC),
- cryptographic material used by the primitive (optional),
- the performance cost induced by the primitive (computational complexity, overhead), and
- cryptographic material exchange with this configuration (optional).

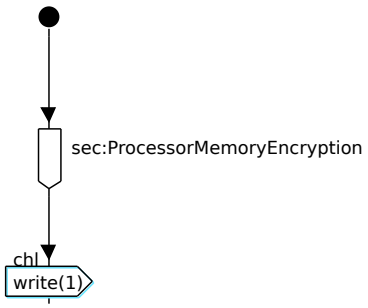



Figure 4.18 – Cryptographic Configuration.

The performance parameters allow us to model the impact of security mechanisms on performance when evaluating candidate mapping. Cryptographic Configurations can also secure other

cryptographic material such as keys. For example, we can model key distribution, where one task forges a key for Symmetric Encryption, and sends it encrypted with another task's public key. The receiving task can then decrypt the message with its own private key to recover the shared key.

The security of the hardware architecture itself is modeled to depict the assumed attacker capabilities. Any of the communication nodes can be tagged by the user as *secure*—which corresponds in our context to the impossibility for an attacker to access it. For example, a Wifi connection could be modeled insecure and all data broadcast on it would be available to an attacker. While internal buses could be modeled as secure.

To send encrypted data along a channel, the designer must create a Cryptographic Configuration, tag all channels along which encrypted data is sent, and finally recover the original data with the Decryption operator. All the keys must be mapped to memories before security verification, and the toolkit warns a designer if access to a key implies transmission across a non-private channel.

#### Security properties

As for security verification during the software design phase presented in the previous section, the properties we are interested in are: reachability, confidentiality and weak and strong authenticity. Since the partitioning phase abstracts away the content of the messages exchanged through channels, these properties don't concern specific values but they are verified on an abstract exchange: a correct implementation of the cryptographic mechanisms modeled by the cryptographic configuration would guarantee that these properties are verified on any value of this exchange.

#### Automated security generation

While Cryptographic Configurations can be manually handled by the designer, it is also possible to automatically generate these security elements. Based on security requirements provided by the designer, our toolkit automatically generates Cryptographic Configurations for each channel whose data must be secure. Currently, we support automatic preservation of confidentiality (protecting against leakage of sensitive data) and strong authenticity (protecting against tampering and replay). When both need to be guaranteed, Cryptographic Configurations are generated so that sensitive data will be concatenated with a nonce and then encrypted before being transmitted across an insecure channel. This automated encryption adds a basic estimation of security, which the designer can later modify.

#### From partitioning diagrams to state machine diagrams

Once a complete application modeling has been mapped to an architecture, a ProVerif specification can be generated from the partitioning model. We leverage the translation presented in the previous section by first converting our partitioning model into an equivalent software model representation, and use the existing translation process to generate a ProVerif model. Each task is first translated to a SysML-Sec block in the block diagram. Each communication path between tasks—mapped to a set of buses, bridges, memories—is translated into an abstract software-model channel. If a path is mapped to a set of communication nodes that are all tagged as *secure*, the resulting software-model channel will be *private*. Otherwise, a *public* channel is used. Most of the components of the activity diagrams are translated straightforwardly to their counterparts in the state machine diagrams. In particular, cryptographic configurations are replaced by their specified cryptographic primitives and concrete values are exchanged through the

channels. An overview of this verification method is given in Figure 4.19.

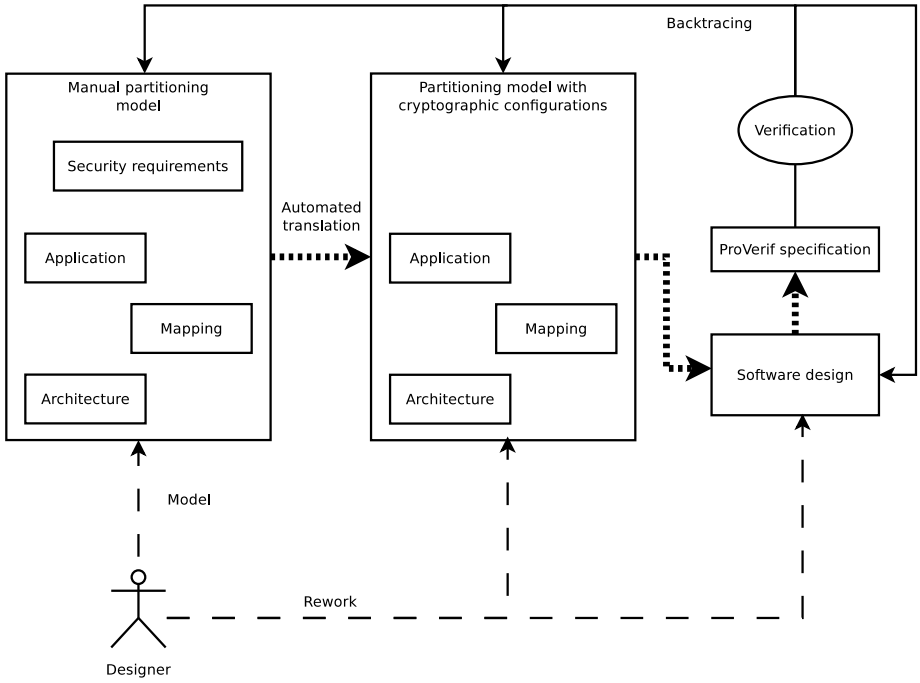



Figure 4.19 – Verification method from partitioning models.

## 4.5 Conclusion

High-level description of systems in early stages of embedded system design can benefit from formal verification as early design choices may lead to hard-to-patch vulnerabilities. During these design stages, the designers need to receive fast feedback on the security impact of their modifications. Automated verification is able to provide such a fast feedback. However, formal verification often requires advanced specialized skills to model a system, express a property, help the verification algorithm and interpret its results. By proposing an automated translation from a modeling language targeting embedded system design to a formal specification suitable for automated verification, we made a step toward providing a press-button approach to formal verification.

Naturally, the verification method proposed here comes with a price: by automatically handling complex design artifacts, the translation algorithm introduces artifacts that are potential sources of incompleteness. This is a classical trade-off where the less human interaction, the harder it is for the algorithm to conclude.

Note that the methods presented in this chapter could be applied to models of system consisting of software and hardware components communicating through well-defined channels. The software components are defined by using operations that are implemented by general-purpose processors. Formally verifying designs where hardware and software are more tightly coupled would be impossible at this level of abstraction and need to be discussed in the next chapter.

# Chapter 5

# Formal verification of tightly coupled hardware and software

In the previous chapter, we were interested in proving security properties on high-level models of a system. While this approach can greatly help in designing secure embedded systems, it fails to uncover security vulnerabilities introduced by lower-level implementations. In this chapter, we will discuss how formal verification—and in particular formal verification of security properties—can be applied to relatively low-level models of systems that express their hardware and software components in different languages.

The need to describe a custom hardware component or architecture is common to most of the embedded systems. We will argue in Section 5.1 that the methods to formally verify such systems greatly differ depending on how tightly the hardware and software components are coupled. Then, we will propose a method to formally verify a system whose hardware and software are tightly coupled and which allows for some hardware customization. In particular, we focus on verifying hardware-assisted security solutions such as memory bus protections [49, 78, 120] (presented in Section 2.1.2.1), hardware-assisted control flow integrity [67,84,85] (presented in Section 2.1.3.1) or software attestation [100, 198] (presented in Section 2.1.3.2) as they often rely on a tight cooperation between a customized hardware and the software running on it.

## 5.1 A problem of interaction

### 5.1.1 Expectations of a verification methodology

In order to guide our discussion and evaluate methods and tools, we list here the properties that one would expect from a formal environment when assessing the security of a hardware/software co-design.

#### 5.1.1.1 Security-aware expressiveness

Software has been steadily increasing in term of quantity and complexity in embedded systems and is still undergoing considerable growth. Implementing critical functions in software may induce bugs or security flaws by increasing the attack surface and thus motivates the need to find solutions that guarantee security properties, such as control-flow integrity or code integrity, on any software. To target such global security properties new solutions often rely on specific

hardware. The global security of the system thus depends on the security of the solution implemented as a tight mix of hardware and software. An efficient methodology dealing with this kind of designs should enable to express properties and give back results in a security-oriented meaningful way.

#### Natural expression of security properties

First, designers would like to express the property they want to prove on their design as directly as possible. Translating the expected property into a combination of properties manageable by the solution but whose meanings are hard to grasp—typically formulas in conjunctive normal form or CTL formulas—is a source of errors. The verifier would thus be more interested by solutions that can naturally handle properties such as the confidentiality of a software variable or the propagation of a taint.

#### Attacker model

On the other side, expressing the capabilities of an attacker should be equally straight forward. It may be by using the Dolev-Yao model [94], or by tainting inputs that the attacker is able to control, for instance. The attacker model is normally coherent with properties the method is able to handle since the latter should be checked against the former, but a tool could also provide an automatic translation of abstract attacker models into low-level logic that the verification engine can handle. For instance in the algorithm presented in [35], the Dolev-Yao attacker capabilities are translated to a set of Horn clauses.

#### Reconstruction of Traces

When the analysis tool determines that the required property may be violated, the designer must correct the erroneous part. The verifier should thus be able to rely on the feedback of the analysis framework to target the part of the design that would need to be redesigned. Since precisely and automatically determining the erroneous part of the design is currently impossible, a compromise often found is to provide the user with a trace summarizing the steps that lead to a state in which the property is violated. On the other side, returning the unsatisfiable core of a CNF formula would be of little interest for the designer.

### 5.1.1.2 Soundness of the verification algorithm

Many hardware/software systems (such as the ones presented in section 2.1.3) provide core features that are critical either for the proper functioning of the system (such as peripheral management), or for its security (e.g., access control, cryptographic primitives). These modules require strong safety and security guarantees that only formal verification is able to provide. Software analysis often has to deal with very large programs, which rules complete verification out. Here, we are concerned with smaller programs that hopefully enable us to mathematically prove that they are correct with respect to the features they were supposed to provide. Approximations are thus considered only as far as they do not affect the soundness of the verification.

### 5.1.1.3 Easy adaptation to hardware modifications

When designing systems composed of hardware and software components, the designer needs to get fast feedback concerning the effect of hardware modifications. Most verifications of software targeting embedded systems rely on a manual expression of the hardware model [88,241]. While finding a generic method that would deal with any hardware description may seem too optimistic,

we believe that analysis of embedded systems or systems on chip would benefit from some modularity in terms of hardware models. We will thus evaluate the ability of each method to verify a system whose hardware is provided as a low-level description.

### 5.1.2 Successive verification

The traditional approach to hardware/software validation is to express a formal model of the hardware and use it during the verification of the software [106]. The hardware may also be proved equivalent to the model, thus ensuring the overall security of the system. We will call these methods *successive verification* since the verification takes place in two steps. The two steps are explicitly or implicitly linked by the designer who provides a formal semantic at the junction of hardware and software.

For designs where hardware and software are tightly coupled, it may however be difficult to find an abstraction that would both enable the hardware to be verified, and require a manageable modification of a generic software analysis framework to integrate the specificities of the hardware. We discuss here how these two worlds could be integrated.

#### 5.1.2.1 Expression of the hardware model

We target here designs where hardware and software must be checked together to ensure system-level properties. There are mainly two classes of such designs: either the hardware was customized in order to change the way the software was executed, or the hardware to verify does not affect the core processor but is a peripheral (such as an MMU or a sensor), and the software part is handling the communication with this peripheral. In the first case, the software analysis tool—which assumes a particular semantic of the instruction set and the execution engine—would need to be modified to take into account the specificities of the hardware. In the second case, a common formal model could be found, and the hardware and the software could be checked separately against this model. In this case, the hardware specificities do not question the *software* abstraction made by traditional verification tools.

To prove that the hardware model—either when it is integrated into the software analysis framework, or when it is common to the software model—is a correct abstraction of the hardware, traditional verification of hardware designs could be applied. This verification is mostly done either by equivalence checking or by model checking. Many industrial and academic tools exist for this purpose such as Vis [45], NuSMV [69], Incisive[1] or Formality.[2]

#### 5.1.2.2 Verification of low-level software

Since we are here interested in both software and architectural vulnerabilities, we would like to take the compiler out of the trusted computing base. This is particularly true for security-critical features—such as MMU management or cryptographic primitives—that are typically directly implemented as machine code. Therefore, we are mainly interested in **software verification tools that can take machine code as input**.

Higher-level concepts such as arrays, objects, functions, or types are not available when using assembly code. Losing such concepts means that we can't benefit from the semantic of coherent objects that the designer manually provided. For instance, it is simpler for the analysis to replace calls to a function by the formal expression that links the output of the function to its inputs and to prove that the function is indeed equivalent to this formal expression, than to analyze the

[1] `http://www.cadence.com/products/fv/enterprise_simulator/pages/default.aspx`
[2] `http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/Formality.aspx`

function each time it is called in each context. However, we believe working with assembly code is more representative of the attack scenarios we want to prevent (shellcodes, ROP) and of the software we want to verify.

In order to verify software at the assembly level, we ideally need a formal semantic of the instruction set. Such semantics are rare in practice, but progress has been made lately in this direction [88,106]. Once this formal model has been found, traditional software analysis methods like model checking and bounded model checking or symbolic execution may be applied.

#### 5.1.2.3 Dealing with hardware customization

Since a designer may want to see the impact of a hardware modification as part of the design process, the amount of work needed to include this modification in the software verification framework and to prove the model to be a refinement of the modified hardware should not be prohibitive.

### 5.1.3 Unified verification

As disjoint verification of hardware and software naturally suffers from the considerable manual effort needed for finding a *good* abstraction that could both be proved to be a refinement of the hardware and be used as a base for verifying the software, some research work has been done to verify hardware/software systems as a whole [132, 157].

Similarly to successive verification of hardware and software, the methodology here also differs depending on how tightly the two are coupled.

#### 5.1.3.1 Hardware and software as tasks

For hardware/software systems where the processor is not modified and the verification effort should focus on some parts of the design that communicate with software through the use of a simple interface (such as memory mapped port or signals), the duality hardware/software is not relevant anymore. Conceptually, both hardware and software parts provide abstract functionalities that are described in two unrelated languages. It becomes thus possible to verify both at the same time, but still keep them distinct. As presented in Section 2.2.2.3, a good example of such verification on loosely coupled hardware and software can be found in [157].

