UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS

LABORATOIRE DES SYSTEMES EMBARQUES CRITIQUES (LSEC), THALES RESEARCH AND TECHNOLOGY

THÈSE présentée par :

Yani ZIANI

soutenue le : **9 octobre 2025**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/ Spécialité : **Informatique**

Vérification formelle de couches de confiance dans les logiciels : application à la TPM Software Stack

THÈSE dirigée par :

M. LOULERGUE Frédéric	Professeur, Université d'Orléans
M. KOSMATOV Nikolaï	Ingénieur-chercheur, Thales

RAPPORTEURS :

Mme. DUBOIS Catherine	Professeure, ENSIIE
M. GIORGETTI Alain	Professeur, Université Marie et Louis Pasteur

JURY :

Mme. DUBOIS Catherine	Professeure, ENSIIE
M. GIORGETTI Alain	Professeur, Université Marie et Louis Pasteur
M. AHRENDT Wolfgang	Professeur, Chalmers University of Technology
M. MARCHÉ Claude	Directeur de recherche, INRIA Saclay, Président du jury

Acknowledgements

First, I would like to thank my supervisors Nikolaï Kosmatov and Frédéric Loulergue, for their invaluable support, as well as for all the interest they showed in my work throughout these three (and a half) years. I also thank them for their kindness, availability and their guidance.

Next, I would like to thank my referees, I would like to thank Catherine Dubois and Alain Giorgetti for agreeing to (thoroughly) review my thesis, and for giving me very detailed and extremely helpful feedback. I would also like to thank the rest of my fantastic jury, Claude Marché and Wolfgang Ahrendt, who took the time to examine my work, read my thesis and attend my defense; and to thank Claude Marché for chairing my thesis committee.

In particular, I would like to thank the jury as a whole for their comments and questions during my defense, which have allowed me to improve this manuscript and prepare for future work.

I would also like to thank Daniel Gracia Perez for the invaluable help he gave me in understanding how the TPM and its Software Stack work and what they do in the context of my work.

A big thank to all the other members of the CES lab/ LSEC at Thales their warm welcome and their willingness to share knowledge (and sometimes, food and stuff). I would like to thank in particular Téo B. for the serious, and the not-so-serious-but-still-very-enjoyable discussions we've had, as well as *the other* Téo B. for exactly the same reasons. I would also like to thank members of other labs in adjacent corridors at Thales, as well as anyone else I might have talked to next to the coffee machine during breaks, but whose names would be too numerous to list here.

Thank you to the members of the Frama-C team for the invaluable help they gave me in using their tool, taking care of my issues, and helping understand and track down problems, as well as working around them.

Last but not least, I would like to thank my parents, my family and my friends for supporting and encouraging me in my choice, and most importantly, for putting up with me whenever I started thinking out loud and/or babbling about "incoherent niche nonsense" for too long.

Résumé étendu en français

Introduction

Vérification de logiciels

Au cours des dernières décennies, les systèmes informatiques, qu'il s'agisse d'ordinateurs personnels, de téléphones ou de systèmes embarqués, sont devenus de plus en plus complexes et intégrés dans notre société. Nous les utilisons constamment dans notre vie quotidienne, que ce soit dans les systèmes grand public (tels que les ordinateurs personnels et les smartphones), ou dans les systèmes industriels. Qu'il s'agisse de systèmes non critiques tels que applications bancaires mobiles et les appareils ménagers, ou de systèmes critiques comme les systèmes de contrôle aériens, les systèmes de contrôle des chemins de fer, ou les appareils et dispositifs médicaux, la plupart de nos activités reposent aujourd'hui fortement sur les systèmes informatiques, qui gèrent ainsi de plus en plus d'aspects critiques de nos vies.

Ces systèmes, en raison de leur caractère important, exigent souvent un degré élevé d'exactitude et de fiabilité. Malgré cela, nous sommes tous trop habitués à rencontrer, à signaler et à trouver des moyens de contourner les bugs des logiciels, dans la mesure où les humains qui conçoivent ces systèmes sont condamnés à faire des erreurs. C'est pourquoi la vérification du logiciel est si importante et peut s'avérer absolument critique.

Dans les applications moins critiques, l'approche standard de la vérification des logiciels qui a fait ses preuves est le test. Lorsqu'ils sont effectués correctement, les tests constituent généralement une solution appropriée, suffisante pour détecter non pas tous les bugs, mais la plupart des bugs présents dans les systèmes logiciels, et par conséquent suffisante pour garantir un niveau de fonctionnement adéquat.

Néanmoins, une telle approche n'est pas toujours suffisante. Dans les systèmes industriels et autres applications critiques qui peuvent être responsables

de la sécurité d'informations personnelles ou d'informations sensibles, ou de la sécurité de vies humaines, un niveau de confiance et un degré d'exactitude beaucoup plus élevés sont nécessaires.

Vérification formelle

Vérification déductive

La vérification déductive [69] est une technique puissante de vérification statique modulaire. Le principe général de cette approche est d'exprimer la correction d'un programme sous la forme d'un énoncé mathématique logique, puis de construire une preuve de sa validité. De tels énoncés sont appelés spécifications et sont utilisés pour générer des obligations de preuve (ou conditions de vérification), dont la preuve implique la conformité du programme à la spécification. Les conditions de vérification peuvent être prouvées à l'aide de prouveurs automatiques (tels que les solveurs SMT comme *Alt-Ergo* [41] ou *CVC4* [9]) ou de prouveurs de théorèmes interactifs (tels que Coq/Rocq [108] ou *Isabelle/HOL* [16]).

La vérification déductive d'un programme est modulaire : chaque fonction d'un programme est vérifiée indépendamment des autres. Pour chaque fonction, nous utilisons sa spécification pour générer des obligations de preuve qui sont suffisamment fortes pour garantir que la fonction est conforme à sa spécification.

Si la fonction contient des appels à d'autres fonctions, nous utilisons les spécifications des fonctions appelées comme hypothèse pour les preuves de la spécification de l'appelant : dans les programmes de grande taille, il est donc préférable de vérifier d'abord les fonctions appelées avant de s'intéresser à la preuve des spécifications des fonctions appelantes.

Cette technique est correcte et complète, mais elle n'est généralement pas automatisable et nécessite en général la rédaction manuelle de diverses annotations appropriées pour guider la preuve, afin de mener à bien la preuve de certaines conditions de vérification.

Vérification d'assertions à l'exécution (Runtime assertion checking)[1]

La vérification à l'exécution [73, 57], ou vérification d'assertion à l'exécution, est une technique de vérification dynamique visant à traiter les problèmes à la frontière entre la vérification formelle et le test. Il s'agit d'une technique de vérification formelle *légère*, dont le principe général est de vérifier que certaines propriétés d'un programme sont maintenues à l'exécution, c'est-à-dire pendant l'exécution d'un programme cible. Elle vise à détecter et à signaler les erreurs en temps réel, dès qu'elles se produisent pendant l'exécution.

La vérification à l'exécution se concentre sur des exécutions individuelles complètes d'un programme, et peut servir d'alternative plus efficace mais moins rigoureuse aux méthodes de vérification statique dans les situations qui peuvent s'avérer trop complexes pour celles-ci. Elle offre en effet des garanties de correction plus faibles que les méthodes plus rigoureuses, car cette méthode *évalue des exécutions individuelles* de programmes, alors que des méthodes telles que la vérification déductive *prouvent* des propriétés pour *toutes* les exécutions du programme.

Interprétation abstraite

L'interprétation abstraite [43] est une technique d'analyse statique. Son principe général consiste à faire correspondre les variables d'un programme à des domaines abstraits qui sur-approximent l'ensemble des valeurs concrètes que ces variables peuvent prendre au cours d'une exécution possible du programme. Par exemple, les variables de type entier peuvent être abstraites avec un intervalle. Cette méthode est robuste et appropriée pour l'automatisation de preuves, mais elle n'est pas complète, car des approximations excessives peuvent provoquer de fausses alertes et la détection de problèmes non-existants dans des programmes pourtant corrects.

À cet effet, et parce que cette méthode repose sur une sur-approximation des variables d'un programme, l'interprétation abstraite est plus automatisable

[1] Le terme historique "runtime assertion checking"/"vérification d'assertions à l'exécution" pourrait être généralisé aujourd'hui par "runtime annotation checking"/"vérification d'annotations à l'exécution", car on peut vérifier tout type d'annotation.

et nécessite moins d'annotations intermédiaires que la vérification déductive, mais elle est moins précise dans ses résultats. De même, elle offre un niveau de confiance plus élevé dans les résultats de la vérification que la vérification d'assertions à l'exécution, mais nécessite souvent un effort de spécification plus important.

Vérification de modèle

Bien que nous n'utilisions pas la vérification de modèles dans cette thèse, nous en fournissons ici une description sommaire par souci d'exhaustivité. La vérification de modèles logicielle est une technique de vérification logicielle qui vise à vérifier un programme (ou un modèle abstrait de celui-ci) en explorant complètement l'ensemble des états possibles que le programme peut atteindre lors de son exécution.

Les spécifications possibles pour les propriétés du programme ne se limitent pas nécessairement à un ensemble d'états incorrects à exclure, et elles sont généralement exprimées en utilisant la logique temporelle. Étant donné que l'ensemble des états peut être très important, il peut être impossible d'explorer tous les états potentiels; dans ce cas, des méthodes d'abstraction sont utilisées pour construire un modèle approximatif du système en utilisant seulement un nombre fini d'états.

Contributions et structure du document

L'objectif principal de cette thèse est d'évaluer dans quelle mesure il est actuellement possible de vérifier formellement (ou semi-formellement) des propriétés de sûreté, des propriétés fonctionnelles et des propriétés de sécurité dans des codes réels critiques qui n'ont pas été conçus pour la vérification.

On s'intéresse ici notamment au Trusted Platform Module (TPM), un cryptoprocesseur conçu pour protéger l'intégrité et la sécurité des ordinateurs modernes. On se concentre plus particulièrement sur la TPM Software Stack (TSS), qui joue le rôle de couche de confiance, et par laquelle passent les communications avec le TPM.

On s'intéressera dans cette thèse à la bibliothèque open-source TPM2-TSS (écrite en C), une implémentation largement utilisée de la TSS. En tant qu'implémentation populaire de la TPM Software Stack, et en tant que pont entre le TPM et les applications, cette bibliothèque est fortement critique, et son code est très complexe et constitue un défi pour les outils de vérification.

Parmi ces outils, on s'intéresse notamment à la plateforme FRAMA-C [74], qui fournit un environnement pour l'analyse et la vérification des programmes ANSI/ISO C. La plateforme est construite autour d'un noyau fournissant des services de base pour l'analyse de codes sources, permettant aux utilisateurs d'effectuer différents types d'analyse et de vérification des programmes C grâce à divers greffons. Dans cette thèse, nous nous appuyons principalement sur le greffons WP [13] pour la vérification déductive, le greffon E-ACSL [73] pour la vérification à l'exécution, et le greffon METACSL [99] pour la définition d'exigences de haut niveau (qui sont particulièrement utiles pour exprimer les propriétés de sécurité telles que l'intégrité et la confidentialité). Nous utilisons également l'outil d'interprétation abstraite MEMCAD [104].

Contributions

Les contributions principales de cette thèse sont les suivantes :

- La spécification et la preuve, dans FRAMA-C/WP, de propriétés de sûreté et de propriétés fonctionnelles de bas niveau, sur un exemple illustratif simple, et sur un sous-ensemble représentatif de fonctions de TPM2-TSS, une bibliothèque C critique qui n'a pas été conçue pour être vérifiée formellement. Nous présentons, à l'aide d'un exemple illustratif, les principaux problèmes que nous avons rencontrés lors de la vérification, et nous décrivons les solutions et les palliatifs temporaires que nous avons proposées. Nous fournissons les preuves COQ pour certains des lemmes nécessaires à la manipulation des listes chaînées. Ces contributions sont présentées dans le chapitre 4.

  Ces contributions sont associées à un artéfact (disponible en ligne à `https://doi.org/10.5281/zenodo.13693011`) qui illustre tous les

problèmes que nous avons rencontrés et les solutions que nous avons proposées sur les fonctions du code réel, annotées en ACSL et entièrement prouvées avec Frama-C/Wp (ainsi que tous les lemmes nécessaires à la vérification et leurs preuves). Cet artefact a reçu les badges EAPLS Availability (Disponible) et Reusability (Réutilisable) lors de la 18ème édition de l'*International Conference on integrated Formal Methods* (iFM 2023).

- Une approche de modélisation reposant sur des structures de données spécifiques, utilisable pour la spécification et la vérification de propriétés de sécurité de haut niveau. Nous spécifions et vérifions à l'exécution, dans Frama-C/E-ACSL, des propriétés de sécurité de haut niveau sur un sous-ensemble de fonctions simplifié représentatif de tpm2-tss. Nous présentons les principaux problèmes rencontrés et évaluons les résultats de la vérification, dont la détection d'erreurs injectées et intentionnellement introduites dans le code. Ces contributions sont présentées dans le chapitre 5.