#### 5.1.3.2 Tightly coupled hardware and software

Hardware-based protection against software vulnerabilities often affects how the processor interprets machine code, by detecting policy violation as in [100], or by adding new instructions as in [67] for instance. In such designs, the processor itself is part of the verification target, and thus, simultaneous verification of disjoint hardware and software cannot be done as in the previous section.

##### Including software as part of the hardware representation

In such cases, we are trying to verify a customized processor implemented in hardware for a particular piece of software. However, we want to verify both as a whole, that is to say we do not want to create a formal model of the processor—which would be an intricate manual process. We cannot prove the software with respect to a generic processor model since our processor is customized and we have no abstract model for our specific hardware. The abstract concept of *software* is thus unusable and program instructions must be considered for analysis in their

Table 5.1 – Comparison of verification tools.

| Tool | Security-oriented | Type of prop. | Soundness | HW/SW modelling |
|---|---|---|---|---|
| NuSMV | no | CTL,LTL | sound | abstract model |
| UPPAAL | no | TCTL | sound | abstract model |
| BLAST | yes | safety | sound | software |
| Vis | no | CTL | sound | hardware |
| KLEE, FIE | yes | safety | sound | software |
| S2E | yes | safety | sound | software |
| ZeBu, Seamless, SoC Designer Plus | no | — | unsound | HW/SW |

true, concrete format: binary data. Concretely, this means filling the memory in the hardware representation with the program in binary format and verifying the combined system as a whole.

#### 5.1.3.3 Proving properties on the whole design

Once software has been integrated into the hardware representation of the design, traditional hardware verification tools could be used to prove the required property. However, some parts of the design are controlled by the attacker, typically some part of the memory corresponding to the procedure arguments could take any value. The value of these bits will affect how the program executes. For the analysis, this means that a huge number of states will have to be explored. Even addressing this problem with symbolic model checking would most likely be difficult. Indeed, symbolic model checking attempts to reduce the state space by finding *good* abstractions that would lead to a reasonably small model. In our case, such abstractions would need to relate software-level objects together, which would have disappeared in the combined HW/SW system. For instance, let us say we are trying to verify a piece of software where a good abstraction—one that would make the property provable on the abstract model—would be *"the length of string $s$ is smaller than the value of variable $v$"*. The meaning of $s$ and $v$, however, is no longer present at the hardware level, and automatically reconstructing these objects, by predicate abstraction [70] for instance, would be difficult. Indeed, there is no hardware concept of what a string is, nor what smaller means. For this reason such a verification scheme would probably be limited to very small programs.

### 5.1.4 Adequacy of existing tools

We summarize in Table 5.1 the adequacy of academic and industrial tools to the requirements described in Section 5.1.1. It is interesting to note that some tools can take abstract models as input and thus can analyze both hardware and software. However, using these tools requires the designer to first manually provide an abstract model corresponding to the system to verify. Also note that ZeBu, Seamless, and SoC Designer Plus were included even if they are emulators and not formal verification tools.

Table 5.2 – A ProVerif process and the corresponding Horn clauses.

| ProVerif Process | Set of Horn Clauses |
|---|---|
| `process`<br>`  in (ch, a: bitstring);`<br>`  out (ch, f(a))` | $mess(ch, a) \rightarrow mess(ch, f(a))$<br>and $attacker(a) \rightarrow attacker(f(a))$<br>if $ch$ is public. |

## 5.2 SMASHUP

In our approach to HW/SW verification, we looked for a method which had some properties of the ideal method we described earlier and which could be adapted easily to designs with customized hardware components. That is to say we were searching a method that could:

- Model a generic processor and instruction set.
- Allow simple modeling of hardware customization.
- Model an attacker and prove security-oriented properties.
- Automatically produce a meaningful result, be it a clear answer if the property is proved to be true, or a trace if the property can be violated.

### 5.2.1 ProVerif deductive algorithm

As presented in Section 4.3.1.1, ProVerif [38] is a tool for analyzing protocols. It focuses on security protocols but the generic language (applied pi-calculus) used for ProVerif specifications and the simple reasoning of the tool, based on Horn clauses make it a good candidate for a wide variety of applications [204].

#### 5.2.1.1 Motivations for using ProVerif

Our requirements led us to search for a tool that would work with basic and generic logic and would target security properties. As ProVerif answered these needs, we chose it despite the fact that it was originally designed for a different field of applications.

**An interesting attacker model**

A lot of security properties can be expressed as the secrecy of a particular asset, and such a property can be natively represented in ProVerif. The tool also enables to query more complex properties such as authentication or observational equivalence.

These properties are checked against an attacker whose capabilities follow the Dolev-Yao model. These kind of capabilities are also interesting in many designs, where the various hardware and software components can be seen as participating in a protocol and the user-controlled software on the device has full access to the abstract channel they are using.

#### A simple reasoning

ProVerif takes as input a description of a protocol in a restricted version of the applied pi-calculus language. This description is internally translated into Horn clauses that ProVerif uses for reasoning. Horn clauses are logical formulas of the form:

$$\bigwedge_i p_i \rightarrow q \qquad \text{or} \qquad q \qquad \text{or} \qquad \bigwedge_i p_i \rightarrow false \tag{5.1}$$

where $p_i$, $q$ are positive literals. The second formula corresponds to the case where there is no premise and the third to the case where the disjunctive normal form only contains negative literals. This simple formulation makes it possible to model how each assembly instruction impacts the state of the system depending on the environment, and thus, allows for easy modeling of the effect of hardware modification on software execution.

#### Trace reconstruction

Another feature of interest in ProVerif is its ability to reconstruct a trace when the queried property is violated. This trace is given as a succession of actions performed by the attacker that eventually lead to a violation of the property. The process of reconstructing this trace may fail (as explained in [9]) due to the approximations done when translating processes in applied pi-calculus into Horn clauses. However as explained earlier, this incompleteness is unavoidable as the global problem is undecidable.

### 5.2.1.2 ProVerif solving algorithm

Our ambition was to prove properties on a relatively small piece of software running on a custom hardware. Since formally proving the property on a model of the software would mean exploring the entire state space, and sticking to a realistic model would limit the possibilities for abstraction, we assumed that our methodology would not scale well. Even though, we believe it may prove useful for small, central, security-critical software, as it is the case for some hardware-assisted security solutions such as the architecture proposed in chapter 3.

We briefly present here how ProVerif is able to reason about the specified protocols. This will help us explain how we exploit it in our method and compare the performance with more traditional techniques.

#### Horn clauses and predicates

Protocols that need to be verified by ProVerif are described as multiple processes that communicate between each other through private or public channels. The attacker can see anything that goes through public channels, intercept messages, create new ones, and send them on public channels.

The fact that the attacker knows about the message $m$ is modeled as a predicate $attacker(m)$. The fact that a message $m$ can be sent on channel $ch$ is modeled as a predicate $mess(ch, m)$. As stated earlier, ProVerif works with Horn clauses so, for instance, the abilities of the attacker regarding channels are:

$$\begin{array}{lrcl} & mess(ch,m) \wedge attacker(ch) & \rightarrow & attacker(m) \\ \text{and} & attacker(ch) \wedge attacker(m) & \rightarrow & mess(ch,m). \end{array} \tag{5.2}$$

Processes are also translated into Horn clauses. For instance a basic process and its translation are presented in Table 5.2. Note that both express the fact that if the attacker has knowledge

of $a$, then she can acquire knowledge of $f(a)$. As it will be explained in the next section, this simple mechanism enables us to model an instruction-accurate version of a processor.

#### Clauses unification

Once the protocol has been translated into Horn clauses, these clauses are combined to derive the total knowledge of the attacker. If the required property is violated during the process, a trace is computed based on the clauses that have been unified to lead to the violation.

In our method, the way the clauses are unified will determine how the state space of the program is explored. Thus more information about the resolution process of ProVerif—as explained in [35]—is going to be exposed.

The idea behind clause unification is to progressively expand the knowledge of the attacker. Let us say we have two clauses:

$$\begin{array}{rrcl} & attacker(m) & \rightarrow & attacker(f(m)) \\ \text{and} & attacker(f(m)) & \rightarrow & attacker(g(m)). \end{array} \tag{5.3}$$

Unifying these two clauses is interesting since it will result in: $attacker(m) \rightarrow attacker(g(m))$, which means that if the attacker has knowledge of any message $m$, then $g(m)$ can also be known. By default, ProVerif considers that unifying two clauses is interesting when all the premises of the first clause are of the form $attacker(x)$ where $x$ is a variable and when the premise of the second clause that can be unified with the conclusion of the first—$attacker(f(m))$ in our previous example—is not of the form $attacker(x)$. It means that it favors unifications that reduce the number of premises that are not of the form $attacker(x)$.

By unifying clauses like that, ProVerif eventually reaches a fixed point where no new clause can be generated. If the required property is the secrecy of a variable $x$, and eventually no clause of the form $attacker(x)$ has been derived, this is a proof that the attacker can't learn the value of $x$.

### 5.2.2 Using ProVerif for simple symbolic execution

We will now present the method we proposed to verify hardware/software systems. This method enables us to verify systems described as a hardware part (in a language that will be presented) and a software part described as assembly code.

We show first how we automated the translation of assembly code into a ProVerif model, and then, how hardware customization was integrated into the model. Finally, we demonstrate the solving process performed by ProVerif and relate it to a more classical software analysis method.

#### 5.2.2.1 The software part

We model our software in an instruction-accurate way: We express the impact of each instruction on the state of the system. This semantic enables us to consider attack scenarios and software designs that are realistic, especially on low-level code: dynamic control-flow graph with indirect jumps for instance. As we were specifically targeting embedded system verification, we used a software implementation described in an assembly language for the MSP430 architure which is common in the embedded system world (and for which soft-cores are freely available [3]). We thus modeled a subset of the MSP430 assembly language, composed only of basic instructions. For a more general approach, it would be better to use an intermediate representation such as

[3] `https://opencores.org/project,openmsp430`

REIL [97] or the BAP intermediate language [48] and use the already existing front ends to compile either assembly code or binary code to this intermediate representation.

An instruction at virtual address $i$ is modeled as a process, which is translated by ProVerif into a Horn clause: $state(i, R, MEM) \rightarrow state(PC', R', MEM')$, in which $PC'$ is a program counter, $R$ and $R'$ are states of the registers, and $MEM$ and $MEM'$ states of the memory. $PC'$, $R'$, and $MEM'$ are expressed as functions of $R$ and $MEM$ and model the effect of the instruction at address $i$ on the state of the memory and registers—for instance $PC' = MEM[R[3]]$.

The $state(PC, R, MEM)$ predicate here would mean that a state of the system is accessible where the program counter is $PC$, the registers' values are $R$, and the memory is in state $MEM$. This predicate is obviously not defined in ProVerif and we must model it. We could do this by using a private channel: each message $(PC, R, MEM)$ sent on the private channel *privch* would mean that the state $(PC, R, MEM)$ is accessible. The effect of an instruction at address $i$ would thus be: $mess(privch, (i, R, MEM)) \rightarrow mess(privch, (PC', R', MEM'))$. However, private channels behave differently with respect to trace reconstruction. For instance, when trying to reconstruct a trace, ProVerif will only allow sending messages on a private channel if a process is ready to read the message on this channel. Therefore, we chose to use an equivalent approach with public channels: $mess(ch, f(i, R, MEM)) \rightarrow mess(ch, f(PC', R', MEM'))$. Where $f$ and its inverse $un\_f$ are private functions (with no explicit definitions) that guarantee that the fact $attacker(f(PC, R, MEM))$—which means that the state $(PC, R, MEM)$ is reachable—does not lead to $attacker(R)$ or $attacker(MEM)$, and reciprocally that the attacker cannot create $f(PC, R, MEM)$ with any $PC$, $R$ and $MEM$. Eventually the corresponding process in applied pi-calculus is:

```
process
    in (ch, state: bitstring);
    let (PC: int, R: registers, MEM: memory)
            = un_f(state) in

    if PC=i then

    out (ch, f(PC', R', MEM'))
```

This process only models one instruction. To model the entire program, we created one process per instruction and replicated it—using the ProVerif operator `!`—so that the instruction could be invoked many times (in case of loops for instance). We wrote an open-source Python program named SMASHUP[4] (*Simple Modeling and Attestation of Software and Hardware Using ProVerif*) that automates the process of translating MSP430 assembly code into a set of such processes.

#### 5.2.2.2 Parallel with symbolic execution

The algorithmic efficiency of ProVerif resides in its ability to derive the complete knowledge of the attacker with as few clause unifications as possible. The policy used to choose which clauses to unify will guide the exploration of the program in our context. We'll show how this works on a basic example:

```
0       mov.w       #0x0000,    r4
    10:
1       add         r3,         r4
2       sub         #1,         r3
```

[4] Available at `https://gitlab.eurecom.fr/Aishuu/smashup`

```
3       jnz         l0
4       ...
```

As will be explained later, ProVerif has originally no representation for numbers. In order to simplify the presentation in this report, we will here ignore this fact and use them as intuition dictates. We will also only consider the first five registers and no memory to shorten the clauses, use $R2$ as the zero flag (instead of just one bit), and ignore overflows. For the sake of simplicity we will use $state(PC, R)$ as a shortcut for $mess(ch, f(PC, R))$ as was done before. Under these assumptions, the Horn clauses generated for the instructions would be:

$$\begin{aligned}
& state(0,(R1,R2,R3,R4)) \\
& \qquad \rightarrow \quad state(1,(R1,R2,R3,0)) \\
& state(1,(R1,R2,R3,R4)) \\
& \qquad \rightarrow \quad state(2,(R1,R2,R3,R3+R4)) \\
& state(2,(R1,R2,1,R4)) \\
& \qquad \rightarrow \quad state(3,(R1,1,0,R4)) \\
R3 \neq 1 \quad \wedge \quad & state(2,(R1,R2,R3,R4)) \\
& \qquad \rightarrow \quad state(3,(R1,0,R3-1,R4)) \\
& state(3,(R1,0,R3,R4)) \\
& \qquad \rightarrow \quad state(1,(R1,0,R3,R4)) \\
R2 \neq 0 \quad \wedge \quad & state(3,(R1,R2,R3,R4)) \\
& \qquad \rightarrow \quad state(4,(R1,R2,R3,R4)).
\end{aligned} \tag{5.4}$$

If we allow execution of the routine only from the beginning this would add a clause:

$$\begin{aligned}
& attacker(R1) \wedge attacker(R2) \wedge attacker(R3) \\
& \wedge attacker(R4) \rightarrow state(0,(R1,R2,R3,R4)).
\end{aligned} \tag{5.5}$$

As mentioned earlier, the solving algorithm of ProVerif will only unify two clauses if the premises of the first one are all of the form $attacker(x)$. In our context, this means it will start the unification with the clause describing how the attacker could call the routine (the last clause given above). Its conclusion is of the form $state(0, ...)$ so it could only be unified with a clause with a $state(0, ...)$ premise (the first one). Unifying these two clauses will result in:

$$\begin{aligned}
attacker(R1) \quad \wedge \quad attacker(R2) \quad \wedge \quad attacker(R3) \\
\rightarrow \quad state(1,(R1,R2,R3,0)).
\end{aligned} \tag{5.6}$$

Once again, this clause is the only one that could be used for unification so it will be unified with the clause corresponding to instruction 1:

$$\begin{aligned}
attacker(R1) \quad \wedge \quad attacker(R2) \quad \wedge \quad attacker(R3) \\
\rightarrow \quad state(2,(R1,R2,R3,R3)).
\end{aligned} \tag{5.7}$$

Here, this clause could be unified with either of the two clauses corresponding to instruction 2 so exploration will *fork* and follow each of the two branches depending on the value of $R3$:

$$\begin{aligned}
attacker(R1) \quad & \rightarrow \quad state(3,(R1,1,0,1)). \\
attacker(R1) \quad \wedge \quad & attacker(R3) \quad \wedge \quad R3 \neq 0 \\
& \rightarrow \quad state(3,(R1,0,R3-1,R3)).
\end{aligned} \tag{5.8}$$

We could make a parallel between this behavior and symbolic execution: In symbolic execution, input variables that the attacker can control are marked as symbolic and a symbolic

execution engine executes the program, forwarding and constraining symbolic values along the different possible paths. When the execution must split according to the value of a symbolic variable, the constraints on the symbolic value for each path are remembered and two separate instances of the execution engine continue the analysis.

Our method shares some similarities: Variables controlled by the attacker are used without giving them concrete values until a conditional instruction—that has been translated into two clauses—is met. The unification process then follows two different paths where premises have been added that constrain the value of the variable.

#### 5.2.2.3 The hardware part

The instruction-accurate description of software presented above enables us—to some extent—to bring hardware customization into the verification process. There are two ways the verifier can use to describe hardware customizations.

First, we designed SMASHUP in a modular way. Each module would represent a hardware component, such as an interrupt controller or a MMU. Concretely, each module is described as a Python class that extends a common interface. Modules can add predicates to the state of the system (like $PC$, $R$ or $MEM$ described above) that can be used and updated by instructions. The effect of the instructions can also be altered by custom hardware by using hooks in some method of the translation process. For instance, an interrupt controller could add a register to remember whether interrupts are enabled or disabled. Also, a module can add new instructions and provide a corresponding *implementation* for each of them. This implementation describes how the instruction impacts the state of the system and the knowledge of the attacker. We show in Listing 5.1 the python class used to express the impact of an interrupt controller on software verification. The overridden methods `state`, `stateInit` and `processDecl` are hooks used by smashup at different points during the translation of assembly code to a ProVerif specification. In particular, `processDecl` enables the module to add two ProVerif modules (`call_interrupt` and `return_interrupt`) which are used to describe the ability of the attacker to interrupt the routine and to modify its state before resuming the execution. Also, methods are added for two instructions so that the module is queried when the translation algorithm needs to handle these instructions (`EINT` and `DINT`).

```
1  class INT_UNIT (HWModule):
2      def __init__(self):
3          super(INT_UNIT, self).__init__('INT_UNIT')
4
5          self.addInstruction('EINT', self._astEINT)
6          self.addInstruction('DINT', self._astDINT)
7
8          self.dependancies = ['CPU']
9
10     @ifInstantiated
11     def state(self, parser):
12         return 'gie: bit', 'gie'
13
14     @ifInstantiated
15     def stateInit(self, parser):
16         return 'gie: bit', 'gie'
17
18     @ifInstantiated
```

```
19      def processDecl(self, parser, rcount, bitwidth, memrange, stateAll, id):
20          declarations = []
21          stateIn  = [s[1] for s in stateAll]
22          stateOutPlusOne = [(s[0], s[2]) for s in stateAll]
23          stateOut = ['r0, ' + ', '.join(s[1].split(', ')[1:]) if s[0] == 'CPU' \
24                  else s[1] for s in stateOutPlusOne]
25          declarations.append(parser.ASTNode( \
26                  'let call_interrupt =\n' + \
27                  '    in (ch, enc_state: bitstring);\n' + \
28                  '    let (' + ', '.join(stateIn) + ') = dummydec(enc_state) in\n\n' + \
29                  '    if gie = ' + parser.bitrepr[1] + ' then\n\n' + \
30                  '    out (ch, (' + ', '.join(stateOut) + ')).\n'))
31          declarations.append(parser.ASTNode( \
32                  'let return_interrupt =\n' + \
33                  '    in (ch, enc_state: bitstring);\n' + \
34                  '    let ('+', '.join([s.replace(':', '_orig:') for s in stateIn]) + \
35                  ') = dummydec(enc_state) in\n' + \
36                  '    in (ch, (' + ', '.join([ \
37                      ', '.join(s[1].split(', ')[1:]) if s[0] == 'CPU' \
38                      else s[1] for s in stateAll]) + '));\n\n' + \
39                  '    if gie_orig = ' + parser.bitrepr[1] + ' then\n\n' + \
40                  '    out (ch, dummyenc(' + ', '.join([ \
41                      'N(r0_orig), ' + ', '.join(s[1].split(', ')[1:]) if s[0] == 'CPU' \
42                      else s[1] for s in stateOutPlusOne]) + ')).\n'))
43          return ['call_interrupt', 'return_interrupt'], declarations
44
45      def _astEINT(self, parser, args, rcount, bitwidth, memrange, state, id):
46          state = [s[1] for s in state]
47          name = 'EINT'
48          state[id] = parser.bitrepr[1]
49          return (name, parser.ASTNode ('out (ch, dummyenc(' + ', '.join(state) + '))'))
50
51      def _astDINT(self, parser, args, rcount, bitwidth, memrange, state, id):
52          state = [s[1] for s in state]
53          name = 'EINT'
54          state[id] = parser.bitrepr[0]
55          return (name, parser.ASTNode ('out (ch, dummyenc(' + ', '.join(state) + '))'))
```

Listing 5.1 – Python class describing the impact of an interrupt controller on software verification.

Creating new modules enables to capture a hardware behaviour for a customized hardware module. However, it requires the designer to concretely model how this new hardware module will impact software verification. In order to allow hardware customization by non-expert designers, we provide a first set of standard modules (such as an interrupt controller) which enable SMASHUP's users to provide a high-level description of the architecture of the hardware by instantiating different modules. Users would then be able to create modules for specific hardware and deliver them as libraries that could be used by other designers.

### 5.2.3 Example

We illustrate the capability of SMASHUP to find vulnerabilities coming from both software and hardware specificities through an example of a simple system composed of a hardware part and a software part. This example consists of a hardware and a software model that can be downloaded together with the SMASHUP tool. The former one is described in Listing 5.2. It only features an execution core and a standard memory.

```
CPU (
    width: 4,
    rcount: 5
)

MEMORY (
    width: 4,
)

# uncomment the following module to test
# interrupts
# INT_UNIT ()
```

Listing 5.2 – Hardware Description of the System.

The software part described in Listing 5.3 shows that a secret (initially stored in register `r1`) is written in memory a certain amount of time (controlled by `r3`). Then the secret is cleared from all the addresses where it was stored.

```
.section .do_mac.call,"ax"
    mov.w       #0x0000,    r4
l0:
    cmp         r3,         r4
    jeq         l1
    mov.w       r1,         @r4
    add.w       #1,         r4
    jmp         l0
l1:
    mov.w       #0x0000,    r4
    mov.w       #0x0000,    r1
; uncomment the following instruction to test
; integer overflow
;    add.w       #1,         r3
l2:
    cmp         r3,         r4
    jeq         l3
    mov.w       #0x0000,    @r4
    add.w       #1,         r4
    jmp         l2
l3:
    nop
```

Listing 5.3 – Software Description of the System.

With this description, we can use SMASHUP and ProVerif to assert that the secret is not leaked:

```
$ ./smashup.py -o test.pv -s .do_mac.call \
$   --clauses examples/test.s43
$ time proverif test.pv
[...]
Starting query not attacker(secret[])
RESULT not attacker(secret[]) is true.
proverif test.pv  0,22s user 0,00s system
```

The output of ProVerif matches the expected answer: the secret is not leaked. We can then modify the system to model the presence of interruptions. In such case, the secret is leaked when an attacker interrupts the routine just after the secret was written to memory. The memory range that is cleared after the secret has been written to memory can also be extended by adding

a software instruction (line 14) to increase the size of the memory range to be cleared by one unit. In this case, an integer overflow could happen if `r3` contains the maximum integer value and the secret would not be cleared from memory at all. This is confirmed by SMASHUP and ProVerif:

```
$ ./smashup.py -o test.pv -s .do_mac.call \
$   --clauses examples/test.s43
$ time proverif test.pv
[...]
Starting query not attacker(secret[])
[...]
The attacker has the message secret.
A trace has been found.
RESULT not attacker(secret[]) is false.
proverif test.pv  0,27s user 0,00s system
```

## 5.3 Limitations and conclusion

### 5.3.1 Limitations

We present here a few limitations of this method. Some are inherent to the method, some could be improved in future works.

#### 5.3.1.1 Working with concrete types

For the method to be efficient, the number of instructions generating multiple Horn clauses, and the number of clauses generated for each of such instruction should remain small. However, ProVerif has no semantic for concrete types (such as bit vectors or even numbers), and it is up to the user to model them. But this modeling is not obvious in ProVerif. Indeed, the definition of functions such as the addition of two bit vectors can be done in two ways.

- Either by constructors that construct *new* values so it would not be possible to express for instance that $1 + 0 = 1$.
- Or by destructors, which do not allow recursive definition, so we would need to explicitly give the result for each possible addition. In this case, if we have an instruction `add r2, r3` where `r2` and `r3` are controlled by the attacker and can take either of $n$ and $m$ values respectively, this instruction will be translated into $n.m$ Horn clauses which will considerably increase the complexity of the analysis.

An efficient representation of numbers should enable a translation of one instruction into only one clause (except for conditional instructions). We could imagine modifying ProVerif to add a new type of function operating over literals. This function would be ignored by the core algorithm of ProVerif. When trying to unify clauses, instead of simply looking for clauses with a conclusion and a premise that a substitution could make equal, ProVerif would call an SMT solver (as it is done by symbolic execution engines). If the solver could find an assignment of the symbolic variables that enables unification, it would add the constraint to the premises of the newly generated clause.

Implementing this modification could be part of future work. We believe it would benefit other applications, for instance, protocols that compute arithmetic expressions.

Another solution would be to find a language that shares some of the expected features presented in Section 5.2.1.1: an interesting attacker model, a simple reasoning and trace reconstruction. The closest language that comes to our mind is Prolog [73]. Like ProVerif, Prolog

enables users to provide clauses to relate predicates. Unlike ProVerif, it is able to work with numbers and to deal with recursive definition of compound terms. Both of these would be interesting for our setup. However, Prolog does not target specifically security properties. As such, trace reconstruction and security properties—not only confidentiality, but also authenticity—may need to be pre-processed to fit our purpose. Another important difference between ProVerif and Prolog is that the former tries to unify clauses to expand the knowledge of the attacker, whereas the latter unifies clauses that can result in the required property. This means that in a simple setup, Prolog would start unifying clauses from the end of the software. If the program contains an indirect jump which can only point to two locations, a backward analysis of the program would need to consider at each step that the indirect jump could potentially target the current instruction. Evaluating the consequence of this process on performance is left for future work.

#### 5.3.1.2 Working with machine code

Our goal was to be able to model complex attack scenarios that would take advantage of the concrete representation of data and code. While having an instruction-accurate model is a first step, we are still not working on a sufficiently low level to model attacks such as return-oriented programming, which would require a representation that preserves the dual semantic of bit vectors and instructions.

#### 5.3.1.3 Reconstructing attack traces

As ProVerif is able to output a trace leading to a violation of a required property, we could automate the process of translating such a trace into a succession of software-related events that would make more sense in our context. This could be valuable for the designer to distinguish between valid attacks and spurious traces.

### 5.3.2 Comparison with other similar projects

Formal verification of hardware-assisted security solutions has been performed and presented in various publications. We present in this section some of the ones that are most similar to our approach.

In [51], Cabodi et al. show how formal verification can be performed on various hardware-assisted remote attestation schemes. To do this, an existing CPU model is leveraged and hardware-specific clauses are added to this model to express the hardware specificities. Then, taint propagation verification is performed on this model using the VIS and PdTrav model checkers. This approach is close to ours as it gives a formal description of how hardware specificities impact the proof of the whole system. However, this work focus on a specific system design and the hardware model is manually modified. Contrary to our translation algorithm, the integration of hardware specificities into the model of the system based on a higher-level description of the hardware is not automated.

In [93], Delaune et al. model the functionalities offered by a TPM as first-order Horn clauses and show that a key sealed by the TPM cannot be recovered even for unbounded reboots and PCR extends. Since the interaction between the hardware (TPM) and the rest of the world is clearly defined (as functions to extend PCRs for instance), it is possible to manually give a formal definition of the hardware. As the previous work, it does not focus on providing an automated translation from high-level hardware model to formal models impacting the verification algorithm.