  Ces contributions sont associées à un artefact compagnon disponible en ligne à `https://doi.org/10.5281/zenodo.12773113`, qui contient le code, les outils, les définitions des propriétés de haut niveau vérifiées, ainsi que les instructions nécessaires pour reproduire nos résultats de vérification. Cet artefact a reçu les badges EAPLS Availability (Disponible) et Functionality (Fonctionnel) lors de la 18ème *International Conference on Tests and Proofs* (TAP 2024).

- Des travaux préliminaires portant sur la combinaison de la vérification déductive avec d'autres techniques comme alternative aux palliatifs sur lesquels nous nous appuyons dans les précédentes contributions. Nous proposons la combinaison de Frama-C/Wp avec l'outil d'analyse de forme MemCAD (qui s'appuie sur l'interprétation abstraite), que nous appliquons aux fonctions manipulant des listes chaînées du chapitre 4, ainsi qu'à une étude de cas réussie sur un ensemble de fonctions de la bibliothèque tpm2-tss. Nous explorons également la combinaison possible de la vérification déductive avec la vérification d'assertion à l'exécution,

pour la vérification des propriétés de haut niveau. Ces contributions sont présentées dans le chapitre 6.

Les contributions portant sur la combinaison de Frama-C/Wp avec MemCAD sont associées à un artefact compagnon disponible en ligne à `http://doi.org/10.5281/zenodo.10497923`, qui comprend une machine virtuelle contenant les outils installés et les exemples de code utilisés. Cet artefact peut être utilisé pour reproduire les résultats présentés dans la Section 6.1. Cet artefact a reçu les deux badges EAPLS Availability (Disponible) et Functionality (Fonctionnel) lors de la 27ème *International Conference on Fundamental Approaches to Software Engineering* (FASE 2024).

Publications et autres réalisations

La préparation de cette thèse a compris la soumission et la publication de plusieurs articles dans des conférences et des revues évaluées par des pairs.

- "Towards Formal Verification of a TPM Software Stack" [118], paru à la 18ème *International Conference on integrated Formal Methods* (iFM 2023). Cette publication présente notre travail sur la vérification déductive de propriétés fonctionnelles et de propriétés de sûreté de bas niveau, présenté dans le chapitre 4 de cette thèse, et nominé candidat au prix du meilleur article pour cette conférence.

- "Runtime Verification for High-Level Security Properties: Case Study on the TPM Software Stack" [119], paru à la 18ème *International Conference on Tests and Proofs* (TAP 2024). Cette publication présente nos travaux sur la vérification à l'exécution de propriétés de sécurité de haut niveau présentés dans le chapitre 5.

- "Combining Deductive Verification with Shape Analysis" [17], paru à la 27ème *International Conference on Fundamental Approaches to Software Engineering* (FASE 2024). Cette publication de nos travaux présentés dans la Section 6.1 porte sur la combinaison de techniques de vérification pour

pallier certaines des limitations de la vérification déductive avec Frama-C/Wp identifiées dans le chapitre 4.

- "Towards Formal Verification of a TPM Software Stack: Achievements and Opportunities" [120], accepté pour publication dans le journal *Formal Aspects of Computing* (FAC). Cette publication est une version étendue de l'article publié à la conférence iFM 2023.
- Trois artefacts compagnons (décrits ci-dessus dans le paragraphe Contributions) qui contiennent tous les fichiers des études de cas réalisées, et permettent aux lecteurs de reproduire nos résultats de vérification.

Plan

Le reste de cette thèse est structuré comme suit :

- Le chapitre 1 correspond à l'introduction de ce manuscrit en anglais.
- Le chapitre 2 présente les préliminaires : le TPM, sa pile logicielle (la TPM Software Stack) et la bibliothèque tpm2-tss, ainsi que Frama-C, le langage ACSL, et les plugins Wp, E-ACSL et MetAcsl. Nous fournissons des exemples détaillant l'utilisation de nombreuses clauses ACSL, ainsi qu'une brève présentation des fonctionnalités de chaque greffon.
- Le chapitre 3 présente l'état de l'art concernant la fiabilité et la sécurité du TPM et de composants sécurisés similaires, ainsi que l'état de l'art actuel portant sur la vérification formelle de codes réels.
- Le chapitre 4 présente notre travail sur la preuve de propriétés de sûreté et de propriétés fonctionnelles dans du code critique de la vie réelle en utilisant le greffon Wp pour la vérification déductive, en se concentrant sur un sous-ensemble de fonctions de la bibliothèque tpm2-tss en charge d'opérations internes importantes.
- L'objectif du chapitre 5 est de proposer une approche de validation alternative à la vérification déductive (avec Wp), ainsi qu'une méthodologie préliminaire de vérification pour des propriétés de sécurité de haut

niveau d'abstraction dans du code critique réel comme TPM2-TSS, en utilisant le greffon E-ACSL pour la vérification, et le greffon METACSL pour la définition de ces exigences.

- Dans le chapitre 6, nous présentons nos travaux préliminaires sur la combinaison de techniques de validation pour aider à atténuer certaines des limites que nous avons identifiées avec la vérification déductive de WP dans le chapitre 4, et avec la vérification d'assertions à l'exécution de E-ACSL pour les exigences de haut niveau dans le chapitre 5. Nous nous concentrons ici sur deux combinaisons : celle de FRAMA-C/WP avec MEM-CAD, un outil d'analyse de forme, pour la vérification des propriétés fonctionnelles et de sûreté de bas niveau, et celle de WP avec E-ACSL, pour la vérification des propriétés de sécurité de haut niveau.
- Le chapitre 7 conclut ce manuscrit et présente quelques perspectives.

CONCLUSION

Dans cette thèse, nous avons étudié dans quelle mesure il est actuellement possible de vérifier formellement (ou semi-formellement) un code critique réel qui n'a ni été conçu pour la vérification, ni été développé avec l'assistance de la vérification formelle. Nous nous sommes concentrés sur trois types principaux de propriétés : les propriétés de sûreté de bas niveau et les propriétés fonctionnelles de bas niveau (dans le chapitre 4 et la Section 6.1), et les propriétés de sécurité de haut niveau (dans le chapitre 5 et la Section 6.2). Nous avons identifié ce qui est actuellement traitable par la vérification déductive et la vérification d'assertion à l'exécution via la plateforme de vérification FRAMA-C, les solutions temporaires pour traiter les limitations que nous avons rencontrées, ainsi que certaines améliorations nécessaires pour mener des efforts de vérification exhaustifs et complets sur de tels codes à l'avenir. En effet, le code de la bibliothèque est très différent des codes industriels plus "traditionnels" : il est très complexe et représente un défi pour les outils de vérification tels que FRAMA-C.

Dans le chapitre 4, nous avons présenté nos résultats de vérification pour un sous-ensemble de 10 fonctions de la couche ESAPI de TPM2-TSS que nous avons vérifié avec WP. Les propriétés vérifiées comprennent à la fois des propriétés fonctionnelles de bas niveau et l'absence d'erreurs à l'exécution (telles que les pointeurs invalides et les dépassements de mémoire, qui conduisent souvent à des vulnérabilités en matière de sécurité). Nous avons décrit les limitations actuelles des outils et les solutions temporaires que nous avons utilisées pour y remédier. Nous avons prouvé tous les lemmes nécessaires (étendant ceux d'une étude de cas précédente pour les listes chaînées [22]) dans COQ. Enfin, nous avons identifié les améliorations à apporter à l'outil pour réaliser une vérification formelle complète de la bibliothèque : le support des allocations dynamiques et des clauses ACSL associées, un modèle de mémoire qui permet la gestion des opérations au niveau de l'octet, et une séparation plus claire des statuts de modification des variables entre les différents segments de la mémoire du programme.

Dans le chapitre 5, nous avons présenté un travail préliminaire portant sur la manière de définir et de vérifier, à l'exécution, des propriétés de sécurité sur des données sensibles dans des codes de la vie réelle comme la bibliothèque TPM2-TSS. Dans la mesure où il s'agit de la couche de communication entre le TPM et les plateformes ou les applications hôtes, il est primordial de pouvoir assurer la sécurité de données sensibles que l'ont souhaite stocker sur le TPM. Il est important de garantir que ces données ne peuvent jamais être récupérées sans autorisation lorsqu'elles passent par la pile logicielle. Nous avons présenté nos résultats de vérification pour un sous-ensemble de 86 fonctions, sur lesquelles nous avons vérifié, à l'exécution, que les données sensibles cibles (par exemple, une clé de chiffrement que l'on souhaite importer dans le TPM) ne sont jamais modifiées ou lues lorsqu'elles ne sont pas censées l'être. Nous avons décrit certaines limitations des outils, ainsi que les solutions temporaires que nous avons utilisées pour y remédier. Le sous-ensemble de fonctions a été légèrement simplifié pour supprimer les dépendances aux bibliothèques externes, mais nous avons maintenu le comportement logique du code.

Dans le chapitre 6, nous avons proposé deux approches préliminaires re-

posant sur la combinaison de techniques de validation pour aider à atténuer certaines des limitations mises en évidence par notre travail dans les deux chapitres précédents. Nous nous sommes concentrés sur deux combinaisons : dans la Section 6.1, nous avons étudié la combinaison de Frama-C/Wp avec MemCAD [104] pour la vérification de propriétés de sûreté de bas niveau et de propriétés fonctionnelles de bas niveau (étendant ainsi les travaux de vérification présentés dans le chapitre 4); dans la Section 6.2, la combinaison de Wp avec E-ACSL, pour la vérification de propriétés de sécurité de haut niveau (en extension aux travaux présentés dans le chapitre 5). Dans la Section 6.1, les propriétés de séparation et les invariants structurels pour les structures de données récursives (telles que les listes chaînées) sont plus facilement prouvés par MemCAD, que nous avons ensuite utilisé comme hypothèses dans Wp, permettant ainsi à ce dernier de se concentrer sur la preuve d'autres propriétés. Dans la Section 6.2, nous avons adopté une approche similaire, en nous appuyant sur Wp pour prouver les propriétés de sécurité (ainsi que des propriétés de sûreté intermédiaires et des propriétés fonctionnelles intermédiaires) dans les fonctions où les preuves sont plus facilement automatisables.

Perspectives

Ce travail ouvre plusieurs possibilités d'extension. Dans cette section, nous évoquons quelques pistes de recherche intéressantes qui s'appuient sur les travaux présentés dans cette thèse.

Vérification déductive de propriétés de sûreté et de propriétés fonctionnelles

Le travail présenté dans le chapitre 4 ouvre la perspective vers un travail de vérification complet de propriétés de bas niveau dans la bibliothèque tpm2-tss, (ou des API de bibliothèques similaires critiques pour la sécurité).

Les travaux futurs comprennent la vérification d'un sous-ensemble plus large de fonctions, y compris les fonctions des couches inférieures et les opérations de bas niveau de la TSS et de la librairie tpm2-tss. La spécification et la

vérification de propriétés de sécurité spécifiques constituent un autre axe de travail futur. Le maintien des preuves entre les différentes versions des outils est également un axe de recherche intéressant, notamment en automatisant la génération de scripts de preuve (comme cela a été proposé récemment dans [42]).

Enfin, la combinaison de modules formellement vérifiés avec des modules qui font l'objet d'une vérification partielle (par exemple, limitée à l'absence d'erreurs d'exécution ou à la vérification d'assertion à l'exécution des spécifications attendues sur de grandes séries de tests) peut constituer une autre piste de travail prometteuse pour renforcer la confiance dans la sécurité de la bibliothèque.

Vérification à l'exécution pour la sécurité

Dans les travaux présentés dans le chapitre 5, nous avons seulement vérifié, à l'exécution, deux propriétés de sécurité de haut niveau (l'intégrité et la confidentialité de données sensibles) sur certaines (mais pas toutes) données sensibles, sur un sous-ensemble relativement simplifié de la bibliothèque TPM2-TSS.

À terme, une piste de recherche intéressante consisterait à effectuer un travail de vérification plus complet dans la libraire TPM2-TSS de ce type de propriétés, en réintroduisant les dépendances aux bibliothèques externes et les capacités cryptographiques de la pile logicielle, et en exécutant le code vérifié couplé à un vrai TPM.

Combinaisons d'approches

Les travaux présentés dans la Section 6.1 ouvrent diverses perspectives de recherche. Par exemple, l'automatisation de la technique de vérification proposée pourrait être une perspective intéressante. Cette automatisation pourrait inclure, par exemple, une génération coordonnée, d'un côté de contrôles de la validité de propriétés (dans MemCAD), et de l'autre d'hypothèses (dans Frama-C/Wp). Elle pourrait également inclure une preuve de la solidité de cette génération. Une autre direction intéressante serait la conception d'un mé-

canisme de spécification commun (de plus haut niveau) à Frama-C/Wp et à MemCAD pour les structures de données récursives (telles que les structures de listes chaînées utilisées dans tpm2-tss), avec une traduction automatique vers MemCAD et Frama-C. Ici aussi, la mise en place d'une évaluation plus étendue de notre méthodologie de vérification est un autre point d'amélioration, tout comme le ciblage d'autres études de cas pertinentes.