In [157], Kroening et al. present a translation algorithm that takes as input a description of a system using SystemC and issues labeled Kripke structures suitable for verification. The description of the system contains hardware and software modules and the transformation algorithm compose these modules by considering each possible synchronisation scenario. Contrary to our method, this transformation algorithm is able to work on a low-level hardware description (in SystemC). However, the composition of hardware and software modules rely on thread synchronisation points. These synchronisation points are the way hardware and software can communicate so there is no possibility to described and verify more tightly-coupled systems (for instance when a hardware module modify the execution of a software instruction).

### 5.3.3 Conclusion

The presented method targets formal verification of security properties on hardware/software designs. While the software part of the system is expressed in an implementation language, the hardware part is described as a higher-level model. This modeling language has been created to be modular and thus enable verifying systems presenting some hardware customization. This trade-off between hardware expressiveness, efficiency of the verification and manual interaction to guide the proof—by providing new hardware classes here—is suitable for verifying hardware-assisted security solutions where hardware and software components cannot be modeled as loosely interacting actors.

# Chapter 6

# Conclusion and perspectives

In this thesis we were interested in the role of automated formal security verification in the context of embedded system design. We have seen that the terms of this problem were very different depending on how hardware and software components interacted. Indeed, one characteristic of design for embedded system is that the constraints in terms of size, power consumption, reliability often require to use customized hardware, or even develop new hardware components. These customized hardware components may deeply affect how software executes and, therefore, how it should be analyzed.

As discussed in the introduction, a formal proof of software is always done with respect to a model of the environment and of a hardware which is an abstract representation of the hardware that would be executing the software. To formally verify a design described as a set of software and hardware components, it is thus possible to use existing software—or other high-level—verification tools only when the hardware components do not invalidate the proof performed on the software components. This consideration enabled us to address the problem of formal verification of embedded systems in two approaches:

- How is it possible to integrate automated formal security verification to the process of embedded system design when the high-level functionalities can be partitioned as hardware and software components?
- How is it possible to provide an automated formal verification method that would accept a description of hardware customization to verify a piece of software running on this modified hardware?

## 6.1 Contributions

**Integrating formal security verification to embedded system design models**

The specificity of embedded system hardware/software partitioning with respect to formal verification is also relevant from a high-level description. Indeed when a system is described in a PIM as abstract interacting components, the mapping of these high-level components to a physical architecture (described in a PSM) affects how the communication channels will be implemented. The channel could be implemented as an inter-process communication channel (queue, shared memory) if the two communicating components are mapped on the same physical processing unit, or it could be implemented as a physical bus if the components are mapped to different physical elements.

This difference in implementation entails a difference in terms of attacker model. A software channel may be considered secure—meaning that an attacker would not be able to spy on this channel—if the software implementing it (e.g., OS) is trusted. On the opposite, a software-implemented channel may be untrusted if it is implemented by an untrusted privileged software. Likewise, physical buses can be considered trusted or untrusted depending on whether an attacker could physically access it or not. The mapping of high-level functional elements to physical components therefore impacts the security of the whole system.

Security is thus—as safety or performance—a criteria that should be evaluated by an embedded system designer during system conception. This is why automated formal security verification is valuable during the conception phase to systematically and reliably evaluate the impact of early development choices made by an embedded system designer. Integrating formal verification to the design process of embedded systems thus appears to be an essential step toward secure embedded system design.

In chapter 4, we have proposed to leverage an existing embedded system modeling language featuring security artifacts to perform automated formal verification during the partitioning and software design phase. The design models are used to perform security, safety and performance analysis, thus minimizing the manual work needed to perform formal verification.

The algorithm implemented translates these design models into a specification suitable to be verified by the ProVerif tool. This translation is automated to enable system designers to perform formal verification with little specific knowledge. The verification results are backtraced to the design diagrams and a trace is provided when available, which helps designers in debugging and correcting their models.

#### Formal verification of tightly coupled hardware and software

When the hardware description modifies how the software is interpreted, a general-purpose model of the hardware cannot be assumed by the verification tool anymore. In order to enable hardware customization to be taken into account in the formal proof of the whole, the verification tool needs to be adapted to fit the hardware. This adaptation can be either static (once for all) or dynamic. An example of static adaptation would be to port a software verification tool to support another architecture. In this case, the tool is manually modified so that it takes the specificities of another hardware architecture.

In the case of embedded system design, the problem of formally verifying a system does not (only) come from the different processor architectures involved. The difficulties to verify such hardware/software systems are due to the amount of hardware customization available to the designer. This customization can take the form either of architecture customization or components customization. The hardware model that should be assumed by the software verification tool is dynamically evolving as the system is being designed.

In chapter 5, we have presented a translation algorithm which enables the designer to describe her system in the form of a software implementation and of a hardware architecture and transform it to a ProVerif specification suitable for formal verification. The hardware architecture can be customized to affect how the translation algorithm operates and thus how the software part will be verified.

## 6.2 Perspectives

The work presented here opens up perspectives for future work concerning the formal analysis of security properties of embedded systems. Some of these perspectives are generic and some are specific to the three domains that were explored during this thesis.

### 6.2.1 Environment for the security analysis of communicating systems

#### Articulation of the verification techniques

In this thesis, we have shown how to formally verify security properties on high-level models of embedded systems. We have also proposed a method to analyze tightly coupled hardware and software by relying on a low-level description of software. One of the perspectives opened by our contributions is thus to work on how these different verification methods could articulate. In particular, how is it possible to link security properties on high-level models to security properties on lower-level implementation, be it hardware/software system as presented in chapter 5 or traditional hardware or software implementation.

To do this, one would need to first detail the properties that are assumed by the high-level models. An example of such a property would be that all computations made internally by a component are trusted to be private and authentic. In order to provide a coherent proof of the design, verifying that this property holds for the refined implementations of the components would be valuable. In order to verify these properties, other formal verification methods could be used: either automated verification (model checking, symbolic execution) or partly manual verification with interactive theorem provers. This could be integrated to the TTool modeling and verification tool by automatically generating proof objectives or by including symbolic execution properties to automatically generated prototypes.

### 6.2.2 Trusted path on Intel SGX

#### Formally verifying the trusted path design

In section 4.3.2, we have presented a formal proof of the high-level protocol used for key exchange in the architecture proposed for trusted path. While this proof is interesting as it guarantees the confidentiality of a message sent by the enclave with the established key, the proof does not guarantee the security of the whole system. Proving that the policy enforcement implemented in the DMA remapping unit does prevent the OS from accessing enclave pages would be another interesting case of security verification of a hardware/software system.

As Intel SGX does not considerably affect how most of the machine instructions behave (apart from memory access control), it should be possible to model most of the system with SysML-Sec diagrams and perform formal verification according to the method presented in chapter 4. However, SysML-Sec currently lacks the ability to directly model tables, which would make modeling I/O page tables harder. As ProVerif has introduced capabilities to model tables, working on how these tables can be modeled on high-level design diagrams would be valuable.

#### Performance evaluation of the proposed solution

In chapter 3, we exposed various scenarios to progressively build a secure architectural solution to the problem of trusted path in an Intel SGX architecture. While some of these scenarios would require hardware modifications to the Intel SGX architecture and are thus difficult for us to implement, most could be implemented with existing components (like an Intel SGX enabled processor) and FPGAs.

Implementing them would enable to run benchmark tests to evaluate the performance of establishing an application–peripheral secure channel. We also expect that the performance would vary depending on the communication pattern of the application (streaming or through bursts, large contiguous memory chunks or small data packets). Evaluating these scenarios against different kind of application (HW accelerator, GPU, input controller) would be interesting

too. Note that a first estimation of the performance could be made with dedicated tools (the one proposed by TTool for instance).

### 6.2.3 Formal security analysis from design diagrams

#### Simplifying semantics of security properties

To enable system designers to quickly perform formal verification on design models, the ability to simply express expected security properties is essential. While ProVerif enables us to express some security properties straight-forwardly, some have a semantics that is hard to grasp for untrained designer.

The best example of this is the *authenticity* property. Intuitively, a received message is said to be authentic if it was crafted by the expected sender at the expected time. In ProVerif an authenticity property can thus be expressed as a logical implication: if message $m$ was received on state $r$ then sometimes before, $m$ was sent on state $s$. In SysML-Sec, it is possible to express an authenticity property to verify as a 4-uple containing:

- the state $s$ of a state machine of a block,
- the attribute $m_1$ that holds $m$ in state $s$,
- the state $r$ and
- the attribute $m_2$ that holds $m$ in state $r$.

Let us imagine that a designer is modeling a situation where a client and a server application are exchanging messages. The server has initially no knowledge about the client it is going to talk to. At the start of the protocol the client and the server exchange a key to protect their communication. Then a first secret message is sent by the client and received by the server. The designer would be interested in proving that (intuitively) once a key has been shared, an attacker is not able to impersonate the client. Modeling this property is not obvious. Indeed, a first idea would be to use the authenticity property as previously described. However, since the server accepts communication requests from anyone, the message received by the server at the end of the key exchange protocol does not necessarily come from the client. In particular, the attacker could contact the server as a genuine client while this would not be considered as a vulnerability of the protocol.

Working on how security properties can be intuitively expressed by the designers to efficiently convey their expectations would be an interesting work that was not addressed during this thesis and is left for future work.

#### Improving reconstruction of traces

Formal verification of a security property is valuable to a designer as it enables her to correct a vulnerable design. As such, the results issued by the verification method would preferably help the designer in precisely pinpointing and understanding the origin of a vulnerability. As mentioned in chapter 4, we have implemented a translation algorithm to provide traces on design diagrams based on the output of ProVerif.

However, traces are complicated by the artifacts added during the SysML-Sec-to-ProVerif translation and by the approximation made by ProVerif. A simple example of that is: when a property may be proved false by a trace where component $a$ receives a message $m$ from component $b$, ProVerif will generate two steps. In the first step, the attacker receives $m$ from $b$ and in the second step, the attacker sends $m$ to $a$. The attacker will thus appear in the trace while its

action is not needed. Conversely, if the trace relies on the fact that $a$ may receive $m$ twice, then the attacker will have to be present to replay the message $m$. This is just an example of why translating ProVerif traces back to high-level design diagrams is non obvious. Note that while these traces may confuse a designer who would not be familiar with the translation and verification algorithms, they are valid attack traces.

#### Exploring other verification back-ends

While ProVerif was a suitable tool for our purpose (ability to model cryptographic primitives, attacker model, channel semantics), its inability to work with concrete types (such as bit vectors) limits the extent of SysML-Sec features that can be taken into account by the proof. It would thus be interesting to either

- add to ProVerif the ability to work with concrete types or
- explore other back-ends for the verification of security properties.

In the first case, it would probably be possible to modify the clauses unification algorithm implemented by ProVerif to rely on a theory-specific solver. This would be both beneficial to verification of SysML-Sec diagrams and to direct verification of pi-calculus cryptographic protocols using concrete types.

In the second case, other verification languages and other verification tools could be interesting for our purpose. Such a language could for instance be Prolog, which enables to work with concrete types. However, Prolog is not specifically targeting security properties and so the generation of attacker clauses and of security properties would need to be manually implemented.

### 6.2.4 Formal verification of tightly coupled hardware and software

#### Other verification algorithms

Likewise, SMASHUP suffers from the inability of ProVerif to deal with concrete types. Also, while the simplicity of the ProVerif specification language is suitable to model specificities introduced by customized hardware, it does not originally target software verification. It would be interesting, from a performance point of view to evaluate other verification methods specifically targeting software implementations. One such example would be symbolic execution.

In order to include customized hardware specificities to the verification algorithm, an existing software verification tool would need to be modified to consider hardware specificities when verifying software. As we did with SMASHUP, a hardware description language would need to be defined (or the one we defined could be reused) and the interface between this language and the verification tool would need to be implemented.

#### Expressiveness of the hardware description language

SMASHUP hardware description language is a high-level description of an architecture composed of parametrizable modules. While this enables to model some hardware customization, it would be beneficial to rely on a lower-level description of the hardware. There are two reasons why:

- Low-level description languages enable to model highly customized designs.
- It is more difficult to—formally or intuitively—verify that an implementation is conform to a high-level model than to a lower-level implementation.

We do not believe that the hardware description language could be as low-level as traditional register transfer languages (verilog, VHDL or even SystemC when it is used as a hardware description language) as expressing how the hardware modifications impact software verification would probably be too difficult. Nonetheless, a lower-level description of the hardware would increase the confidence the designer can put in the overall proof.

**Verification of the hardware**

Last but not least, it could be interesting to study how this verification method could articulate with a hardware formal verification technique. Indeed, while integrating the hardware customized architecture to the verification of software is valuable to enable designer to quickly see the impact of hardware modification on security, it does not cover a system down to a low-level implementation. The high-level hardware description model assumes some properties on the hardware modules (reflected by how they affect software verification) but there is no proof that the low-level implementation will comply to these abstract properties.

This proof could thus be extended by verifying that the hardware implementation is a refinement of the high-level model. The higher-level the hardware description model, the harder this refinement proof would be. Working on how formal hardware verification method can articulate with the hardware description language is therefore another interesting question that is left for future work.

# Appendices

# Appendix A

# Proof of correctness of the SysML-Sec to ProVerif translation algorithm

We give in this appendix a formal proof of Theorem 1.

## A.1 Objective

To simplify the presentation we say that—given the parallel composition of a SysML-Sec design $\mathcal{C}$ and the ProVerif process $P_0$ resulting from the translation of $\mathcal{C}$ through the algorithm formally described in this chapter—$\mathcal{T}_{P_0}$ is a trace of $P_0$ if it is a trace of $P_0$ from $\mathcal{S}_0$ where $\mathcal{S}_0$ is the set containing the names of all channels used in input and output actions in $\mathcal{C}$ (we do not consider private channels). The notations $\mathcal{C}$, $\mathcal{S}_0$ and $P_0$ will be used in the proof and they will always keep this meaning. Also, we will use the notations introduced during the presentation of the translation algorithm. In particular, $\mathcal{E}_v$ is the set which contains all the states of all the blocks of the design which are roots of a basic block (for instance in Figure 4.6, the roots of the basic blocks are the initial state, $s_6$ and $s_7$). Note also that if $\sigma = \{(x_1, M_1), ..., (x_n, M_n)\}$ is a substitution and $P$ a process, we note $P\sigma$ the result of the substitution applied to $P$: $P\sigma = P\{M_1/x_1, ..., M_n/x_n\}$.