Le travail accompli et l'approche que nous avons proposée dans la Section 6.2 présentent également plusieurs points d'amélioration et perspectives de recherche. Par exemple, le filtrage des propriétés que nous utilisons actuellement pour répartir l'effort de vérification entre Wp et E-ACSL ne s'applique qu'aux assertions de bas niveau, mais réussir à l'étendre aux contrats de fonction ou même aux propriétés de plus haut niveau serait une nette amélioration. De plus, le filtrage actuel repose sur l'usage d'expressions régulières, mais l'utilisation d'une approche plus rigoureuse telle que le filtrage des propriétés directement au niveau des représentations intermédiaires de Frama-C (telles que les arbres de la syntaxe abstraite) que Frama-C génère lors de l'analyse syntaxique pourrait être une autre perspective intéressante.

Inversement, une autre direction de recherche intéressante pourrait être d'établir une méthodologie qui se concentre sur l'application d'autres analyseurs de la plateforme Frama-C (tels que Wp ou EVA) pour dans un premier temps vérifier ces propriétés de haut niveau, puis dans un second temps laisser ce qui n'a pas pu être vérifié avec succès à la vérification à l'exécution. Cette approche serait idéalement automatique et séquentielle : par exemple, après avoir défini les exigences de haut niveau, MetAcsl serait appelé pour générer les assertions de bas niveau correspondantes; ensuite, l'analyse de valeur du greffon EVA pourrait être automatiquement appliquée pour les vérifier, et transférerait l'effort de vérification des propriétés qui n'auraient pas pu être traitées à Wp; ce greffon ferait à son tour la même chose et transmettrait ce que la vérification déductive ne pourrait pas gérer à la vérification à l'exécution proposée par E-ACSL.

Une autre piste de recherche intéressante serait de chercher à établir une méthodologie de vérification basée sur celle proposée dans le chapitre 5, qui pourrait consister à vérifier les propriétés de haut niveau au moment de

l'exécution, et à introduire progressivement, propriété par propriété, la vérification déductive dans l'effort de vérification.

L'intelligence artificielle au service de la vérification

Les progrès récents de l'intelligence artificielle (IA) et en particulier des grands modèles de langage (LLM) ouvrent la voie à des approches de vérification assistées par l'IA.

De premiers efforts pour évaluer les avantages des techniques assistées par les LLM [14, 64, 72] démontrent leur potentiel et la nécessité d'étudier et de définir une utilisation optimale des approches assistées par l'IA pour la vérification des logiciels. En particulier, les auteurs de [72] ont effectué des travaux préliminaires encourageants vis-à-vis de la génération automatique d'invariants de boucle ACSL en utilisant des LLM.

Cela pourrait également permettre de générer automatiquement des invariants structurels pour les structures de données récursives (telles que ceux que nous définissons et utilisons manuellement pour les listes chaînées dans les chapitres 4 et 6), d'aider à l'automatisation proposée comme perspective du chapitre 6, ou encore de faciliter l'automatisation du traitement des données sensibles grâce au modèle compagnon donné dans le chapitre 5.

Contents

List of Figures

Introduction 1

Contents

1.1 Software Verification

In the past few decades, computer systems, be it personal computers, phones or embedded systems, have become increasingly complex and intertwined with our society. We rely on them constantly in our daily lives, be it in consumer oriented systems (such as personal computers and smartphones), or in industrial systems. From non-critical systems such as mobile banking apps and household appliances, to critical systems like airplane control systems, railroad control systems, and medical devices, most of our activities are now heavily dependent on computer systems, handling more and more critical aspects of our lives.

These systems, due to their crucial nature, often require a high degree of correctness and reliability. In spite of that, we are all too much used to encountering, reporting and finding ways to work around software bugs, as humans designing such systems are bound to make mistakes. That is why software verification is so important and can prove to be absolutely critical.

In less critical applications, the standard approach to software verification is testing. When done correctly, it is usually an appropriate solution, sufficient

to detect not all, but most bugs present in software systems and ensure an adequate level of operation.

Nevertheless, such an approach is not always satisfactory. In industrial systems and other critical applications that can be responsible for the security of personal or sensitive information, or for the safety of human lives, a much higher level of confidence and degree of correctness are required.

1.2 Formal Verification

Deductive Verification

Deductive verification [69] is a powerful modular static verification technique. The main principle of this approach is to express the correctness of a program as a mathematical logical statement, and then to construct a proof of its validity. Such statements are called specifications, and are used to generate proof obligations, or verification conditions, the truth of which imply the conformance of the program to the specification. The verification conditions can be proved using either automatic provers (such as SMT solvers like *Alt-Ergo* [41] or *CVC4* [9]) or interactive theorem provers (like Coq/Rocq [108] or *Isabelle/HOL* [16]).

Deductive verification of a program is modular: each function of the program is verified independently in isolation from the other ones. For each function, we use its specification to generate proof obligations that are strong enough to guarantee that the function conforms to its specification. In case the function contains calls to other functions, we use the specifications of the callees as hypothesis for the proofs of the specification of the caller: in large programs, it is thus preferable to first verify lower level callees before proving the specifications of higher level callers.

This technique is complete and sound, but is usually not automatable and requires manual writing of various suitable proof-guiding annotations to conduct the proof of some verification conditions.

Runtime Assertion Checking[1]

Runtime verification [73, 57], or runtime assertion checking (RAC), is a dynamic verification technique aimed at addressing problems at the boundary between formal verification and testing. It is a lightweight formal verification technique, whose main principle is to verify that some formal properties about a program hold at runtime, that is to say during the execution of the target program. It aims at detecting and reporting errors as soon as they happen during execution.

Runtime verification focuses on individual complete program executions, and can serve as a more efficient but less effective alternative for situations that can prove to be too complex for static verification methods. Indeed, it offers weaker correctness guarantees than more rigorous methods, as RAC *checks individual* program executions, whereas methods such as deductive verification *prove* the correctness of properties for *all* program executions.

Abstract Interpretation

Abstract interpretation [43] is a static analysis technique. Its main principle is to map program variables to abstract domains that over-approximate the set of concrete values they can take during any possible execution of the program. For instance, integer-like variables can be abstracted with an interval. This method is sound and suitable for automated proof, but is not complete, as over-approximations can cause false alarms and the detection of problems in otherwise correct programs.

In that respect and because it relies on over-approximation of program variables, abstract interpretation is more automatable and requires less intermediary annotations than deductive verification, but is less precise in its results. At the same time, it provides a higher level of confidence in verification results than runtime assertion checking, but often requires a bigger specification effort.

[1] The historical term "runtime assertion checking" could be generalized today as "runtime annotation checking", since any type of annotation can be checked at runtime.

Model Checking

While we do not use this technique in this thesis, we shall provide a rough description of it. Software model checking [36] is a software verification technique, aimed at verifying a program (or an abstract model of it) by completely exploring the set of possible states the program can reach during its execution.

The possible specifications for properties of the system are potentially not limited to a set of incorrect states to exclude, and they are generally expressed in temporal logic. Since the set of states can be very large, exploring all possible states can be impossible; in which case, abstraction methods are used to construct an approximate model of the verified system using only a finite number of states.

1.3 Contributions and Document Structure

The aim of this thesis is to evaluate to what extent it is currently possible to formally (or semi-formally) verify safety properties, functional properties, and security properties, on real-life critical code not designed with verification in mind.

We notably focus here on the Trusted Platform Module (TPM), a crypto-processor designed to protect integrity and security of modern computers. We focus more particularly on the TPM Software Stack (TSS), which assumes the role of a trusted layer for the TPM, through which communications with the TPM pass.

As a popular C implementation of the TPM Software Stack, and making the bridge between the TPM and applications, the tpm2-tss library is highly critical, and the library code is very complex and challenging for verification tools.

Such tools include the Frama-C [74] platform, which provides a framework for the analysis and verification of ANSI/ISO C programs. The platform is built around a kernel providing basic services for source-code analysis, allowing users to perform different kinds of analysis and verification of C programs through various plugins. In this thesis, we rely mostly on the Wp [13] plugin

for deductive verification, the E-ACSL [73] plugin for runtime verification, and the MetAcsl [99] plugin for the definition of high-level requirements (which are especially useful to express security properties such as integrity and confidentiality). We also rely on the MemCAD [104] abstract interpretation tool.

Contributions

The main contributions of this thesis are:

- The specification and the proof, in Frama-C/Wp, of low-level safety and functional properties on a simple illustrative example and on a representative subset of functions of tpm2-tss, a real-life critical C library not designed with formal verification in mind. We present the main issues we faced during the verification with an illustrative example, and we describe the solutions and workarounds we found. We provide the Coq proofs for some lemmas necessary to handle linked lists. These contributions are presented in Chapter 4.

  These contributions are associated with a companion artifact [117] illustrating all issues we encountered and solutions we proposed on the real-life functions, annotated in ACSL and fully proved in Frama-C/Wp (as well as all necessary lemmas and their proofs). This artifact received both EAPLS Availability and Reusability badges at the *18th International Conference on integrated Formal Methods* (iFM 2023).

- A modeling approach relying on specific data structures, usable for the specification and verification of high-level security properties. We specify and verify, at runtime, in Frama-C/E-ACSL, high-level security properties on a representative simplified subset of functions of tpm2-tss. We present the main issues we faced and evaluate verification results including the detection of injected errors. These contributions are presented in Chapter 5.

  These contributions are associated with a companion artifact [116] containing the code, tools, definitions of high-level requirements to be verified, and instructions required to replicate our verification results. This

artifact received both EAPLS Availability and Functionality badges at the *18th International Conference on Tests and Proofs* (TAP 2024).

- Preliminary work on combining deductive verification with other techniques as an alternative to the temporary workarounds we rely on in other contributions. We propose the combination of Frama-C/Wp with the shape analysis tool MemCAD (relying on abstract interpretation) on functions manipulating linked lists, as well as a successful case study on a set of functions of the tpm2-tss library. Additionally, we explore the possible combination of deductive verification and runtime assertion checking for the verification of high-level properties. These contributions are presented in Chapter 6.

  The contributions related to the combination of Frama-C/Wp with MemCAD are associated with a companion artifact [18], which includes a Virtual Machine containing the installed tools and code examples used, and can be used to reproduce the results presented in Section 6.1. This artifact received both EAPLS Availability and Functionality badges at the *27th International Conference on Fundamental Approaches to Software Engineering* (FASE 2024).

Publications and other realizations

The preparation of this thesis included the submission and publication of several articles in peer-reviewed conferences and journals.

- "Towards Formal Verification of a TPM Software Stack" [118], appeared at the *18th International Conference on integrated Formal Methods* (iFM 2023). This publication presents our work on the deductive verification of low-level safety and functional properties presented in Chapter 4 of this thesis, and was nominated as a best paper award candidate for this conference.

- "Runtime Verification for High-Level Security Properties: Case Study on the TPM Software Stack" [119], appeared at the *18th International Conference on Tests and Proofs* (TAP 2024). This publication presents our work

on the runtime verification of high-level security properties presented in Chapter 5.

- "Combining Deductive Verification with Shape Analysis" [17], appeared at the *27th International Conference on Fundamental Approaches to Software Engineering* (FASE 2024). This publication presents our work in Section 6.1 on the combination of verification techniques to alleviate some of the limitations of deductive verification identified in Chapter 4.
- "Towards Formal Verification of a TPM Software Stack: Achievements and Opportunities" [120], accepted in the *Formal Aspects of Computing* (FAC) journal. This publication is an extended version of the iFM 2023 paper.
- Three companion artifacts (described above in the Contributions paragraph) that contain all files of the realized case studies and allow the readers to reproduce our verification results.

Outline

The rest of this thesis is structured as follows:

- Chapter 2 introduces in more detail the TPM, its Software Stack and the TPM2-TSS library, FRAMA-C, ACSL, WP, E-ACSL and METACSL. We provide examples detailing the usage of various ACSL clauses, as well as a brief presentation of each plugin's functionalities.
- Chapter 3 presents the state of the art regarding the reliability and the security of the TPM and similar secure components, as well as the current state of the art on formal verification of real-life code.
- Chapter 4 presents our work on the proof of safety and functional properties in real-life security-critical code using the WP plugin for deductive verification, focusing on a subset of functions of the TPM2-TSS library in charge of important internal operations.

- The purpose of Chapter 5 is to propose an alternative validation approach to deductive verification (with Wp), as well as a preliminary verification methodology for high-level security requirements in real-life critical code like tpm2-tss, using E-ACSL for the verification, and MetAcsl for the definition of such requirements.
- In Chapter 6 we present preliminary work on combining validation techniques to help alleviate some of the limitations we identified with the deductive verification of Wp in Chapter 4 and the runtime verification of E-ACSL for high-level requirements in Chapter 5. We focus here on two combinations: Frama-C/Wp and MemCAD [104], a shape analysis tool, for the verification of low-level safety and functional properties, and Wp with E-ACSL, for the verification of high-level security properties.
- Chapter 7 concludes and presents some perspective.