The essence of the proof is to show that $\forall n \in \mathbb{N}$ the following property $\mathcal{H}_n$ holds:

**Property 1** *$\mathcal{H}_n$ : For any trace of $\mathcal{C}$ of length $n$*

$$\mathcal{T}_{\mathcal{C}} = \mathcal{N}_1, \Sigma_1, \mathcal{K}_1, \sigma_1 \ \rightarrow \ ... \ \rightarrow \ \mathcal{N}_n, \Sigma_n, \mathcal{K}_n, \sigma_n$$

*$\exists k \in \mathbb{N}$ and there exists a trace of $P_0$*

$$\mathcal{T}_{P_0} = \mathcal{E}_1, \mathcal{P}_1, \mathcal{S}_1 \ \rightarrow \ ... \ \rightarrow \ \mathcal{E}_k, \mathcal{P}_k, \mathcal{S}_k$$

*such that*

- $\mathcal{E}_1 = \mathcal{S}_0$, $\mathcal{P}_1 = \{P_0\}$ *and* $\mathcal{S}_1 = \mathcal{S}_0$,
- $\mathcal{N}_n \subset \mathcal{E}_k$
- $\mathcal{P}_k = \bigcup_{q \in \mathcal{E}_v} \{ \ !\llbracket q \rrbracket^p \ \} \quad \cup \quad \bigcup_{s \in \Sigma_n} \{ \ \llbracket s \rrbracket^s \sigma_n \ \}$
- $\mathcal{K}_n \subset \mathcal{S}_k$

## A.2 Proof

We show this property by induction on the length of the trace $n$.

### A.2.1 Base case

First we show that $\mathcal{H}_0$ holds. The main process $P_0$ is defined on page 99 as:

$$Main_{\mathcal{E}}(\mathcal{D}) = \Big( \bigoplus_{b \in block(\mathcal{D})} \Big( \bigoplus_{a \in att(b)} \text{"new } a\text{;"} \oplus \text{"in(chctrl, nonce);} \\ \text{out}\Big(\text{chctrl}, \text{tok}\big(\text{token\_}\mathbb{L}(q_0), \text{nonce}, args\big)\Big)\text{"}\Big)\Big) \\ \text{"}\Big|\text{"}_{q \in \mathcal{E}_v} \Big(\text{"!proclabel\_}\mathbb{L}(q)\text{"}\Big)$$

If we apply the reduction rule *Red Par* (see the definition on page 97) once for each $q \in \mathcal{E}_v$ we get a trace:

$$\mathcal{E}_1, \mathcal{P}_1, \mathcal{S}_1 \xrightarrow{RedPar} \ldots \xrightarrow{RedPar} \mathcal{E}_1, \bigcup_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \\ \cup \Bigg\{ \bigoplus_{b \in block(\mathcal{D})} \Big( \bigoplus_{a \in att(b)} \text{"new } a\text{;"} \oplus \text{"in(chctrl, nonce);} \\ \text{out}\Big(\text{chctrl}, \text{tok}\big(\text{token\_}\mathbb{L}(q_0), \text{nonce}, \overrightarrow{a}^{a \in att(b)}\big)\Big)\text{"}\Big)\Bigg\}, \\ \mathcal{S}_1$$

From here on, we will use the notation

$$\rightarrow\ \mathcal{E}, \mathcal{P}, \mathcal{S}$$

to mean that the configuration $\mathcal{E}, \mathcal{P}, \mathcal{S}$ can be reduced from the last presented step by applying one or more reduction rules. The reduction rules that need to be applied will be explicitly stated at each step.

The rule *Red Repl* is then applied once for each $b \in block(\mathcal{D})$ to the process created from the initial state of $b$, i.e., $\llbracket q_0(b) \rrbracket^p$. We obtain:

$$\rightarrow \mathcal{E}_1, \bigcup_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \quad \cup \quad \bigcup_{b \in block(\mathcal{D})} \{\ \llbracket q_0(b) \rrbracket^p\ \} \\ \cup \Bigg\{ \bigoplus_{b \in block(\mathcal{D})} \Big( \bigoplus_{a \in att(b)} \text{"new } a\text{;"} \oplus \text{"in(chctrl, nonce);} \\ \text{out}\Big(\text{chctrl}, \text{tok}\big(\text{token\_}\mathbb{L}(q_0), \text{nonce}, \overrightarrow{a}^{a \in att(b)}\big)\Big)\text{"}\Big)\Bigg\}, \\ \mathcal{S}_1$$

For each block $b \in block(\mathcal{D})$, several reduction rules are applied sequentially in the following order:

- the rule *Red Restr* is applied for each $a \in att(b)$,

- the rule *Red Restr* is applied for the nonce of $[\![q_0(b)]\!]^p$,
- the rule *Red Out* is applied to the out of the nonce that was just restricted,
- the rule *Red In* is applied so that the name nonce is replaced by the name that was extracted from the other process,
- the rules *Red Out* and *Red In* are again applied to pass the token,
- the rule *Red Let1* is applied to $[\![q_0]\!]^p$ to remove the token function and to test equality of the label of the process and of the nonce.

$$\begin{array}{l} \rightarrow \mathcal{E}_1 \quad \cup \quad \bigcup_{b \in block(\mathcal{D})} att(b) \quad \cup \quad \mathcal{E}', \\ \bigcup_{q \in \mathcal{E}_v} \{\ ![\![q]\!]^p\ \} \quad \cup \quad \bigcup_{b \in block(\mathcal{D})} \{\ [\![q_0(b)]\!]^p\ \}, \\ \mathcal{S}_1 \quad \cup \quad \mathcal{S}' \end{array}$$

Where $\mathcal{E}'$ contains all nonces created by the processes and $\mathcal{S}'$ contains all these nonces and all the tokens exchanged.

All traces of $\mathcal{C}$ start from the state:

$$\left( \bigcup_{b \in block(\mathcal{D})} att(b)\ ,\quad \bigcup_{b \in block(\mathcal{D})} \{q_0(b)\}\ ,\quad \emptyset\ ,\quad Id \right)$$

where $Id$ is the identity function.

So the trace that was just constructed verifies $\mathcal{H}_0$.

### A.2.2 Induction step

Let now $n \in \mathbb{N}$ and suppose $\mathcal{H}_n$. Let $\mathcal{T}_\mathcal{C}$ be a trace of $\mathcal{C}$ of length $n+1$:

$$\mathcal{T}_\mathcal{C} = \mathcal{N}_1, \Sigma_1, \mathcal{K}_1, \sigma_1 \ \rightarrow\ ...\ \rightarrow\ \mathcal{N}_n, \Sigma_n, \mathcal{K}_n, \sigma_n$$

By $\mathcal{H}_n$, $\exists k \in \mathbb{N}$ and there exists a trace of $P_0$

$$\mathcal{T}_{P_0} = \mathcal{E}_1, \mathcal{P}_1, \mathcal{S}_1 \ \rightarrow\ ...\ \rightarrow\ \mathcal{E}_k, \mathcal{P}_k, \mathcal{S}_k$$

such that

- $\mathcal{N}_n \subset \mathcal{E}_k$
- $\mathcal{P}_k = \bigcup_{q \in \mathcal{E}_v} \{\ ![\![q]\!]^p\ \} \quad \cup \quad \bigcup_{s \in \Sigma_n} \{\ [\![s]\!]^s \sigma_n\ \}$
- $\mathcal{K}_n \subset \mathcal{S}_k$

By definition of $\mathcal{C}$, the transition $\mathcal{N}_n, \Sigma_n, \mathcal{K}_n, \sigma_n \ \rightarrow\ \mathcal{N}_{n+1}, \Sigma_{n+1}, \mathcal{K}_{n+1}, \sigma_{n+1}$ can only be of two types:

- either it corresponds to $atk_{new}$, to $atk_{function}$ or to a transition from the state machine of a block $b \in \mathcal{B}$
- or it corresponds to two transitions in the original state machines: one bearing an input action (of a block or $atk_{in}$) and one bearing an output action (of a block or $atk_{out}$).

#### A.2.2.1 One transition was triggered

In the first case, there are three sub-cases: the transition in $\mathcal{C}$ corresponds to either $atk_{new}$, to $atk_{function}$ or to a transition in the state machine of a block.

**Sub-case 1:** $atk_{new}$

If the transition corresponds to $atk_{new}$, then this means that the attacker has created a new name $a$. By definition, we have:

- $\mathcal{N}_{n+1} = \mathcal{N}_n \cup \{a\}$
- $\Sigma_{n+1} = \Sigma_n$
- $\mathcal{K}_{n+1} = \mathcal{K}_n \cup \{a\}$
- $\sigma_{n+1} = \sigma_n$

By applying the rule *Red New* to the current state $\mathcal{E}_k, \mathcal{P}_k, \mathcal{S}_k$ we obtain $\mathcal{E}_k \cup \{a\}$, $\mathcal{P}_k$, $\mathcal{S}_k \cup \{a\}$. Note that:

- Since $\mathcal{N}_n \subset \mathcal{E}_k$, we have $\mathcal{N}_{n+1} \subset \mathcal{E}_{k+1} = \mathcal{E}_k \cup \{a\}$
- $\mathcal{P}_{k+1} = \mathcal{P}_k = \bigcup_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \cup \bigcup_{s \in \Sigma_{n+1}} \{\ \llbracket s \rrbracket^s \sigma_{n+1}\ \}$ (since $\Sigma_{n+1} = \Sigma_n$ and $\sigma_{n+1} = \sigma_n$)
- Since $\mathcal{K}_n \subset \mathcal{S}_k$, we have $\mathcal{K}_{n+1} \subset \mathcal{S}_{k+1} = \mathcal{S}_k \cup \{a\}$

**Sub-case 2:** $atk_{function}$

In the second sub-case, the transition corresponds to $atk_{function}$. The attacker has applied a function $f$ to known terms $M_1, ..., M_m$. We have:

- $\mathcal{N}_{n+1} = \mathcal{N}_n$
- $\Sigma_{n+1} = \Sigma_n$
- $\mathcal{K}_{n+1} = \mathcal{K}_n \cup \{f(M_1, ..., M_m)\}$
- $\sigma_{n+1} = \sigma_n$

The function can be either a constructor or a destructor. In both cases it is possible to construct a trace which guarantees the expected properties by applying respectively the rules *Red Constr* or *Red Destr*.

**Sub-case 3: Transition in the state machine of a block**

In the third sub-case, $\exists b \in block(\mathcal{D})$ such that the transition in $\mathcal{C}$ corresponds to a transition in $b$. Let us call this transition $e$. If $\Sigma_n = (s_n^1, ..., s_n^m)$ then $\exists i \in \llbracket 1, m \rrbracket,\ s_n^i = source(e)$. In this case we have $\Sigma_{n+1} = (s_{n+1}^1, ..., s_{n+1}^m)$ and $\forall j \in \llbracket 1, m \rrbracket,\ j \neq i \implies s_{n+1}^j = s_n^j$ and $s_{n+1}^i = target(e)$. It should also be noted that since $s_n^i$ is the source of an edge, it is not the final state of its state machine. Finally, since only one transition was triggered, it means that the action on the edge is not an in or out action. This transition will thus not increase the knowledge of the attacker: $\mathcal{K}_{n+1} = \mathcal{K}_n$.

We have $\mathcal{P}_k = \bigcup_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \quad \cup \quad \bigcup_{j \in [\![1,m]\!]} \{\ \llbracket s_n^j \rrbracket^s \sigma_n\ \}$. We apply the rule *Red Event* to $\llbracket s_n^i \rrbracket^s \sigma_n$ which yields a trace to a configuration where $\llbracket s_n^i \rrbracket^s \sigma_n$ has been reduced to either $\llbracket s_n^i, e \rrbracket^t \sigma_n$ if $UniqueOut(s_n^i)$ or $\llbracket s_n^i \rrbracket^m \sigma_n$ otherwise.

If $s_n^i$ has multiple outgoing transitions, we apply for each $e' \in \mathcal{O}ut(s_n^i)$ the rules *Red Restr* and *Red Out* to $\llbracket s_n^i \rrbracket^m \sigma_n$. We have then a trace of $P_0$:

$$\begin{aligned}
\mathcal{E}_1, \mathcal{P}_1, \mathcal{S}_1 \ &\rightarrow \ ... \ \rightarrow \ \mathcal{E}_k, \mathcal{P}_k, \mathcal{S}_k \ \rightarrow \ ... \\
&\rightarrow \ \mathcal{E}_k \cup \bigcup_{e' \in \mathcal{O}ut(s_n^i)} \{x_{e'}\}, \\
&\quad \bigcup_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \quad \cup \quad \bigcup_{j \in [\![1,m]\!], j \neq i} \{\ \llbracket s_n^j \rrbracket^s \sigma_n\ \} \\
&\quad \cup \left\{ \texttt{"in (chctrl, } c\texttt{);"} \bigoplus_{e' \in \mathcal{O}ut(s_n^i)} \left( \texttt{"if } c = x_{e'} \texttt{ then"} \oplus \llbracket s_n^i, e' \rrbracket^t \sigma_n \right) \right\}, \\
&\quad \mathcal{S}_k \cup \bigcup_{e' \in \mathcal{O}ut(s_n^i)} \{x_{e'}\}
\end{aligned}$$

We then apply the rule *Red In* to substitute $c$ with $x_e$, then the rule *Red Let2* for each $e' \neq e$ and finally the rule *Red Let1*. Then $\exists k' \in \mathbb{N}$ such that

$$\begin{aligned}
\rightarrow \ & \mathcal{E}_{k'}, \\
& \bigcup_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \quad \cup \quad \bigcup_{j \in [\![1,m]\!], j \neq i} \{\ \llbracket s_n^j \rrbracket^s \sigma_n\ \} \ \cup \ \{\llbracket s_n^i, e \rrbracket^t \sigma_n\}, \\
& \mathcal{S}_{k'}
\end{aligned}$$

with $\mathcal{E}_k \subset \mathcal{E}_{k'}$ and $\mathcal{S}_k \subset \mathcal{S}_{k'}$. Note that this result is true also if $UniqueOut(s_n^i)$.

If the guard on $e$ was not empty, then its result for the current value of the attributes will evaluate to true as the transition $e$ was triggered in the valid trace $\mathcal{T}_\mathcal{C}$. This means that the rule *Red Let1* can be applied to consume the guard. We obtain then:

$$\begin{aligned}
\rightarrow \ & \mathcal{E}_{k'}, \\
& \bigcup_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \quad \cup \quad \bigcup_{j \in [\![1,m]\!], j \neq i} \{\ \llbracket s_n^j \rrbracket^s \sigma_n\ \} \ \cup \ \{\llbracket s_n^i, e \rrbracket^a \sigma_n\}, \\
& \mathcal{S}_{k'}
\end{aligned}$$

We now need to consume the action. The reduction rule to apply depends on the action:

- If $action(e) = x := exp$ then $exp$ is either
  - the application of a destructor to attributes $g(a_1, ..., a_j)$,
  - the application of a constructor to attributes $f(a_1, ..., a_j)$ or
  - the affectation of the value of an attribute $a$.

  If we consider the simple destructor $M \rightarrow M$, these three cases can be grouped under a common case of a destructor applied to terms which can be either an attribute or a constructor applied to attributes. Moreover, since the transition $e$ was triggered, it means that the eventual destructors present in $exp$ could be reduced (there was a term $M$ such that $g(a_1, ..., a_j)\sigma_n \ \rightarrow \ M$). In the last two cases, $M$ respectively equals $f(a_1, ..., a_j)\sigma_n$

and $\sigma_n(a)$. $M$ is a term over the elements of $\mathcal{N}_n$ and after the transition the attribute $x$ recieves the value $M$ so $\sigma_{n+1} = \sigma_n\{M/x\}$. We can apply the rule *Red Let1* and obtain:

$$\rightarrow \begin{array}{l} \mathcal{E}_{k'}, \\ \bigcup\limits_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \quad \cup \quad \bigcup\limits_{j \in [\![1,m]\!], j \neq i} \{\ \llbracket s_n^j \rrbracket^s \sigma_n\ \} \cup \{\llbracket target(e) \rrbracket^c \sigma_n\{M/x\}\}, \\ \mathcal{S}_{k'} \end{array}$$

Moreover, since all blocks manipulate disjoints sets of attributes, we have

$$\forall j \in [\![1,m]\!],\ j \neq i \implies \llbracket s_n^j \rrbracket^s \sigma_n = \llbracket s_{n+1}^j \rrbracket^s \sigma_n = \llbracket s_{n+1}^j \rrbracket^s \sigma_{n+1}$$

We also have $\mathcal{N}_{n+1} = \mathcal{N}_n \subset \mathcal{E}_{k'}$.

- If $action(e) = \nu.x$ then we have $\sigma_{n+1} = \sigma_n$ and $\mathcal{N}_{n+1} = \mathcal{N}_n \cup \{x\}$. If we apply the rule *Red Restr*, we obtain:

$$\rightarrow \begin{array}{l} \mathcal{E}_{k'} \cup \{x\}, \\ \bigcup\limits_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \quad \cup \quad \bigcup\limits_{j \in [\![1,m]\!], j \neq i} \{\ \llbracket s_n^j \rrbracket^s \sigma_n\ \} \cup \{\llbracket target(e) \rrbracket^c \sigma_n\}, \\ \mathcal{S}_{k'} \end{array}$$

  Since $\mathcal{N}_n \subset \mathcal{E}_k \subset \mathcal{E}_{k'}$, we have $\mathcal{N}_{n+1} \subset \mathcal{E}_{k'} \cup \{x\}$

- If $action(e) = \epsilon$, nothing needs to be done.

In all of these cases, we have shown that $\exists k'' \in \mathbb{N}$ and there is a trace of $P_0$

$$\rightarrow \begin{array}{l} \mathcal{E}_{k''}, \\ \bigcup\limits_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \quad \cup \quad \bigcup\limits_{j \in [\![1,m]\!], j \neq i} \{\ \llbracket s_{n+1}^j \rrbracket^s \sigma_{n+1}\ \} \cup \{\llbracket s_{n+1}^i \rrbracket^c \sigma_{n+1}\}, \\ \mathcal{S}_{k''} \end{array}$$

with $\mathcal{N}_{n+1} \subset \mathcal{E}_{k''}$ and $\mathcal{K}_{n+1} \subset \mathcal{S}_{k''}$.

If $UniqueIn(s_{n+1}^i)$, $\llbracket s_{n+1}^i \rrbracket^c = \llbracket s_{n+1}^i \rrbracket^s$ so we have found a trace with the expected properties. Otherwise, $s_{n+1}^i \in \mathcal{E}_v$

$$\rightarrow \begin{array}{l} \mathcal{E}_{k''}, \\ \bigcup\limits_{q \in \mathcal{E}_v} \{\ !\llbracket q \rrbracket^p\ \} \quad \cup \quad \bigcup\limits_{j \in [\![1,m]\!], j \neq i} \{\ \llbracket s_{n+1}^j \rrbracket^s \sigma_{n+1}\ \} \\ \cup \left\{ \text{"}\mathsf{in}(\mathsf{chctrl}, \mathsf{nonce})\,;\ \mathsf{out}\Big(\mathsf{chctrl}, \mathsf{tok}\big(\mathsf{token\_}\mathbb{L}(s_{n+1}^i), \mathsf{nonce}, \overrightarrow{d}^{a \in att(b)}\sigma_{n+1}\big)\Big)\text{.}\text{"} \right\}, \\ \mathcal{S}_{k''} \end{array}$$

- We apply the rule *Red Repl* to $!\llbracket s_{n+1}^i \rrbracket^p$.
- We apply the rule *Red Restr* to consume the nonce created by $\llbracket s_{n+1}^i \rrbracket^p$.
- We apply the rules *Red Out* and *Red In* to transmit the nonce.
- We apply the rules *Red Out* and *Red In* to transmit the token.
- The rule *Red Let1* is applied to $\llbracket s_{n+1}^i \rrbracket^p$ to remove the token function and to test equality of the label of the process and of the nonce.

Eventually we obtain ($k''' \in \mathbb{N}$):

$$\rightarrow \quad \mathcal{E}_{k'''}, \bigcup_{\substack{q \in \mathcal{E}_v \\ \mathcal{S}_{k'''}}} \{\ !\llbracket q \rrbracket^p\ \} \quad \cup \bigcup_{s \in \Sigma_{n+1}} \{\ \llbracket s \rrbracket^s \sigma_{n+1}\ \},$$

with $\mathcal{N}_{n+1} \subset \mathcal{E}_{k''} \subset \mathcal{E}_{k'''}$ and $\mathcal{K}_{n+1} \subset \mathcal{S}_{k''} \subset \mathcal{S}_{k'''}$.

#### A.2.2.2 Two transition were triggered

In the second case where two transitions are triggered simultaneously, the proof is mostly identical. The difference lies in the fact that for one induction step, both transitions must be considered and that the *Red In* and *Red Out* rules need to be applied to consume the actions. This concludes the proof of $\mathcal{H}_n$ for all $n \in \mathbb{N}$.

## A.3 Conclusion

Let $\mathcal{C}$ be the parallel composition of a SysML-Sec design and $P_0$ be its translation in ProVerif language. Let $a$ be a non-confidential attribute of $\mathcal{C}$. By definition, there exists a trace of length $n$

$$\mathcal{T}_{\mathcal{C}} = \mathcal{N}_1, \Sigma_1, \mathcal{K}_1, \sigma_1 \ \rightarrow \ ... \ \rightarrow \ \mathcal{N}_n, \Sigma_n, \mathcal{K}_n, \sigma_n$$

such that $a \in \mathcal{K}_n$. According to $\mathcal{H}_n$, there is a trace of $P_0$

$$\mathcal{T}_{P_0} = \mathcal{E}_1, \mathcal{P}_1, \mathcal{S}_1 \ \rightarrow \ ... \ \rightarrow \ \mathcal{E}_k, \mathcal{P}_k, \mathcal{S}_k$$

such that $\mathcal{K}_n \subset \mathcal{S}_k$. So $a \in \mathcal{S}_k$ and thus $a$ is not confidential in the ProVerif specification.

# Bibliography


[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. In *ACM Transactions on Information and System Security (TISSEC)*, volume 13, pages 4:1–4:40, New York, NY, USA, November 2009. ACM.

[2] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, Jan 2002.

[3] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology – CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*, pages 225–242, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] Michel Agoyan, Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, Anne-Lise Ribotta, and Assia Tria. How to flip a bit? In *2010 IEEE 16th International On-Line Testing Symposium*, pages 235–239, July 2010.

[6] Sunha Ahn and Sharad Malik. Automated firmware testing using firmware-hardware interaction patterns. In *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, Oct 2014.

[7] Mehdi-Laurent Akkar and Christophe Giraud. An implementation of DES and AES, secure against some attacks. In *Cryptographic Hardware and Embedded Systems — CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings*, pages 309–318, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[8] Yomna Ali, Sherif El-Kassas, and Mohy Mahmoud. A rigorous methodology for security architecture modeling and verification. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–10, Jan 2009.

[9] Xavier Allamigeon and Bruno Blanchet. Reconstruction of Attacks against Cryptographic Protocols. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, 2005.

[10] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security, 01 2004.

[11] Roberto M. Amadio, Denis Lugiez, and Vincent Vanackère. On the symbolic reduction of processes with cryptographic functions. In *Theor. Comput. Sci.*, 2003.

[12] Luca Amarú, Pierre-Emmanuel Gaillardon, Robert Wille, and Giovanni De Micheli. Exploiting inherent characteristics of reversible circuits for faster combinational equivalence checking. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, DATE '16, pages 175–180, San Jose, CA, USA, 2016. EDA Consortium.

[13] Axelle Apvrille. Geek usages for your Fitbit Flex tracker. Slides at framadrive.org/index.php/s/Wk6nxAKMpVTdQl4, October 2015.

[14] Ludovic Apvrille, Waseem Muhammad, Rabéa Ameur-Boulifa, Sophie Coudert, and Renaud Pacalet. A UML-based environment for system design space exploration. In *2006 13th IEEE International Conference on Electronics, Circuits and Systems*, pages 1272–1275, Dec 2006.

[15] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*, pages 65–71, May 1997.

[16] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings*, pages 281–285, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[17] Alessandro Armando, Luca Compagna, and Pierre Ganty. SAT-based model-checking of security protocols using planning graph analysis. In *FME 2003: Formal Methods: International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003. Proceedings*, pages 875–893, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[18] Frédéric Badeau and Arnaud Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In *ZB 2005: Formal Specification and Development in Z and B: 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005. Proceedings*, pages 334–354, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[19] Nikilesh Balakrishnan, Lucian Carata, Thomas Bytheway, Ripduman Sohan, and Andy Hopper. Non-repudiable disk i/o in untrusted kernels. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 24:1–24:6, New York, NY, USA, 2017. ACM.

[20] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, April 2003.

[21] Tom Ball, Byron Cook, Vladimir Levin, and Sriram Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. Technical report, Microsoft, January 2004.

[22] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. The page-fault weird machine: Lessons in instruction-less computation. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Washington, D.C., 2013. USENIX.

[23] Alessandro Barenghi, Guido Bertoni, Emanuele Parrinello, and Gerardo Pelosi. Low voltage fault attacks on the RSA cryptosystem. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 23–31, Sept 2009.

[24] Alessandro Barenghi, Guido M. Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. Low voltage fault attacks to AES. In *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 7–12, June 2010.

[25] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, pages 146–166, Cham, 2014. Springer International Publishing.

[26] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 71–90, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[27] David Basin, Sebastian Mödersheim, and Luca Viganò. An on-the-fly model-checker for security protocol analysis. In *Computer Security – ESORICS 2003: 8th European Symposium on Research in Computer Security, Gjøvik, Norway, October 13-15, 2003. Proceedings*, pages 253–270, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[28] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of computer programming*, 16(2):103–149, 1991.

[29] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[30] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[31] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[32] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, Oct 2007.

[33] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[34] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology — CRYPTO '97: 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings*, pages 513–525, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[35] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, pages 82–, Washington, DC, USA, 2001. IEEE Computer Society.

[36] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 15 pp.–154, May 2006.

[37] Bruno Blanchet. "security protocol verification: Symbolic and computational models. In *Principles of Security and Trust: First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, pages 3–29, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[38] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. Automatic Cryptographic Protocol Verifier, User Manual and Tutorial. Technical report, INRIA, 2015.

[39] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 353–362, New York, NY, USA, 2011. ACM.

[40] Simon Bliudze and Joseph Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR 2008 - Concurrency Theory: 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, pages 508–522, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[41] Dominique Bolignano. Towards a mechanization of cryptographic protocol verification. In *Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings*, pages 131–142, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[42] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology — EUROCRYPT '97: International Conference on the Theory and Application of Cryptographic Techniques Konstanz, Germany, May 11–15, 1997 Proceedings*, pages 37–51, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[43] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems - CHES 2006: 8th International Workshop, Yokohama, Japan, October 10-13, 2006. Proceedings*, pages 201–215, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[44] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. IKOS: A framework for static analysis based on abstract interpretation. In *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, pages 271–277, Cham, 2014. Springer International Publishing.

[45] Robert K Brayton, Gary D Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, et al. VIS: A system for verification and synthesis. In *International conference on computer aided verification*, pages 428–432. Springer, 1996.

[46] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities, 2006. jean-pierre.seifert@intel.com 13192 received 13 Feb 2006.

[47] Schneier Bruce. Attack trees. *Dr Dobb's Journal*, 24(12), 1999.

[48] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[49] Jeremie Brunel, Renaud Pacalet, Salaheddine Ouaarab, and Guillaume Duc. Secbus, a software/hardware architecture for securing external memories. In *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 277–282, April 2014.

[50] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural support for software-based protection. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 42–51, New York, NY, USA, 2006. ACM.

[51] Gianpiero Cabodi, Paolo Camurati, Carmelo Loiacono, Giovanni Pipitone, Francesco Savarese, and Danilo Vendraminetto. Formal verification of embedded systems for remote attestation. *WSEAS TRANSACTIONS ON COMPUTERS*, 14:760–769, 2015.