Preliminaries 2

Contents

2.1 The Trusted Platform Module and its Software Stack

This section briefly presents the Trusted Platform Module (TPM), its software stack and the implementation we chose to study: the tpm2-tss library.

For more details, readers can refer to the official TPM [109] and TSS specification [110] from the Trusted Computing Group (TCG) and reference books such as [5].

The Trusted Platform Module The TPM is a standard conceived by the TCG[1] for a secure cryptoprocessor designed to protect secure hardware from software-based threats. The TPM is passive, in the sense that it "simply" receives commands and returns responses, but it does not possess the ability to run code or operations by itself. At its base, a TPM is implemented as a discrete cryptoprocessor chip, attached to the main processor chip and designed

[1] https://trustedcomputinggroup.org/

to perform cryptographic operations. However, it can also be implemented as part of the firmware of a regular processor or as a software component.

Nowadays, the TPM is well known for its usage in regular consumer personal computers to ensure properties such as the integrity of the used Operating System (OS) during boot, and to provide a secure storage for the keys used to encrypt the disk with software such as *Bitlocker* and *dm-crypt*.

It can also be (and is actually) used to provide other cryptographic services to the OS or applications. For that purpose, the TCG defines the TPM Software Stack (TSS), a set of software specifications designed to provide standard APIs to access the functionalities and commands of the TPM, regardless of the hardware, OS, or environment used.

We shall make the distinction here between *users* (such as, any OS, application, application developer, or human user utilizing the TPM) and *developers* (such as, coders and engineers in charge of developing TSS implementations.)

TPM Software Stack. The TSS APIs provide different levels of complexity, from higher level layers such as the *Feature API* (FAPI) for simple services and better user accessibility, to the *System API* (SAPI) providing greater flexibility but with a more complex usage. In between lies the *Enhanced System* API (ESAPI) as some sort of middle ground between FAPI and SAPI. Other TSS APIs complete the previous ones for common operations like data formatting and connection with the TPM.

There are, to this date, several TPM libraries that use their own richer API, but very few that implement the TCG specification of the TSS. The TPM2-TSS open-source library is one of them, and as it is the most used by Linux applications and distributions, we will be focusing on it in this thesis.

The TSS APIs, as any software component or the TPM itself, can have vulnerabilities. Among potential vulnerabilities, we can mention, in particular, the following:

- **CVE-2020-24455** in older versions of the TPM2-TSS library, where a missing initialization could potentially enable an escalation of privilege via local access.

- **CVE-2023-22745**, where a buffer overrun in some versions of the TPM2-TSS library could result in arbitrary code execution.
- **CVE-2024-29040**, where the TSS could potentially not detect a malicious device impersonating a regular TPM.

These vulnerabilities (and other potential ones) could be exploited by attackers to recover sensitive data communicated with the TPM or take control of the system.

The next paragraphs will briefly present the structure of the TSS, and the corresponding elements in the TPM2-TSS library.

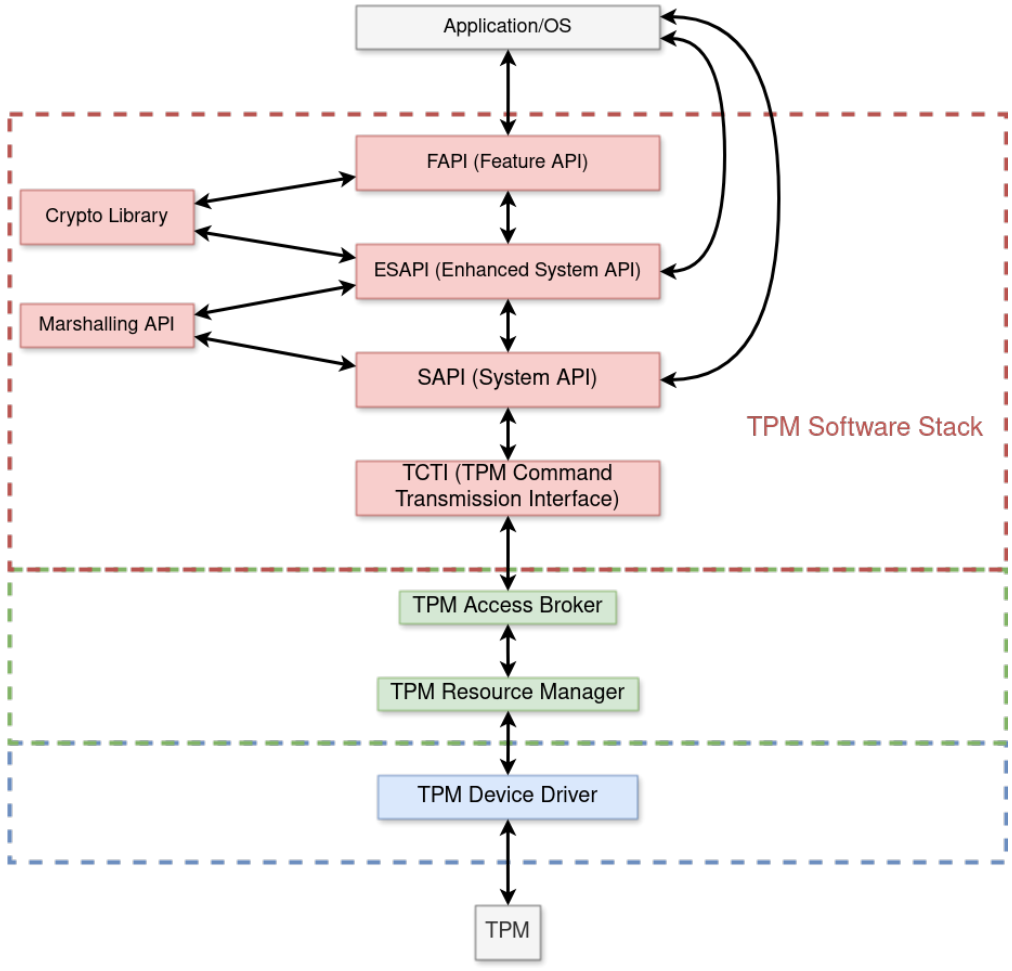


Figure 2.1 – *Overview of the TPM Software Stack*

The TSS is split into several layers, as illustrated in Figure 2.1:

- The Feature API (FAPI), a high-level API, designed to make programming with the TPM as simple as possible. This layer is implemented as the `tss2-fapi` module in the library. It is designed to capture most common use cases for user applications. Writing code in this layer is akin to writing in object languages such as C# and Java.
- The Enhanced System API (ESAPI), which provides a one-to-one mapping to TPM commands, as well as support for cryptographic and session management capabilities (including but not limited to, tracking object meta-data and computing encryption values). This layer is implemented as the `tss2-esys` module in the library. Due to the nature of its functionalities, this layer may also be used to develop more sensitive applications. Writing code in this layer is equivalent to writing in C++.
- The System API (SAPI), which also provides a one-to-one mapping to TPM commands and access to all the functionalities of the TPM. This layer is implemented as the `tss2-sys` module in the library (but writing applications there requires a high expertise regarding the TPM). It contains asynchronous variants of each command (default is synchronous).
- The TPM Command Transmission Interface (TCTI), a standard interface to transmit TPM commands and receive response buffers. This layer is implemented as the `tss2-tcti` module in the library. The TPM2-TSS library provides several implementations of the TCTI, each corresponding to a type of usable TPM (be it for instance a dedicated chip, a firmware TPM, or an emulated or simulated one).

The TSS also provides a marshaling API (the `tss2-mu` module), shared between the ESAPI and SAPI layers, used to transform all TPM2 types that map directly to fixed size primitive types in the C, so that they can be sent to lower layers and the TPM.

Feature API The FAPI layer (`tss2-fapi` module in TPM2-TSS) is designed with the hope that most of the applications that would eventually use the

TPM could be written using the FAPI layer, without having to resort to any of the lower-level layers. It provides APIs designed to isolate the user from the messiness of the TPM 2.0 specification. It is very large with over 55,000 lines of C.

More specifically, it was designed not only to render the most used capabilities of the TPM more easily available to programmers, but also to minimize the number of calls programmers have to use, and the number of parameters they have to define. In that sense, it essentially works as a very high-level API to the TPM. As such, the layer uses profile files as default presets of settings, encryption algorithms, key sizes, signing schemes, and other relevant parameters so that the developer does not need to select them manually when creating and using keys.

As other layers of the software stack, it relies on a notion of context (using the `FAPI_CONTEXT` type) containing all internal state information of FAPI, in an effort to allow several independent contexts to operate in parallel within the same process environment. The context structure is defined as an opaque structure in the official TCG Specifications.

Enhanced System API The ESAPI layer (`tss2-esys` module in TPM2-TSS) provides functions for decryption and encryption, managing session data and policies, thus playing an essential role in the TSS. It is almost as large as the FAPI layer (with over 50,000 lines of C) and is mainly split into two parts: the API part containing functions in a one-to-one correspondence with TPM commands (for instance, the `Esys_Create` function of the TSS will correspond to — and call — the `TPM2_Create` command of the TPM), and the back-end containing the core of that layer's functionalities. Each API function will call several functions of the back-end to carry out various operations on command parameters, before invoking the lower layers and finally the TPM.

The ESAPI layer relies on a notion of context (`ESYS_CONTEXT`) containing all data the layer needs to store between calls, so it does not need to maintain a global state. The memory for the `ESYS_CONTEXT` is allocated by the ESAPI. Similarly to that of FAPI, it is defined for external applications as an opaque structure. The context includes, according to the documentation, data needed

to communicate to the TPM, meta-data for each TPM resource, and state information. The specification, however, does not impose any precise data structure: it is up to the developers to provide a suitable definition.

System API The SAPI layer (`tss2-sys` module in TPM2-TSS) is the lowest layer that contains software functions to perform calls to all TPM commands. In that way, the layer provides access to all the functionality of the TPM, but requires a high level of expertise to directly use and to write applications there. It is much smaller in size than the higher layers with approximately 15,000 lines of C.

In the same way as for the ESAPI, the SAPI layer relies on a notion of context (`SYS_CONTEXT`) containing the data the layer needs to store between calls. It is defined as an opaque structure as well, containing the state of communications and all data the layer needs to store between calls. All SAPI data structures, however, shall be allocated by the caller and not by the layer, as opposed to ESAPI.

There are four types of SAPI "commands", corresponding to four types of functions in the `tss2-sys` module: context allocation, command preparation, command execution and command completion functions. Context allocation functions are common to all calls to TPM commands. The other three groups correspond to different stages of a command call: for instance, the call to the `TPM2_Create` TPM command and the handling of its response are split between the SAPI functions `Tss2_Sys_Create_Prepare`, `Tss2_Sys_Create_Execute` and `Tss2_Sys_Create_Complete`, with the preparation function in charge of preparing the command buffer to be sent to the TPM, the execution function to send it, and the completion function to receive and parse the TPM response buffer.

TPM Command Transmission Interface TCTI is the abstraction layer used to transmit TPM command byte streams to the TPM, and receive response byte streams from the TPM. It is the smallest layer of the stack, with "only" about 5,000 lines of code.

The TCTI relies on a context structure to tell the SAPI functions how to

communicate with the TPM. It contains function pointers for the two main TCTI operations, transmit and receive, as well as less frequently used functions (such as cancel). Just like the ESAPI and SAPI contexts, it is not maintained as a global state variable, and shall either set up at compile time, or dynamically when the OS boots.

The TCTI does not have the means, however, to automatically detect a TPM to communicate with, and to initialize a context depending on the type of TPM. In general, some operation prior to calling the SAPI or TCTI layers is in charge of initializing a TCTI context structure with the proper function pointers for communication.

Marshaling API While not being, strictly speaking, a layer of the TSS, the Marshaling/Serialization API (`tss2-mu` module in tpm2-tss) completes the previous one by providing an API to build TPM command byte streams (marshalling) and to decompose TPM response byte streams (unmarshalling). It is notably useful to transfer data back and forth from ESAPI to SAPI, and from higher-level representation to byte streams. Marshaling and unmarshaling operations, as they are used by several layers of the TSS, are therefore kept in their own API. More specifically, the TSS defines in this API a standard set of functions for marshaling and unmarshaling the C representation of each TPM structure defined in the TPM2.0 Structures Specification[2].

2.2 Frama-C Verification Platform

This section presents the Frama-C platform, which provides a framework for the analysis and verification of ANSI/ISO C programs. The platform is built around a kernel providing basic services for source-code analysis, allowing users to perform various kinds of analysis and verification of C programs.