[52] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[53] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. In *Communications of the ACM*, volume 56, pages 82–90, New York, NY, USA, February 2013. ACM.

[54] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 3–11, Cham, 2015. Springer International Publishing.

[55] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, 2014. USENIX Association.

[56] Elad Carmon, Jean-Pierre Seifert, and Avishai Wool. Photonic side channel attacks against RSA. *IACR Cryptology ePrint Archive*, 2017:108, 2017.

[57] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 400–409, New York, NY, USA, 2009. ACM.

[58] David Champagne and Ruby B. Lee. Scalable architectural support for trusted software. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010.

[59] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[60] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *SIGPLAN Not.*, volume 48, pages 253–264, New York, NY, USA, March 2013. ACM.

[61] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt. Differential power analysis of a McEliece cryptosystem. In *Applied Cryptography and Network Security: 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, pages 538–556, Cham, 2015. Springer International Publishing.

[62] Xi Chen, Robert P. Dick, and Alok Choudhary. Operating system controlled processor-memory bus encryption. In *2008 Design, Automation and Test in Europe*, pages 1154–1159, March 2008.

[63] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *SIGPLAN Not.*, volume 43, pages 2–13, New York, NY, USA, March 2008. ACM.

[64] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Ding Xuhua, and Robert Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 01 2014.

[65] Yannick Chevalier and Laurent Vigneron. A tool for lazy verification of security protocols. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 373–376, Nov 2001.

[66] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *SIGPLAN Not.*, volume 47, pages 265–278, New York, NY, USA, March 2011. ACM.

[67] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 38–49, New York, NY, USA, 2016. ACM.

[68] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. In *Advances in Cryptology – ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 62–81, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[69] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An open-source tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 359–364, London, UK, UK, 2002. Springer-Verlag.

[70] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[71] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 368–371, New York, NY, USA, 2003. ACM.

[72] Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[73] William Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[74] Lucian Constantin. Researchers hack Tesla Model S with remote attack. http://www.pcworld.com/article/3121999/security/researchers-demonstrate-remote-attack-against-tesla-model-s.html, September 2016.

[75] Intel Corporation. Intel data direct i/o technology overview, 2012.

[76] Intel Corporation. Intel virtualization technology for directed i/o: Spec, 2016.

[77] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016.

[78] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, 2016. USENIX Association.

[79] Andrei Costin. Ghost is in the air(traffic): On insecurity of ADS-B protocol and practical attacks on ADS-B devices, 2012.

[80] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[81] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *Esop*, volume 5, pages 21–30. Springer, 2005.

[82] Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43(Supplement C):139 – 155, 2015.

[83] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, New York, NY, USA, 2015. ACM.

[84] Jean-Luc Danger, Sylvain Guilley, Thibault Porteboeuf, Florian Praden, and Michaël Timbert. HCODE: Hardware-enhanced real-time CFI. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, PPREW-4, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[85] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[86] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[87] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, 2014. USENIX Association.

[88] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013.

[89] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. In *Commun. ACM*, volume 5, pages 394–397, New York, NY, USA, July 1962. ACM.

[90] Ruan de Clercq, Ronald de Keulenaer, Pieter Maena, Bart Preneel, Bjorn De Sutter, and Ingrid Verbauwhede. SCM: Secure code memory architecture. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 771–776, New York, NY, USA, 2017. ACM.

[91] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[92] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of AES. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 7–15, Sept 2012.

[93] Stéphanie Delaune, Steve Kremer, Mark D. Ryan, and Graham Steel. Formal analysis of protocols based on TPM state registers. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 66–80, June 2011.

[94] Danny Dolev and Anbang Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, Mar 1983.

[95] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[96] Guillaume Duc and Ronan Keryell. Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 483–492, Dec 2006.

[97] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis, 2009.

[98] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. In *Journal of Computer Security*, volume 12, pages 247–311, Amsterdam, The Netherlands, The Netherlands, April 2004. IOS Press.

[99] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, Michel Bardouillet, and Albert Martinez. A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 506–509, July 2006.

[100] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, volume 12, pages 1–15, 2012.

[101] Adrienne Felt and David Wagner. Phishing on mobile devices. *Proceedings of 5th workshop Web 2.0 Security & Privacy*, 05 2012.

[102] Earlence Fernandes, Qi Alfred Chen, Georg Essl, Alex Halderman, Zhuoqing Morley Mao, and Atul Prakash. TIVOs: Trusted visual I/O paths for android. In *University of Michigan CSE Technical Report CSE-TR-586-14*, 2014.

[103] Atanas Filyanov, Jonathan M. McCuney, Ahmad-Reza Sadeghiz, and Marcel Winandy. Uni-directional trusted path: Transaction confirmation on just one device. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 1–12, June 2011.

[104] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12, pages 3–8, New York, NY, USA, 2012. ACM.

[105] Pierre-Alain Fouque, Reynald Lercier, Denis Réal, and Frédéric Valette. Fault attack on elliptic curve montgomery ladder implementation. In *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 92–98, Aug 2008.

[106] Anthony Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings of the First International Conference on Interactive Theorem Proving*, ITP'10, pages 243–258, Berlin, Heidelberg, 2010. Springer-Verlag.

[107] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings*, pages 175–188, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[108] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, Apr 2013.

[109] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 207–228, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[110] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *Advances in Cryptology – CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pages 444–461, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[111] Samaneh Ghandali, Georg T. Becker, Daniel Holcomb, and Christof Paar. A design methodology for stealthy parametric trojans and its application to bug attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 625–647, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[112] Christophe Giraud. DFA on AES. In *Advanced Encryption Standard – AES: 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers*, pages 27–41, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[113] Evgueni Goldberg, Mukul Prasad, and Robert Brayton. Using SAT for combinational equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '01, pages 114–121, Piscataway, NJ, USA, 2001. IEEE Press.

[114] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 182–194, New York, NY, USA, 1987. ACM.

[115] Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. Fast and constant-time implementation of modular exponentiation. *Embedded Systems and Communications Security, Niagara Falls, NY, US*, 01 2009.

[116] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15*, pages 77–91, 2002.

[117] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, pages 2:1–2:6, New York, NY, USA, 2017. ACM.

[118] Dahmun Goudarzi and Matthieu Rivain. On the multiplicative complexity of boolean functions and bitsliced higher-order masking. In *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 457–478, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[119] Eric Graves. Applying agile to hardware development. `https://www.playbookhq.co/blog/agileinhardwarenewproductdevelopment/`, 2016. [Online; accessed 10-November-2017].

[120] Shay Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. `http://eprint.iacr.org/2016/204`.

[121] Shay Gueron and Sanu Mathew. Hardware implementation of AES using area-optimal polynomials for composite-field representation $\mathrm{GF}(2^4)^2$ of $\mathrm{GF}(2^8)$. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 112–117, July 2016.

[122] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The seahorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015.

[123] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, May 2014.

[124] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991.

[125] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. In *Commun. ACM*, volume 52, pages 91–98, New York, NY, USA, May 2009. ACM.

[126] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Mar 2000.

[127] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[128] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *SIGPLAN Not.*, volume 48, pages 265–278, New York, NY, USA, March 2013. ACM.

[129] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. Modeling and enforcing secure object flows in process-driven SOAs: an integrated model-driven approach. *Software & Systems Modeling*, 13(2):513–548, May 2014.

[130] Gerard J. Holzmann. The model checker SPIN. In *IEEE Trans. Softw. Eng.*, volume 23, pages 279–295, Piscataway, NJ, USA, May 1997. IEEE Press.

[131] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. A case of system-level hardware/software co-design and co-verification of a commodity multi-processor system with custom hardware. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 513–520. ACM, 2012.

[132] Sungpack Hong, Tayo Oguntebi, Nathan Casper, J redand Bronson, Christos Kozyrakis, and Kunle Olukotun. A Case of System-level Hardware/Software Co-design and Co-verification of a Commodity Multi-processor System with Custom Hardware. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2012.

[133] Alex Horn, Michael Tautschnig, Celina Val, Lihao Liang, Tom Melham, Jim Grundy, and Daniel Kroening. Formal co-validation of low-level hardware/software interfaces. In *2013 Formal Methods in Computer-Aided Design*, pages 121–128, Oct 2013.

[134] Andrew Huang. Hacking the Xbox: an introduction to reverse engineering, 2002.

[135] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *Smart Card Research and Advanced Applications: 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, pages 219–235, Cham, 2014. Springer International Publishing.

[136] ICS-CERT. Hospira lifecare PCA infusion system vulnerabilities, advisory (ICSA-15-125-01B). https://ics-cert.us-cert.gov/advisories/ICSA-15-125-01B, june 2015.

[137] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3*, 9 2016.

[138] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain cross-VM attacks on Xen and VMware. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, pages 737–744, Dec 2014.

[139] Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. Unbounded symbolic execution for program verification. In *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, pages 396–411, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[140] Simon P. Johnson, Uday R. Savagaonkar, Vincent R. Scarlata, Francis X. McKeen, and Carlos V. Rozas. Technique for supporting multiple secure enclaves, 2012.

[141] Jan Jürjens. UMLsec: Extending UML for secure systems development. In *"UML" 2002 — The Unified Modeling Language: Model Engineering, Concepts, and Tools 5th International Conference Dresden, Germany, September 30 – October 4, 2002 Proceedings*, pages 412–425, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[142] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In *Journal of Computer Security*, volume 8, pages 141–158, Amsterdam, The Netherlands, The Netherlands, August 2000. IOS Press.

[143] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 21–21, Berkeley, CA, USA, 2003. USENIX Association.

[144] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. In *ACM Trans. Des. Autom. Electron. Syst.*, volume 4, pages 123–193, New York, NY, USA, April 1999. ACM.

[145] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter Van Der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 338–349, July 1997.

[146] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees Vissers. A methodology to design programmable embedded systems. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation — SAMOS*, pages 18–37, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[147] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, June 2014.

[148] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[149] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology — CRYPTO '96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[150] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, London, UK, UK, 1999. Springer-Verlag.

[151] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, New York, NY, USA, 2014. ACM.

[152] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. DDoS in the IoT: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[153] Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 393–404, Feb 2009.

[154] Joonho Kong, Farinaz Koushanfar, Praveen K. Pendyala, Ahmad=Reza Sadeghi, and Christian Wachsmann. PUFatt: Embedded platform attestation based on novel processor-based PUFs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[155] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. DAG-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer Science Review*, 13-14(Supplement C):1 – 38, 2014.

[156] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. New results for timing-based attestation. In *2012 IEEE Symposium on Security and Privacy*, pages 239–253, May 2012.

[157] Daniel Kroening and Natasha Sharygina. Formal verification of systemc by automatic hardware/software partitioning. In *Proceedings of the 2Nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '05, pages 101–110, Washington, DC, USA, 2005. IEEE Computer Society.

[158] Markus G. Kuhn. Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP. In *IEEE Trans. Comput.*, volume 47, pages 1153–1157, Washington, DC, USA, October 1998. IEEE Computer Society.

[159] Cynthia Kuo, Mark Luk, Rohit Negi, and Adrian Perrig. Message-in-a-bottle: User-friendly and secure key deployment for sensor nodes. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 233–246, New York, NY, USA, 2007. ACM.

[160] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 143–156, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[161] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *SIGPLAN Not.*, volume 47, pages 193–204, New York, NY, USA, June 2012. ACM.

[162] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, Dec 1997.

[163] Xavier Leroy. Formal verification of a realistic compiler. In *Commun. ACM*, volume 52, pages 107–115, New York, NY, USA, July 2009. ACM.

[164] Juncao Li, Fei Xie, Thomas Ball, Vladimir Levin, and Con McGarvey. An automata-theoretic approach to hardware/software co-verification. In *Fundamental Approaches to Software Engineering: 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 248–262, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[165] Letitia W. Li, Florian Lugou, and Ludovic Apvrille. Security-aware modeling and analysis for HW/SW partitioning. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017.*, pages 302–311, 2017.

[166] Letitia W Li, Florian Lugou, and Ludovic Apvrille. Security modeling for embedded system design. In *GraMSec 2017, 4th International Workshop on Graphical Models for Security*, Santa Barbara, CA, USA, August 2017.

[167] Letitia W. Li, Florian Lugou, and Ludovic Apvrille. Evolving attacker perspectives for secure embedded system design. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Portugal, January 22-24, 2018.*, 2018.

[168] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 8:1–8:7, New York, NY, USA, 2014. ACM.

[169] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 3–16, New York, NY, USA, 2011. ACM.

[170] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 168–177, New York, NY, USA, 2000. ACM.

[171] L. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. Introducing abuse frames for analysing security requirements. In *Proceedings. 11th IEEE International Requirements Engineering Conference, 2003.*, pages 371–372, Sept 2003.

[172] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *Cryptographic Hardware and Embedded Systems - CHES 2009: 11th International Workshop Lausanne, Switzerland, September 6-9, 2009 Proceedings*, pages 382–395, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[173] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622, Washington, DC, USA, 2015. IEEE Computer Society.

[174] Torsten Lodderstedt, David Basin, and Jürgen Doser. SecureUML: A UML-based modeling language for model-driven security. In *"UML" 2002 — The Unified Modeling Language: Model Engineering, Concepts, and Tools 5th International Conference Dresden, Germany, September 30 – October 4, 2002 Proceedings*, pages 426–441, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[175] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. SoC it to EM: Electromagnetic side-channel attacks on a complex system-on-chip. In *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 620–640, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[176] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems: Second International Workshop, TACAS '96 Passau, Germany, March 27–29, 1996 Proceedings*, pages 147–166, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[177] Florian Lugou, Ludovic Apvrille, and Aurélien Francillon. Toward a methodology for unified verication of hardware/software co-designs. In *PROOFS 2015, Security Proofs for Embedded Systems, 17 September 2015, Saint-Malo, France, Springer*, Saint-Malo, FRANCE, September 2015.

[178] Florian Lugou, Ludovic Apvrille, and Aurélien Francillon. SMASHUP: a toolchain for unified verification of hardware/software co-designs. *Journal of Cryptographic Engineering*, 7(1):63–74, Apr 2017.

[179] Florian Lugou, Letitia W. Li, Ludovic Apvrille, and Rabéa Ameur-Boulifa. SysML models and model transformation for security. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 331–338, Feb 2016.

[180] Martin Maas. PHANTOM: Practical oblivious computation in a secure processor. Master's thesis, EECS Department, University of California, Berkeley, May 2014.

[181] Fabio Massacci, John Mylopoulos, and Nicola Zannone. Computer-aided support for secure tropos. *Automated Software Engineering*, 14(3):341–364, Sep 2007.

[182] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *SIGOPS Oper. Syst. Rev.*, volume 42, pages 315–328, New York, NY, USA, April 2008. ACM.

[183] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model

for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[184] Kenneth L. McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60, Boston, MA, 1993. Springer US.

[185] Nancy R Mead and Ted Stehney. *Security quality requirements engineering (SQUARE) methodology*, volume 30. ACM, 2005.

[186] Catherine Meadows. The NRL protocol analyzer: An overview. *The Journal of Logic Programming*, 26(2):113 – 131, 1996.

[187] Ralph C. Merkle. Protocols for public key cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122, April 1980.

[188] Jonathan K Millen. CAPSL: Common authentication protocol specification language. In *NSPW*, volume 96, page 132, 1996.

[189] Joaquin Miller and Jishnu Mukerji. Mda guide version 1.0. 1, object management group. *URL: http://www.omg.org/docs/omg/03-06-01.pdf*, 2003.

[190] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to combinational equivalence checking. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 836–843, Nov 2006.

[191] Leonardo De Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *In Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, pages 244–251, 2002.

[192] Rajdeep Mukherjee, Daniel Kroening, Tom Melham, and Mandayam Srivas. Equivalence checking using trace partitioning. In *2015 IEEE Computer Society Annual Symposium on VLSI*, pages 13–18, July 2015.

[193] Rajdeep Mukherjee, Mitra Purandare, Raphael Polig, and Daniel Kroening. Formal techniques for effective co-verification of hardware/software co-designs. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 35:1–35:6, New York, NY, USA, 2017. ACM.

[194] Yuichi Nakamura, Kouhei Hosokawa, Ichiro Kuroda, Ko Yoshikawa, and Takeshi Yoshimura. A fast hardware/software co-verification method for systern-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication. In *Proceedings. 41st Design Automation Conference, 2004.*, pages 299–304, July 2004.

[195] Armstrong Nhlabatsi, Bashar Nuseibeh, and Yijun Yu. Security requirements engineering for evolving software systems: A survey. In *Int. J. Secur. Softw. Eng.*, volume 1, pages 54–73, Hershey, PA, USA, January 2010. IGI Global.

[196] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security: 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006. Proceedings*, pages 529–545, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[197] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[198] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 479–498, Washington, D.C., 2013. USENIX.

[199] Ebenezer A. Oladimeji. Security threat modeling and analysis: A goal-oriented approach, 2006.

[200] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology – CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[201] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. In *IACR Cryptology ePrint Archive*, volume 2002, page 169, 01 2002.

[202] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, Washington, D.C., 2013. USENIX.

[203] Lawrence C Paulson. The inductive approach to verifying cryptographic protocols. *Journal of computer security*, 6(1-2):85–128, 1998.

[204] Gabriel Pedroza, Ludovic Apvrille, and Daniel Knorreck. AVATAR: A SysML environment for the formal verification of safety and security properties. In *2011 11th Annual International Conference on New Technologies of Distributed Systems*, pages 1–10, May 2011.

[205] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing*, Cloud Computing '13, pages 3–10, New York, NY, USA, 2013. ACM.

[206] Jonathan M McCune Adrian Perrig and Michael K Reiter. Safe passage for passwords and other sensitive data. In *Proceeding of the 16th Annual Network and Distributed System Security Symposium*, 2009.

[207] Christophe Ponsard, Gautier Dallons, and Philippe Massonet. Goal-oriented co-engineering of security and safety requirements in cyber-physical systems. In *International Conference on Computer Safety, Reliability, and Security*, pages 334–345. Springer, 2016.

[208] Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[209] Jürgen Pulkus and Srinivas Vivek. Reducing the number of non-linear multiplications in masking schemes. In *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 479–497, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[210] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Smart Card Programming and Security: International Conference on Research in Smart Cards, E-smart 2001 Cannes, France, September 19–21, 2001 Proceedings*, pages 200–210, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[211] Ramaswamy Ramanujam and SP Suresh. Tagging makes secrecy decidable with unbounded nonces as well. In *FSTTCS*, volume 3, pages 363–374. Springer, 2003.

[212] Awais Rashid, Syed Asad Ali Naqvi, Rajiv Ramdhany, Matthew Edwards, Ruzanna Chitchyan, and Muhammad Ali Babar. Discovering unknown known security requirements. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 866–876, May 2016.

[213] Yanting Ren, An Wang, and Liji Wu. Transient-steady effect attack on block ciphers. In *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 433–450, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[214] Jenni Susan Reuben. A survey on virtual machine security. *Helsinki University of Technology*, 2(36), 2007.

[215] Pilar Rodríguez, Jouni Markkula, Markku Oivo, and Kimmo Turula. Survey on agile and lean usage in finnish software industry. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 139–148, New York, NY, USA, 2012. ACM.

[216] Pascal Roques. MBSE with the Arcadia method and the Capella tool. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[217] Yves Roudier and Ludovic Apvrille. SysML-Sec: A model driven approach for designing safe and secure systems. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 655–664, Feb 2015.

[218] Mattia Salnitri, Fabiano Dalpiaz, and Paolo Giorgini. Modeling and verifying security policies in business processes. In *Enterprise, Business-Process and Information Systems Modeling*, pages 200–214. Springer, 2014.

[219] Hamid Savoj, Alan Mishchenko, and Robert Brayton. Sequential equivalence checking for clock-gated circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(2):305–317, Feb 2014.

[220] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 78–94, June 2012.

[221] Jörn-Marc Schmidt, Michael Hutter, and Thomas Plos. Optical fault attacks on AES: A threat in violet. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 13–22, Sept 2009.

[222] Steffen Schulz, Ahmad-Reza Sadeghi, and Christian Wachsmann. Short paper: Lightweight remote attestation using physical functions. In *Proceedings of the Fourth ACM Conference on Wireless Network Security*, WiSec '11, pages 109–114, New York, NY, USA, 2011. ACM.

[223] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[224] Luc Semeria and Abhijit Ghosh. Methodology for hardware/software co-verification in C/C++. In *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, pages 405–408, June 2000.

[225] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[226] Arvind Seshadri, Mark Luk, and Adrian Perrig. SAKE: Software attestation for key establishment in sensor networks. In *Distributed Computing in Sensor Systems: 4th IEEE International Conference, DCOSS 2008 Santorini Island, Greece, June 11-14, 2008 Proceedings*, pages 372–385, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[227] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doom, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Malware Detection*, pages 253–289, Boston, MA, 2007. Springer US.

[228] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM Workshop on Wireless Security*, WiSe '06, pages 85–94, New York, NY, USA, 2006. ACM.

[229] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 272–282, May 2004.

[230] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

[231] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote software-based attestation for wireless sensors. In *Security and Privacy in Ad-hoc and Sensor Networks: Second European Workshop, ESAS 2005, Visegrad, Hungary, July 13-14, 2005. Revised Selected Papers*, pages 27–41, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[232] Umesh Shankar, Monica Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[233] Dillibabu Shanmugam, Ravikumar Selvam, and Suganya Annadurai. Differential power analysis attack on SIMON and LED block ciphers. In *Security, Privacy, and Applied Cryptography Engineering: 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings*, pages 110–125, Cham, 2014. Springer International Publishing.

[234] Gang Shen, Xiaohong Li, Ruitao Feng, Guangquan Xu, Jing Hu, and Zhiyong Feng. An extended UML method for the verification of security protocols. In *2014 19th International Conference on Engineering of Complex Computer Systems*, pages 19–28, Aug 2014.

[235] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 317–328, New York, NY, USA, 2016. ACM.

[236] Shweta Shinde, Shruti Tople, Deepak Kathayat, and Prateek Saxena. PODARCH: Protecting legacy applications with a purely hardware TCB. In *National University of Singapore, Tech. Rep*, 2015.

[237] Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, Jan 2005.

[238] Igor Smolyar, Muli Ben-Yehuda, and Dan Tsafrir. Securing self-virtualizing ethernet devices. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[239] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[240] Michael Steil and Felix Domke. The Xbox 360 security system and its weaknesses. `https://www.youtube.com/watch?v=uxjpmc8ZIxM`, 2008. [Online; accessed 23-January-2018].

[241] Pramod Subramanyan and Divya Arora. Formal Verification of Taint-Propagation Security Properties in a Commercial SoC Design. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.

[242] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. Template-based synthesis of instruction-level abstractions for SoC verification. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 160–167, Austin, TX, 2015. FMCAD Inc.

[243] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[244] Husam Suleiman and Davor Svetinovic. Evaluating the effectiveness of the security quality requirements engineering (SQUARE) method: a case study using smart grid advanced metering infrastructure. *Requirements Engineering*, 18(3):251–279, Sep 2013.

[245] Robert A. Thacker, Chris J. Myers, Kevin Jones, and Scott R. Little. A new verification method for embedded systems. In *2009 IEEE International Conference on Computer Design*, pages 193–200, Oct 2009.

[246] Kris Tiri and Ingrid Verbauwhede. Securing encryption algorithms against DPA at the logic level: Next generation smart card technology. In *Cryptographic Hardware and Embedded Systems - CHES 2003: 5th International Workshop, Cologne, Germany, September 8–10, 2003. Proceedings*, pages 125–136, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[247] Victor Tomashevich, Yaara Neumeier, Raghavan Kumar, Osnat Keren, and Ilia Polian. Protecting cryptographic hardware against malicious attacks by nonlinear robust codes. In *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 40–45, Oct 2014.

[248] Tianhao Tong and David Evans. Guardroid: A trusted path for password entry. *Proceedings of Mobile Security Technologies (MoST)*, 2013.

[249] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *Cryptographic Hardware and Embedded Systems - CHES 2003: 5th International Workshop, Cologne, Germany, September 8–10, 2003. Proceedings*, pages 62–76, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[250] Axel Van Lamsweerde et al. Engineering requirements for system reliability and security. *NATO Security Through Science Series D-Information and Communication Security*, 9:196, 2007.

[251] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M Swift. Scheduler-based defenses against cross-VM side-channels. In *USENIX Security Symposium*, pages 687–702, 2014.

[252] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In *Advances in Cryptology – ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, pages 740–757, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[253] Jorgiano Vidal, Florent de Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 226–231, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[254] Carlos Villarraga, Bernard Schmidt, Binghao Bao, Rakesh Raman, Christian Bartsch, Thomas Fehmel, Dominik Stoffel, and Wolfgang Kunz. Software in a hardware view: New models for HW-dependent software in SoC verification and test. In *2014 International Test Conference*, pages 1–9, Oct 2014.

[255] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395, May 2010.

[256] Thomas Weigold, Thorsten Kramp, Reto Hermann, Frank Höring, Peter Buhler, and Michael Baentsch. The zurich trusted information channel – an efficient defence against man-in-the-middle and malicious software attacks. In *Trusted Computing - Challenges and Applications: First International Conference on Trusted Computing and Trust in Information Technologies, Trust 2008 Villach, Austria, March 11-12, 2008 Proceedings*, pages 75–91, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[257] Samuel Weiser and Mario Werner. SGXIO: Generic trusted I/O path for Intel SGX. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 261–268, New York, NY, USA, 2017. ACM.

[258] Ken Whitaker. Yes, you can develop embedded software using agile methodology. `https://www.embedded.com/design/prototyping-and-development/4441737/1/Yes--you-can-develop-embedded-software-using-agile-methodology-`, 2016. [Online; accessed 10-November-2017].

[259] Dan Williams and Emin Gun Sirer. Optimal parameter selection for efficient memory integrity verification using merkle hash trees. In *Network Computing and Applications, 2004.(NCA 2004). Proceedings. Third IEEE International Symposium on*, pages 383–388. IEEE, 2004.

[260] Rafal Wojtczuk and Joanna Rutkowska. Attacking intel trusted execution technology. *Black Hat DC*, 2009, 2009.

[261] Yubin Xia, Yutao Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, June 2012.

[262] Fei Xie, Xiaolu Song, Hung Chung, and Ranajoy Nandi. Translation-based co-verification. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.*, pages 111–120, July 2005.

[263] Fei Xie, Guowu Yang, and Xiaoyu Song. Component-based hardware/software co-verification for building trustworthy embedded systems. In *J. Syst. Softw.*, volume 80, pages 643–654, New York, NY, USA, May 2007. Elsevier Science Inc.

[264] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society.

[265] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 351–, Washington, DC, USA, 2003. IEEE Computer Society.

[266] Jiewen Yao and Vincent J Zimmer. White paper a tour beyond BIOS launching a STM to monitor SMM in EFI developer kit II. Technical report, Intel, 2015.

[267] Miao Yu, Virgil D. Gligor, and Zongwei Zhou. Trusted display on untrusted commodity platforms. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 989–1003, New York, NY, USA, 2015. ACM.

[268] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 29–40, New York, NY, USA, 2011. ACM.

[269] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA, 2013. IEEE Computer Society.

[270] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., 2013. USENIX.

[271] Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. Anomalous path detection with hardware support. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '05, pages 43–54, New York, NY, USA, 2005. ACM.

[272] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

[273] Yu Zhang, Fei Xie, Yunwei Dong, Xingshe Zhou, and Chunyan Ma. Cyber/physical co-verification for developing reliable cyber-physical systems. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 539–548, July 2013.

[274] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. In *2012 IEEE Symposium on Security and Privacy*, pages 616–630, May 2012.

[275] Zongwei Zhou, Miao Yu, and Virgil D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *2014 IEEE Symposium on Security and Privacy*, pages 308–323, May 2014.

[276] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: An infrastructure for efficiently protecting information leakage on the address bus. In *SIGOPS Oper. Syst. Rev.*, volume 38, pages 72–84, New York, NY, USA, October 2004. ACM.