[2] `https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-1.83-Part-2-Structures.pdf`

A Plugin-based Framework for C Program Analysis

Currently, the C programming language is still widely used in the industry, mainly as a system programming language or for embedded systems development. Indeed, thanks to its low abstraction level, the C language provides the ability to interact closely with hardware, is very efficient in terms of resource usages, and allows for low-level memory manipulation. Such characteristics are crucial also for devices with limited processing power and memory.

However, such versatility comes at a cost, as the C language is notoriously known for being difficult to use to write programs without bugs, due to an absence of memory safeguards and many unsafe constructs. Thus, there have been efforts over the years to develop tools dedicated to the analysis of critical C code in order to reduce the odds of bugs, including Frama-C.

Frama-C consists of a kernel that parses C source files and generates an intermediate internal representation from them. That representation can then be passed to plugins which can inspect it, and perform different kinds of analyses and verifications. In particular, Frama-C can perform deductive verification of specified program properties using the Wp plugin (Section 2.2.2), or verify them at runtime using E-ACSL (Section 2.2.2), and the MetAcsl plugin can be used to express high-level properties over whole programs.

Frama-C relies on the ANSI/ISO C Specification Language (ACSL) for the specification and formalization of properties, which may be used as a common specification mechanism for different analyses and different plugins. Section 2.2.1 presents all the concepts of ACSL needed in the thesis.

2.2.1 The ACSL Specification Language

The ANSI/ISO C Specification Language, abbreviated as ACSL, is a formal specification language for C. It is the formal language supported by Frama-C for writing specifications in the form of code annotations such as assertions or function contracts. It allows the specification of behavioral properties of C source code. Effectively, it is a *Behavioral Interface Specification Language* (BISL) [66] for the C language. It relies on the concept of *design-by-contract* introduced by Meyer in [86] on which the Eiffel [85] language heavily relies, and

its syntax is akin to that of Java Modeling Language [28, 77] (JML), a similar behavioral specification language for Java programs.

In the rest of this section, we introduce the concepts of ACSL needed for this manuscript. We use the `search` function of Figures 2.2-2.7 to illustrate these concepts. The function takes as arguments an array `T` of **`int`**, its size `size`, `x` the **`int`** value to be searched in the array, and a pointer `pos`. If `x` is in `T`, then `search` will return 1 or 0 depending on whether or not `x` was found in the array, and write its index in `*pos` if it is found. For an exhaustive presentation of the language, the reader is invited to consult the ACSL specification document [11] and the *Guide to Software Verification with Frama-C* [74].

```
1  int search(int* T, unsigned size, int x, unsigned *pos) {
2    int res = 0;
3    for(unsigned i = 0 ; i < size ; ++i){
4      if(T[i] == x){
5        res = 1;
6        *pos = i;
7      }
8    }
9    return res;
10 }
```

Figure 2.2 – *Example search function*

Function contract. As any BISL, the language is based on a notion of function contract. A function contract in ACSL is an ACSL specification preceding the definition or the declaration of a function f in the source code (e.g. lines 1-16 of Figure 2.5 for the `search` function). It usually consists of three main parts:

- **The preconditions** are supposed to be true before a call to f. They express properties that must hold in the state of the program whenever that function is called. The preconditions are specified in **`requires`** clauses.

  For instance, line 2 of Figure 2.5 requires that variable `x` should be between 0 and 1000 whenever `search` is called.

- **The postconditions** should be satisfied after the call to *f*. They are properties that must hold in the state of the program when that function returns. The postconditions are specified in **`ensures`** clauses.

  For instance, line 11 of Figure 2.5 ensures that if the return value of the function is `1`, then the value pointed by `pos` is the index of an occurrence of `x` in array `T`.

- **The frame condition** states an exhaustive list of (non-local) memory locations of global memory state that can be modified by function *f*, and may therefore have a different value before and after the call to *f*. A frame condition is expressed using the **`assigns`** clause.
  For instance, line 7 of Figure 2.5 indicates that only the result returned by `search` and the value stored at address `pos` (passed as argument of the function) may be modified.
  As is the case here, the frame clause is optionally followed by a **`\from`** clause, indicating the modified values can only depend upon the mentioned locations, in this case, the content of the `T` array, its `size`, and the searched value `x`.

  Frame conditions are in general an over-approximation of the set of side effects. Fundamentally, the main point of an **`assigns`** is to indicate that any non-local variable that is not mentioned in the clause is unchanged by the function, rather than to provide a list of potentially modified variables.

C-specific constructs. ACSL is based on mathematical first-order logic through the use of logic expressions. They correspond to pure C expressions, with additional constructs that allow the expression of useful properties about C values and expressions.

Figure 2.3 describes some common ACSL operators useful for this thesis. As it is often done, some ACSL notations (e.g. `\forall`, `integer`) will be pretty-printed when used (resp., as $\forall$, $\mathbb{Z}$) as presented in Figure 2.3.

As ACSL is based on classical first-order logic there is a distinction to be made between ACSL predicates (which correspond to propositions in first-

Syntax	Pretty-printed as	Semantics
`&&, ||, ==>, <==>, !`	$\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg$	Connectives
`>, <, >=, <=` `==, !=`	$>, <, \geq, \leq$ $==, \neq$	Comparison operators
`\in`	$\in$	Element of operator
`\forall type var; P` `\exists type var; P`	$\forall$ `type var; P` $\exists$ `type var; P`	Universal quantification Existential quantification
`integer, real, boolean`	$\mathbb{Z}, \mathbb{R}, \mathbb{B}$	Built-in logic types
`\at(expr, label)` `\old`		Value of location at label
`\valid(p)` `\valid_read(p)`		Pointer validity Pointer read validity
`\separated(p1, p2)`		Memory separation

Figure 2.3 – *Main* ACSL *terms and predicates*

order logic, evaluating to true or false) and terms (which evaluate to values). Most of the operators are self-descriptive. However, the last three lines of the table contain constructs specific to C and its handling of the global memory of a program:

- The `\at(expr, label)` construct allows one to refer to the value of the expression `expr` at a specific program point identified by the label `label`. By default, writing just an expression `expr` is equivalent to writing `\at(expr, Here)`, where `Here` is the built-in logic label referring to the point where the property is defined.

  There exist several built-in labels in ACSL, such as `Pre` (resp. `Post`) which is only usable in function contracts and refers to the state before (resp. after) the current function. More details on these labels are described in Figure 2.4. Additionally, any previously defined C label can be used in the same way.

- ACSL provides the built-in `\valid` and `\valid_read` functions to deal with pointer validity[3]: `\valid(p)` (resp. `\valid(s)`) holds *if and only*

[3]In C, some operations involving pointers may lead to runtime errors, which in some cases may be avoided by ensuring pointer validity. A pointer `p` is said to be valid if `*p` is guaranteed to produce a definite value according to the C standard [60].

Label	Permitted locations	Meaning
`Here`	statement annotations	state where the annotation is
	function contracts	pre-state of the function
`Old`	function contracts	pre-state of the function
`Pre`	statement annotations	pre-state of the function
`Post`	assigns and ensures clauses	post-state of contract
`LoopEntry`	loop annotations and loop statements	state prior to the first loop entry
`LoopCurrent`	loop annotations and loop statements	state at the beginning of current loop iteration
`Init`	all annotations	state before call to main

Figure 2.4 – *Built-in* ACSL *labels*

if dereferencing the pointer `p` (resp. any pointer `p ∈ s` where `s` is a set of pointers) is safe, both for reading from `*p` and writing to it (the `\valid_read` variant is similar but only specifies that the pointer is safe for dereferencing for reading).
Note that the validity of a pointer is related to its type, since the type of `p` changes the size of the pointed region: `\valid((int*)p)` and `\valid((char*)p)` are not equivalent. The type is also used when specifying a range of valid locations.
For instance, line 3 of Figure 2.5 requires that the range of `size` elements of type `int` at address `T` is safe for deferencing for reading (that is to say, that the first `size` elements of the `T` array of type `int` are safe to read).

- The predicate `\separated` applies to two sets of pointers of some type `s1` and `s2`, and holds if and only if there are no overlaps between the memory locations pointed by `s1` and those pointed by `s2`, i.e.
  `∀ p1 ∈ s1` and `∀ p2 ∈ s2`,
  `∀ ℤ i, j; 0 ≤ i < sizeof(*p1), 0 ≤ j < sizeof(*p2) ⇒`
  `(char*)p1 + i ≠ (char*)p2 + j`
  For instance, line 5 of Figure 2.5 requires that the range of `size` elements of type `int` at address `T` (that is to say, the first `size` elements of the `T` array of type `int`) is separated in memory from the `sizeof(*pos)` bytes at address `pos`.

```
1  /*@
2    requires 0 ≤ x ≤ 1000;
3    requires \valid_read(T + (0 .. (size - 1)));
4    requires \valid(pos);
5    requires \separated(T + (0 .. (size - 1)), pos);
6
7    assigns \result, *pos \from *(T + (0 .. size - 1)), size, x;
8
9    ensures \result == 0⟺
10     ∀ unsigned j; 0 ≤ j < size ⇒ T[j] ≠ x;
11   ensures \result == 1 ⇒ T[*pos] == x;
12   ensures \separated(T + (0 .. (size - 1)), pos);
13   ensures \result == 0 ∨ \result == 1;
14
15   ensures \result == 0 ⇒ *pos == \old(*pos);
16 */
17 int search(int* T, unsigned size, int x, unsigned *pos)
```

Figure 2.5 – *User-defined* ACSL *function contract for the search function*

Behaviors. Named behaviors can be used to make function contracts more structured: postconditions associated to a named behavior must only be ensured if the associated assumption is verified at the beginning of the function. They are declared using the **behavior** keyword in function contracts.

Behaviors are composed of two main clauses, **assumes** and **ensures** (as well as **assigns** — whose support by Frama-C plugins is partial — and **requires** — by extension of the function contract). The **assumes** clause sets the condition for which the behavior must be ensured, and the **ensures** allows for the definition of postconditions specific to the corresponding behavior.

For example, Figure 2.6 additionally provides two behaviors to the function contract of the search function : `x_in_T` defined in lines 16-20, and `x_not_in_T` defined in lines 22-26. The named behavior `x_in_T` means that if the **assumes** clause line 16 holds (that is, if **int** `x` can be found in array `T`) when `search` is called, then the postconditions lines 19-20 must also hold (that is to say, `x` is stored in `T` — at least once — at the index given by the new value pointed by `pos`, and the returned value is `1`).

While it is not required by default, it is also possible to specify that the behaviors must be complete (resp. disjoint) using the **complete** keyword like in line 27 (resp. the **disjoint** keyword line 28). More specifically, the **complete behaviors** clause means that the set of behaviors is complete, i.e. that
$R \Rightarrow (A_1 \vee A_2 \vee ... \vee A_n)$ holds (where R is the conjunction of all **requires** clauses of the function, and A_k is the conjunction of all **assumes** clauses of the k-th behavior). The **disjoint behaviors** clause means that the behaviors are disjoint, i.e. that for all distinct i and j, $R \Rightarrow \neg(A_i \wedge A_j)$ holds. In the example of Figure 2.6, both properties (line 27-28) hold: the two behaviors are complete and disjoint.

```
1  /*@
   ...
14   ensures \separated(T + (0 .. (size - 1)), pos);
15
16   behavior x_in_T:
17     assumes ¬(∀ unsigned j; 0 ≤ j < size ⇒ T[j] ≠  x);
18
19     ensures T[*pos] == x;
20     ensures \result == 1;
21
22   behavior x_not_in_T:
23     assumes ∀ unsigned j; 0 ≤ j < size ⇒ T[j] ≠  x;
24
25     ensures *pos == \old(*pos);
26     ensures \result == 0;
27   complete behaviors;
28   disjoint behaviors;
29 */
30 int search(int* T, unsigned size, int x, unsigned *pos)
```

Figure 2.6 – *User-defined behaviors in the function contract for the search function*

Assertions. Statement annotations allow writing annotations directly on a statement. The **assert** `P` clause is an example of a statement annotation that ensures that a condition `P` holds at a given program point. In proofs, this

statement can thus be used as hypothesis for the subsequent annotations (e.g. assertions, postconditions, loop annotations) in the code. See the assertion lines 23-24 of Figure 2.7 for example.

Additionally the `check P` clause behaves similarly, except that it only checks whether condition P holds at a given program point, but does not assume it for later proof annotations. Thus, the "assertion" cannot be used as hypothesis for further proofs. The `admit P` variant clause, for its part, serves to use condition P as hypothesis for future proofs, without verifying whether or not it holds.

In that sense, `assert P` is essentially equivalent to writing a `check P` annotation followed by an `admit P` annotation.

Loop annotations. Loop annotations are written before a loop and are divided into three clauses:

- loop invariants[4] in ACSL have the following semantic: the `loop invariant I` clause means that
  - `I` must hold in the state before the loop, and
  - *if* `I` holds at the start of a loop iteration, *then* `I` must also hold at the end of this iteration.

  For instance, line 6 of Figure 2.7 states that `unsigned i` used to iterate over the loop must always be between `0` and `size` before and after each iteration of the loop. Lines 8-9 state that `*pos` must have the same value at the start of each iteration of the loop (as stated by `LoopCurrent`) as it did at the start of the loop (as stated by `LoopEntry`).
- loop variants[4] in ACSL have the following semantic: the `loop variant V` (where V is of type `integer`) clause means that:
  - at the beginning of any iteration, the value of `V` must be ≥ 0, and

[4] loop invariants in ACSL have the same semantic as in Hoare logic.

```
1  int search(int* T, unsigned size, int x, unsigned *pos) {
2    int res = 0;
3
4    /*@
5      loop invariant 0 ≤ i ≤ size;
6      loop invariant res == 0⟺
7        (∀ unsigned j; 0 ≤ j < i ⇒ T[j] ≠ x);
8      loop invariant res == 0 ⇒
9        \at(*pos, LoopCurrent) == \at(*pos, LoopEntry);
10     loop invariant res == 1 ⇒ T[*pos] == x;
11     loop invariant res ≠ 0 ⇒ res == 1;
12
13     loop assigns i, *pos, res;
14
15     loop variant size - i;
16 */
17   for(unsigned i = 0 ; i < size ; ++i){
18     if(T[i] == x){
19       res = 1;
20       *pos = i;
21     }
22   }
23   /*@ assert ∀ unsigned i; 0 ≤ i < size ⇒
24                T[i] == \at(T[i], Pre);*/
25   return res;
26 }
```

Figure 2.7 – *Inline loop annotations for search*

  - at the end of any complete iteration of the loop (excluding interrupted iterations by statements such as **break** or **return** clauses), the value of `V` must be strictly smaller than at the start of the iteration.

  For instance, line 15 of Figure 2.7 provides `size - i` as a variant.

- Frame conditions for loops are written using the **loop assigns** clause, whose semantic is similar to that of the regular **assigns** clause for function contracts: **loop assigns** is used to state an exhaustive list of mem-

ory locations (non-local to the loop body) of the memory state of the function that can be modified by the loop, and therefore may have a different value before and after every iteration of the loop.

For instance, line 13 of Figure 2.7 states that only `i`, `*pos` and `res` may be modified by the loop.

```
1  //@ ghost int q;
2
3  /*@
4    requires a ≥ 0 ∧ 0 < b;
5    ensures a == q * b + \result;
6    ensures 0 ≤ \result < b;
7  */
8  int div_remainder(const int a, const int b) {
9    //@ ghost q = 0;
10   int r = a;
11   /*@
12     loop invariant a == b*q+r ∧ r≥0;
13     loop assigns q,r;
14     loop variant r;
15    */
16   while (b ≤ r) {
17       //@ ghost q = q + 1;
18       r -= b;
19   }
20   return r;
21 }
```

Figure 2.8 – *Function definition and* ACSL *contract to compute the remainder of an Euclidean division*

We will use the Euclidean division function of Figure 2.8 to present ghost code. The `div_remainder` function defined in lines 8-21 serves to compute and return the remainder `r` of the Euclidean division of `int a` by `int b`.

Ghost code. Ghost variables and statements are like C variables and C statements, but they are declared for specification purposes only and cannot

be used by the original code. They are introduced by the `ghost` keyword at the beginning of the annotation. Ghost statements can read the content of ghost and non-ghost variables, but may only modify ghost variables (more generally, ghost code cannot modify a non-ghost C variable or structure field). They cannot modify the control flow of the original program either. In particular, no ghost `return` can appear in the body of a non-ghost function. Similarly, ghost `goto`, `break`, and `continue` cannot jump outside of ghost code.

In Figure 2.8, we use ghost code in line 1 to simply declare a variable `int q` we shall use to compute the quotient of the Euclidean division done in function `div_remainder`. We set it to `0` in line 9, and increment it by `1` in line 17 in every iteration of the loop. We also use the loop invariant line 12 to ensure that in every iteration of the loop, we always have `r >= 0`, and `a` equal to `b*q+r`.

2.2.2 A Plugin-based Framework

While the Frama-C kernel does not perform any analysis by itself, it provides a framework for parsing C programs annotated in ACSL, which thus works as a common specification mechanism between plugins. Some plugins can and may be combined in order to perform a wide variety of analyses and transformations.

We focus here on the Wp, RTE, E-ACSL and MetAcsl plugins used in this thesis.

The Wp Plugin for Deductive Verification. Frama-C offers Wp [13], a plugin for deductive verification.

Given a C program annotated in ACSL, the Weakest Precondition [12] (Wp) plugin generates corresponding proof obligations (also called verification conditions) encoding the semantics of ACSL annotations and C code. These obligations can then be proved either directly by Wp or fed to the Why3 [26, 59] verification platform. Why3 can then dispatch them to SMT solvers (such as Alt-Ergo [41], CVC4 [9] or Z3 [48]), or to an interactive proof assistant like Coq [19], where the user interactively performs the proof. The verification method here is modular: the contract and annotations of each function are

verified independently from the other functions, using only the contracts of the callees.

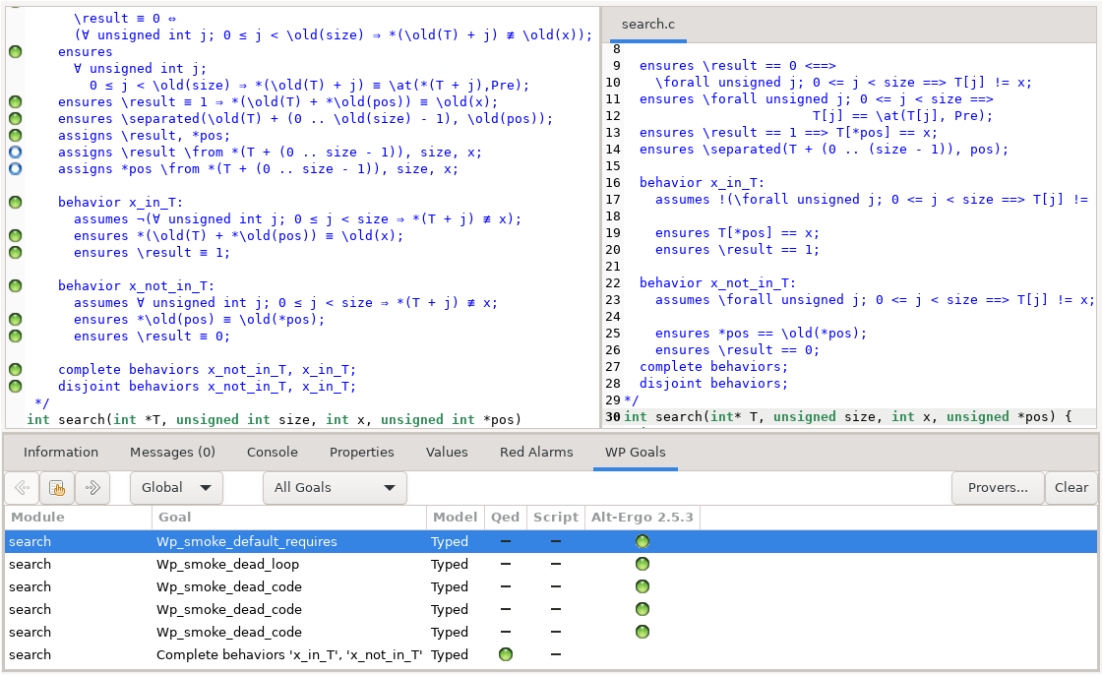


Figure 2.9 – *Graphical User Interface for Frama-C*

A preferred way to check verification results is through the GUI, part of it is shown in Figure 2.9. The right part of the interface displays the original source code, while the left panel show the code pre-processed and parsed for Wp. ACSL annotations are displayed in blue text, with their verification status given by colored icons on the left part of the panel. In our case, the green icons show that all properties defined in the contract of our `search` function have been successfully proved valid by Wp, and the empty icon that no proof was attempted for the **`\from`** clause (which is currently only partially supported). A green and orange icon would indicate a valid property under certain conditions or dependencies, and an orange icon that the corresponding property has not been proved.

The RTE Plugin. The RTE [68] plugin is dedicated, for each expression potentially leading to an undefined behavior (e.g. invalid pointer dereference,

arithmetic overflow), to generating an ACSL assertion with a condition that prevents the corresponding issue. If proved, these assertions ensure that there are no runtime errors in the analyzed program. RTE is usually used as a pre-processor for the Wp or E-ACSL plugins, which can be used to verify that the generated assertions are valid (hence, that no runtime errors can occur).

The E-ACSL Plugin for Runtime Verification. The E-ACSL [73] plugin provides a way to dynamically verify ACSL requirements at runtime without needing any additional annotations (by performing runtime assertion checking). Given a program annotated with a subset of ACSL, named E-ACSL, the plugin translates it into an instrumented version of the initial C code that could be compiled (which means the code needs to have an entry point), and would fail at runtime if an annotation were violated.

Unlike Wp, E-ACSL cannot, in general, guarantee the absence of errors for all possible executions but detects and precisely locates errors for a concrete input.

```
68 int main(){
69     int res;
70     unsigned pos = 0;
71     unsigned size = 7;
72     int x = 12;
73     int T[] = {2, 42, 616, 10, 25, 12, 1000};
74
75     search(T, size, x, &pos);
76     //@ assert pos == 6;
77     return 0;
78 }
```

Figure 2.10 – *Entry point for the search function*

For instance, we define in lines 68-78 of Figure 2.10 an entry point for the `search` function. More precisely, we initialize a local array and relevant variables in lines 69-73 for the call line 75, and we add a (false) assertion line 76. Indeed, arrays in C are indexed starting at 0, as opposed to starting at 1; thus,

`12` being the 6-th element of array `T`, its index is `5` after the call to `search` line 75.

```
search.c: In function 'main'
search.c:76: Error: Assertion failed:
        The failing predicate is:
        pos == 6.
        With values at failure point:
        - pos: 5
```

Figure 2.11 – *Output provided by* E-ACSL *for the function of Figure 2.10.*

Once we have parsed the code with E-ACSL and compiled it, executing it yields the output provided in Figure 2.11, which reports the assertion we defined as false.

The MetAcsl Plugin for High-Level ACSL Requirements. MetAcsl [99] is an external plugin, not part of the Frama-C distribution (but actively maintained) dedicated to specifying and verifying high-level properties, that is, properties that are supposed to be checked at many points of the code, for which writing the corresponding ACSL annotations manually would be extremely tedious and prone to errors. Such high-level ACSL requirements are called HILAREs (*HI*gh-*L*evel *A*CSL *RE*quirement). To allow for the specification of such properties, MetAcsl essentially extends the syntax of ACSL to describe HILAREs with three main elements:

- A target, which describes the set of functions where the HILARE should hold, defined using the **`\targets`** clause. For instance, line 4 of Figure 2.12 states that the `integrity` HILARE shall be applied to the `foo`, `bar` and `search` functions. For simplicity, it is possible to use the keyword **`\ALL`** to refer to all functions of the analyzed code base, and **`\diff`** to compute set difference.

- A context, corresponding to the kind of program points that are concerned by the HILAREs, declared with the **`\context`** clause. Two important contexts are **`\writing`** and **`\reading`** accesses. For instance,

line 5 of Figure 2.12 states that the `integrity` HILARE should hold each time the memory is written to in the body of the `foo`, `bar` and `search` functions.

- The property itself, defined as an ACSL predicate. For instance, a **`\writing`** context gives rise to a **`\written`** meta-variable denoting the location being written to. For the `integrity` HILARE we consider, line 6 of Figure 2.12 requires that the address of the modified location (given by **`\written`**) must be separated from the memory pointed by pointer `T` given as argument of the functions. The **`\formal`** keyword can be used to reference a common parameter in the target set of functions.
  Similarly, a **`\reading`** context gives rise to a **`\read`** meta-variable denoting the location being read.

The MetAcsl plugin proceeds by generating all ACSL annotations corresponding to each HILARE.

```
1 /*@ meta
2     \prop,
3     \name(integrity),
4     \targets(search, foo, bar),
5     \context(\writing),
6     \separated(\written, \formal(T));
7 */
```

Figure 2.12 – HILARE *example*

For instance, if we apply the requirement defined in Figure 2.12 to generate low level assertions in the `search` function we used in Section 2.2.1, the loop in lines 17-22 of Figure 2.7 becomes the one on lines 54-63 of Figure 2.13. Notably, we can see that MetAcsl generated 3 new assertions in lines 56, 58 and 61 corresponding to our HILARE.

Annotation generators (like RTE and MetAcsl) are naturally combinable with verification plugins. But analyzers can be combined too, which makes Frama-C even more powerful. For example, the value analysis plugin EVA was used [93] to detect pieces of code for which there may exist a vulnerability

```
54 while (i < size) {
55   if (*(T + i) == x) {
56     /*@ assert integrity: meta: \separated(&res, T); */
57     res = 1;
58     /*@ assert integrity: meta: \separated(pos, T); */
59     *pos = i;
60   }
61   /*@ assert integrity: meta: \separated(&i, T); */
62   i ++;
63 }
```

Figure 2.13 – *Generated assertions in the search function from the requirement defined in Figure 2.12*

and E-ACSL to generate a monitor to check at runtime that these potential vulnerabilities are not exploited. The SANTE [31] tool combined EVA with a test generation plugin, PathCrawler [113], to validate or invalidate the generated alarms, while STADY [94] combined Wp with PathCrawler to try to identify a reason of proof failures. A combination of test and proof tools for verification of complex data structures with constraints was studied in [55]. Other examples of combinations can be found in [75].

State of the Art 3

Contents

In this chapter, we present a state-of-the-art review of the existing work over TPMs, TPM library APIs, and the verification techniques relevant to the work carried out in this thesis.

Existing work is split between two main aspects: the reliability and security of the TPM and similar secure components in Section 3.1, and the formal verification aspect of this thesis in Section 3.2.

More specifically, the whole of Section 3.1 is relevant to the motivation of this thesis as a whole. The work presented in Section 3.2.1 is relevant to the contributions of both Chapters 4 and 5. Section 3.2.2 is heavily linked to

the work of Chapter 4, while Section 3.2.3 and Section 3.2.4 are respectively relevant to the contributions of Chapters 5 and 6.

3.1 Reliability and Security of the Trusted Platform Module and Similar Secure Components

TPM Related Safety and Security

Various case studies centered around functionalities of the TPM 2.0 itself and its security have emerged over the last decade. While there are different types of TPM implementations, ranging from discrete TPMs (as a dedicated chip on the platform) to software TPMs, these works focus mostly on hardware and firmware implementations.

3.1.1 Discrete TPMs

Regarding the security of discrete TPMs 2.0, the authors of [115] proposed a security model to capture protections of the TPM on keys and computation environments on authenticated key exchange protocols. They use this model (based on the works of [15] and [29]) to perform a formal analysis of the secure communication interfaces of TPM 2.0.

The authors of [112] propose a security model for the cryptographic support commands in TPM 2.0, proved using the CryptoVerif [24] tool, an automatic protocol prover for specifying and verifying security properties of cryptographic primitives. This work is based on and similar to [111], where the authors had proposed a detailed modelling of the authorization protocols, that they used to formalize and mechanically prove security properties of authorization protocols in the TPM using CryptoVerif.

The authors of [101] defined a model of TPM commands to formalize the session-based HMAC authorization and encryption mechanisms, and define secrecy and authentication properties. They use the SAPIC [76] tool, a security protocol verification platform, and the TAMARIN prover [10], a security protocol verification tool, to analyse their model and identify practical attacking scenarios where the session-bound values might be disclosed and lead to potential attack scenarios.

In [90], Nemec et al. discovered an algorithmic flaw in the construction of primes for RSA key generation (common vulnerability CVE-2017-15361), in version 1.02.013 of the widely-used Infineon RSA library. In particular, the library was found in NIST FIPS 140-2, and CC EAL 5+ certified devices used for a wide range of real-world applications, such as identity cards, and tokens for authentication or software signing, and TPMs.

3.1.2 Firmware TPMs

To the best of our knowledge, there are very few studies that focus on firmware TPMs 2.0, that is, a TPM directly integrated to a CPU, and running in its Trusted Execution Environment [102].

Nevertheless, we can highlight the work of Jacob et al. in [70], where the authors used a voltage-glitching fault injection attack [27] on an AMD Secure Processor's Trusted Execution Environment. Their attack leads to a full TPM state compromise, which they have demonstrated can be used, for instance, against a Full Disk Encryption backed by a firmware TPM in a matter of hours. In that respect, their work "complements the established attacks against discrete TPMs with an even more potent attack against AMD firmware TPMs".

Moghimi et al. [87] discovered that some TPMs 2.0 feature secret-dependent execution times during ECC signature generation. In particular, they discovered that, on an Intel firmware TPM (CVE-2019-11090) as well as an STMicro-

electronics hardware TPM (CVE-2019-16863), these timing leakage can allow an attacker to recover private keys. The issue affects certain Intel firmware TPMs and STM discrete TPMs certified up to CC EAL4+.

3.1.3 Other TPMs, Software Components and Similar Secure Components.

The works previously presented focus on the TPM itself, but to the best of our knowledge, none of the state-of-the-art work aim to verify or test the security of the TPM2-TSS library, let alone any implementation of the TSS.

However, in the past few years, engineers and researchers have started to take an interest in the usability of TPM library APIs. Authors of [98] conducted a survey and established a study environment for the usability and security of TPM library APIs, including but not limited to the tpm2-tools library (which relies on TPM2-TSS). They conducted the first qualitative study targeting TPM library APIs such as TPM2-TSS and found that as designed, they are not developer-friendly.

Svenda et al. [105] conducted the first systematic study of the TPM ecosystem, cryptographic properties and vulnerability mitigation. The study is wide-scale, over 78 TPM versions (manufactured by 6 vendors, including discrete, firmware and integrated TPMs), and focuses on security-relevant properties of TPM 2.0 chips. The study unveiled numerous cases of unreported or inconsistently documented vulnerabilities, but such cases are explainable due to the black-box nature of TPM implementations.

Similar Secure Components

As previously mentioned, the TPM as a standard for a secure cryptoprocessor designed to protect secure hardware from software-based threats, can also be implemented as part of the firmware of a regular processor.

Similarly, a Trusted Execution Environment (TEE) [63, 102] is an area on a chipset that works like a TPM, but is not physically isolated from the rest of the chip. A representative example of a TEE would be the ARM TrustZone [4], a hardware-based security built into SoCs by chip designers who want to provide secure end points and a device root of trust. It is designed to provide a foundation for system-wide security and the creation of a trusted platform. TrustZone technology is commonly used to run trusted boot and a trusted OS to create a TEE in mobile devices. In that respect, TrustZone and TEEs in general are very similar to the TPM.

Using Isabelle/HOL [91, 16], a Higher Order Logic interactive automated theorem prover, Ma et al. [82] formalize and verify a model of memory isolation for ARM TrustZone-based TEEs. They formally explicit and verify the correctness properties of memory management along with the information flow security properties of the memory isolation mechanism.

Previously, using the HOL4 [103] theorem prover, the authors of [44] had formally verified the information flow security for a simple ARM-based separation kernel.

In 2024, the authors of [114] presented a formal specification of TEE APIs using Maude [37], a modeling language and system based on rewriting logic, which supports powerful object-oriented specification. They focused on Trusted Storage API and Cryptographic Operations API, which are foundational to mobile and IoT applications. TEE APIs play similar role for the TEE as that of the TSS for the TPM.

And similarly to TPM library APIs, as explained by Shepherd et al. in [102], "there is an evident deficiency in formal models tailored for TEE specification and its associated APIs. This gap is concerning because without rigorous verification and modeling, the integrity of TEEs could be compromised, potentially exposing vulnerabilities."

3.2 Formal Verification of Real-life Code

3.2.1 Deductive Verification for Real-life Code

Deductive verification on real-life code has been spreading in the last decades, with various verification case studies where bugs were often found by annotating and verifying the code [65]. Such studies include the work of Dordowsky [52], providing feedback on the authors' experience of using ACSL and Frama-C on a real-world example.

The authors of [78] highlight some issues specific to the verification of the Hyper-V hypervisor, and how they can be solved with VCC, a deductive verification tool for C.

The authors of [34] report on the formal verification of memory safety, thread safety and functional correctness of the interprocess communication mechanism of FreeRTOS, using the VeriFast [71] deductive verifier, raising the level of assurance for this mechanism.

Tasche et al. [106] proposed a deductive verification approach for embedded systems that are modeled with SystemC, using VerCors [25], a deductive verification tool for concurrent programs.

In [92], Oortwijn and Huisman found undesired behavior using VerCors in an industrial safety-critical traffic tunnel control system employed in Dutch traffic. In an older version of the code provided by the developers, they detected undesired behavior due to an intricate, unlucky combination of timing and events, demonstrating in the process that formal methods can help detect such behaviors within reasonable time.

In [46, 47], de Gouw et. al. investigate the Java standard library and its main sorting method using KeY [1], a semi-automatic interactive theorem prover for Java that relies on symbolic execution: they found that Java 8's implementation of TimSort performed incomplete checks of the algorithm's invariants, which could lead to an array-out-of-bound exception.

The authors of [107] used KeY to formally specify OpenJDK's `BitSet` class and identified several bugs.

Deductive verification has also been applied to ensure the correctness of smart contracts. For instance, the authors of [89] have shown that the Why3 platform [26, 59] can be used to ensure functional properties and the absence of runtime errors, and to write provably correct contracts that can be compiled on the runtime environment for smart contracts on the Ethereum blockchain.

Another recent work relies on Dafny [79], a powerful verification-friendly programming language. In particular, Cassez, Fuller and Quiles [30] propose a methodology using Dafny to model and reason about Ethereum smart contracts, allowing the authors to specify and prove the main properties of a smart contract. Dafny was also used in [35] to implement, specify and verify the QOI image format.

While all the cited tools have been successful in verifying real-life code, their maturity in terms of usability by engineers differ. Some tools have a scarce documentation and are no longer maintained (for e.g. VCC) while others are very actively maintained and extended (Frama-C, KeY, Dafny, Why3, VeriFast, SPARK) with good documentation and even books [75, 80, 2, 84].

In addition, Frama-C is a recognized verification tool by certification authorities, and a JavaCard virtual machine verified with Frama-C was awarded the highest Evaluation Assurance Level (EAL7) of the Common Criteria security evaluation [51].

3.2.2 Formal Verification of Linked Lists and Other Recursive Data Structures

In Chapter 4, we will use the logical definitions from [22] to formalize and manipulate C linked lists as ACSL logic lists. Another approach to handle C linked lists [21], using Frama-C Wp, relies on a parallel view of a linked list via a companion ghost array. Both approaches were tested in [21, 22] on the

linked list module of the Contiki OS [56], which relies on static allocations and simple structures.

The authors of [88] developed a library in Agda and Coq for linked nested datatypes. This library notably focuses on a set of functions and properties, that can be spread directly from the parameter on which such a nested datatype is built (for instance, the head of a linked list). These properties include the congruence of map, and the satisfaction of a given predicate for at least one, or all, elements of a structure.

Realized in SPARK [84], a deductive verification tool for a subset of the Ada language and also the name of this subset, an approach to the verification of red-black trees [54] is related to the verification of linked lists in Frama-C using ghost arrays including the auto-verification aspects [23]. However, the trees themselves were implemented using arrays , since pointers have only been recently introduced in SPARK [53]. Programs with pointers in SPARK are based on an ownership policy enforcing non-aliasing which makes their verification closer to that of Rust programs than that of C programs.

In the verification of C programs, specifying data-structures in a high-level way requires the possibility to express abstract data types (ADT) in the specification language. That is what the logic types of ACSL basically provide.

In high-level languages, in particular object-oriented languages such as Eiffel or Java, the language itself is expressive enough to define side-effect free immutable ADTs that can be leveraged in specifications [96]. Such data structures are often called models and are related to implementation classes in a fashion similar to how we relate C linked lists with ACSL logic lists in our work. An advantage of such model classes over logic lists is that they are valid classes of the programming language and as such can be executed for dynamic verification tasks. On the deductive verification side, most of the time these classes represent concepts (e.g. sets) that can be translated into elements of theories of provers (and it is possible to verify the faithfulness of

the translation [45]).

In EVE (Eiffel Verification Environment), model classes together with auto-active verification techniques [61] were used to verify a container library [97].

Gladisch and Tyszberowicz [62] specify Java methods on linked data structures, including lists, in JML. However, unlike the previously described approaches, they do not relate Java lists to logical lists or model classes. Instead, they use a pure observer method (hence that can be used in specifications) that returns the object held by the list at a given index.

3.2.3 Formal Verification of High-level Properties.

In [50], Djoudi et al. present a large scale formal verification of global security properties on the C code of the JavaCard Virtual Machine. Using the Wp and MetAcsl plugins of Frama-C, they use deductive verification to prove security properties such as integrity and confidentiality.

Authors of [100] used MetAcsl to define meta-properties on two illustrative case studies, that they verified firstly with deductive verification using Wp, then secondly with testing using E-ACSL. According to the authors, the "study demonstrates that it is easy to check meta-properties at runtime without extra annotation effort thanks to the combination of MetAcsl and E-ACSL, as long as the specified properties are supported by the tools".

3.2.4 Combinations of Approaches

The authors of [58] have studied the combination of LARVA [40, 39], a runtime verification tool, with the test case generator QuickCheck. More specifically, their approach aims to show how testing oracles can be used to derive correct runtime verification monitor. This work focuses on the use of runtime verification to check the behavior of a system with regards to the model used to

generate the monitor.

In [6], Azzopardi et al. proposed a model-based framework, combining static and dynamic verification techniques, serving as base for their future works. In [7, 8], they proposed a combined approach of static and runtime verification applied to Ethereum smart contracts. Their approach introduces a way to perform static tainting, which they use to filter out some of the instrumentation judged inconsequential for dynamic tainting.

In [32] and [3], the authors introduce *StaRVOOrS* (Static and Runtime Verification of Object-Oriented Software), a tool combining static and runtime verification technique, for the specification and verification of data-oriented and control-oriented properties in Java programs. The tool combines KeY as deductive theorem prover, and the *LARVA* [40, 39] runtime verification tool.

In [33], the authors presented a development methodology based on a mix of test driven development and model-based testing enhanced by the integration of a combination of deductive verification and runtime verification on its workflow. They have also elaborated on the benefits obtained from the integration of these techniques alongside software development.

SPARK and KLEE were used by Cluzel et al. in [38] to translate and formally verify the TCP layer of an embedded TCP/IP library. More specifically, the authors translated the TCP layer in SPARK, in which they modeled the behavior of lower layers. They used symbolic execution with KLEE to formally verify the contracts used in SPARK to model the corresponding behaviors.

He et al. [67] extract functional specification from imperative programs using a memory-safe type system and insert dynamic checks into the specification. GRASShopper [95] combines separation logic with an SMT-based verifier. Unlike in our work, GRASShopper does not integrate abstract interpretation based shape analysis (which allows us to infer structural invariants

with MemCAD without having to provide loop invariants for this tool).

Recent work [20] explicitly mentions the interest of an integration of deductive verification with separation logic. To the best of our knowledge, a combination of deductive verification with shape analysis on the source-code level such as the one presented in Chapter 6 was not studied and evaluated before.

Deductive Verification of Safety and Functional Properties

4

Contents

This chapter presents our work on the proof of safety and functional properties in real-life security-critical code using the WP plugin of FRAMA-C for deductive verification. We focus here on a subset of functions of the TPM2-TSS library in charge of several internal operations, such as manipulating linked lists and computing hashes. Higher-level function calls to TPM commands rely heavily on these operations, including but not limited to calls to `TPM2_Create`, a TPM command which may be used to store an encryption key in the TPM.

The work carried out in this chapter are based on the "Towards Formal Verification of a TPM Software Stack" [118] article, appeared at the *18th International Conference on integrated Formal Methods* (iFM 2023), and the "Towards Formal Verification of a TPM Software Stack: Achievements and Opportunities" [120] article, accepted in the journal *Formal Aspects of Computing* (FAC). These contributions are associated with a companion artifact [117].

The remainder of the chapter will be structured as follows. In Section 4.1, we will introduce our target subset of functions to present limitations we have encountered with deductive verification for safety and functional properties on real-life code using FRAMA-C, along with our proposed solution. In Section 4.2, we will present how we handle the proof of properties relying on dynamic memory allocations, whose support by FRAMA-C (up to the latest version, v30.0 Zinc) is limited. The work presented in this chapter uses version 26.1 Iron. In Section 4.3, we will explain issues we encountered with memory management (related to complex data structures, pointers, dynamic allocation and the memory model of WP) and solutions we have proposed. With those explained, we will discuss necessary lemmas needed in Section 4.4. Finally, in Section 4.5 we present our verification results and some of the limitations of our work.

4.1 Example Overview

We illustrate our verification case study with a simplified version of some library functions manipulating linked lists. The illustrative example is split

into Figures 4.1–4.10 that will be explained below step-by-step. Its full code being available in the companion artifact[1], we omit in this example some less significant definitions and assertions which are not mandatory to understand the chapter. This example is heavily simplified yet it is representative for most issues we faced. It contains a main list manipulation function, `getNode` (`esys_GetResourceObject` in the real code), used to search for a resource in the list of resources and return it if it is found, or to create and add it using function `createNode` (`esys_CreateResourceObject` in the real code) if not. Return codes (defined as 32-bits **`unsigned int`** in the TSS) normally used by functions of the library are replaced with arbitrary integers.

Figure 4.1 on page 48 provides the linked list structure as well as logic definitions used to handle logic lists in specifications. Our custom allocator (used by `createNode`) is defined in Figure 4.3 (page 52). Figure 4.4 (page 54) defines a (simplified) context and additional logic definitions to handle pointer separation and memory freshness. The search function is shown in Figure 4.6 (page 59), Figure 4.7 (page 60) and Figure 4.8 (page 62). In Section 4.2, we detail Figures 4.1–4.4.

4.2 Dynamic Memory Allocation of Resource Nodes

4.2.1 Lists of Resources

Lines 11–15 of Figure 4.1 show a simplified definition of the linked list of resources used in the ESAPI layer of the library. Each node of the list consists of a structure containing a handle used as a reference for this node, a resource to be stored inside, and a pointer to the next element. The handle is supposed to be unique. In our example, a resource (omitted in Fig. 4.1) is assumed to contain only a few fields of relatively simple types. The real code uses the more extensive and complex definition of the **`IESYS_RESOURCE`** type (with several

[1] Available (with the illustrative example, all necessary lemmas and their proofs) on `https://doi.org/10.5281/zenodo.13693011`.

```
11 typedef struct NODE_T {
12   uint32_t      handle;   // the handle used as reference
13   RESOURCE      rsrc;     // the metadata for this rsrc
14   struct NODE_T *   next; // next node in the list
15 } NODE_T; // linked list of resource
   ...
25 /*@
26   predicate ptr_sep_from_list{L}(NODE_T* e, \list<NODE_T*> ll) =
27     ∀ ℤ n; 0 ≤ n < \length(ll) ⇒ \separated(e, \nth(ll, n));
28   predicate dptr_sep_from_list{L}(NODE_T** e, \list<NODE_T*> ll) =
29     ∀ ℤ n; 0 ≤ n < \length(ll) ⇒ \separated(e, \nth(ll, n));
30   predicate in_list{L}(NODE_T* e, \list<NODE_T*> ll) =
31     ∃ ℤ n; 0 ≤ n < \length(ll) ∧ \nth(ll, n) == e;
32   predicate in_list_handle{L}(uint32_t out_handle, \list<NODE_T*> ll) =
33     ∃ ℤ n; 0 ≤ n < \length(ll) ∧ \nth(ll, n)->handle == out_handle;
34   inductive linked_ll{L}(NODE_T *bl, NODE_T *el, \list<NODE_T*> ll) {
35     case linked_ll_nil{L}: ∀ NODE_T *el; linked_ll{L}(el, el, \Nil);
36     case linked_ll_cons{L}: ∀ NODE_T *bl, *el, \list<NODE_T*> tail;
37       (\separated(bl, el) ∧ \valid(bl) ∧ linked_ll{L}(bl->next, el, tail) ∧
38       ptr_sep_from_list(bl, tail)) ⇒
39         linked_ll{L}(bl, el, \Cons(bl, tail));
40   }
41   predicate unchanged_ll{L1, L2}(\list<NODE_T*> ll) =
42     ∀ ℤ n; 0 ≤ n < \length(ll) ⇒
43       \valid{L1}(\nth(ll,n)) ∧ \valid{L2}(\nth(ll,n)) ∧
44       \at((\nth(ll,n))->next, L1) == \at((\nth(ll,n))->next, L2);
   ...
48   axiomatic Node_To_ll {
49     logic \list<NODE_T*> to_ll{L}(NODE_T* beg, NODE_T* end)
50       reads {node->next | NODE_T* node; \valid(node) ∧
51                           in_list(node, to_ll(beg, end))};
52     axiom to_ll_nil{L}: ∀ NODE_T *node; to_ll{L}(node, node) == \Nil;
53     axiom to_ll_cons{L}: ∀ NODE_T *beg, *end;
54       (\separated(beg, end) ∧ \valid{L}(beg) ∧
55       ptr_sep_from_list{L}(beg, to_ll{L}(beg->next, end))) ⇒
56         to_ll{L}(beg, end) == \Cons(beg, to_ll{L}(beg->next, end));
57   }
58 */
59
60 #include "lemmas_node_t.h"
```

Figure 4.1 – *Linked list and logic definitions.*

levels of nested structures and unions), covering all possible types of TPM resources. While it does add some complexity to prove certain properties (as some of them may require to completely unfold all resource substructures), it

does not introduce new pointers that may affect memory separation properties, so our example remains representative of the real code regarding linked lists and separation properties.

In particular, we need to ensure that the resource list is well-formed — that is, it is not circular, and does not contain any overlap between nodes — and stays that way throughout the layer. To accomplish that, we use and adapt the logic definitions from [22], given on lines 26–44 and 48–57 of Fig. 4.1. To prove the code, we need to manipulate linked lists and segments of linked lists. Lines 48–57 define the *translating function* `to_ll` that translates a C list defined by two `NODE_T` pointers, its head `beg` and the last pointed node `end`, into the corresponding ACSL logic list of (pointers to) its nodes. By convention, the last element `end` is not included into the resulting logic list. It can be either `NULL` for a full linked list, or a non-null pointer to a node for a *linked list segment* which stops just before that node. The use of the `to_ll` function is essential in our specifications. Indeed, while it is possible to establish the existence of such a list in precondition of a `foo(NODE_T *bl, NODE_T *el)` function with a simple `requires ∃\list<NODE_T*>ll; linked_ll(bl, el, ll)` clause, it would be impossible to refer to that very same list in postcondition.

Lines 34–40 show the *linking predicate* `linked_ll` establishing the equivalence between a C linked list and an ACSL logic list. This inductive definition includes memory separation between nodes, validity of access for each node, as well as the notion of reachability in linked lists.

Lines 26–29 provide predicates to handle separation between a list pointer (or double pointer) and a full list. `\nth(l,n)` denotes the `n`-th element of logic list `l`, while `\length(l)` corresponds to the length of the list. The predicate `unchanged_ll` in lines 41–44 states that between two labels (i.e. program points) `L1` and `L2`, all list elements in a logic list refer to a valid memory location at both points, and that their respective next fields retain the same value. It is used to maintain the structure of the list throughout the code. Line 60 includes lemmas necessary to conduct the proof, further discussed in Sec. 4.4.

4.2.2 Lack of Support for Dynamic Memory Allocation

As mentioned in Section 2.1, per the official specifications, the ESAPI layer — and in general, the TSS as a whole — does not maintain any global state variables between calls to TPM commands. The library code uses `ESYS_CONTEXT` contexts on the ESAPI layer with linked lists of TPM resources, so list nodes need to be dynamically allocated at runtime. The ACSL language provides clauses to handle memory allocations: in particular, `\allocable{L}(p)` states that a pointer `p` refers to the base address of an unallocated memory block, and `\fresh{L1,L2}(p,n)` indicates that p refers to the base address of an unallocated block at label `L1`, and to an allocated memory block of size `n` at label `L2`. Unfortunately, while the Frama-C/Wp memory model[2] is able to handle dynamic allocation (used internally to manage local variables), these clauses are not currently supported. Without allocability and freshness, proving goals involving validity or separation between a newly allocated node and any other pointer is impossible.

```
/*@
  behavior allocation:
    assumes can_allocate: is_allocable(nmemb * size);
    ensures allocation: \fresh(\result, nmemb * size);
    ensures initialization:
                ∈itialized(((char *)\result)+(0..nmemb*size-1));
    ensures zero_initialization:
                \subset(((char *)\result)[0..nmemb*size-1], {0});

  behavior no_allocation:
    assumes cannot_allocate: ¬is_allocable(nmemb * size);
    ensures null_result: \result == \null;

  complete behaviors;
  disjoint behaviors;
*/
extern void *calloc(size_t nmemb, size_t size);
```

Figure 4.2 – *Simplified function contract for* `calloc` *from the* Frama-C *libc.*

[2] that is, intuitively, the way in which program variables and memory locations are internally represented and manipulated by the tool.

4.2.3 Introducing a Static Memory Allocator for Resource Nodes

To circumvent that issue, we define in Fig. 4.3 a bank-based static allocator `calloc_NODE_T` to manually replace calls to `calloc` used to create new nodes of resources in the real-life code. It attributes preallocated cells, following some existing implementations (like the memb module of Contiki [83]). Line 63 defines a node bank, that is, a static array of nodes of size `_alloc_max`. Line 64 introduces an allocation index we use to track the next allocable node and to determine whether an allocation is possible. Predicate **`valid_rsrc_mem_bank`** on line 66 states a validity condition for the bank: `_alloc_idx` must always be between 0 and `_alloc_max`. It is equal to the upper bound if all nodes have been allocated. Predicates lines 67–73 specify separation between a logic list of nodes (resp., a pointer or a double pointer to a node) and the allocable part of the heap, and is used later on to simulate memory freshness.

Lines 76–99 show a part of the function contract for the allocator defined on lines 100–111. The validity of the bank should be true before and after the function execution (lines 77, 79). Line 78 specifies the variables the function is allowed to modify. The contract is specified using several ACSL behaviors. In our case, the provided behaviors are complete (i.e. cover all states allowed by the function precondition) and their corresponding subsets are disjoint (line 98). We show only one behavior (lines 89–97) describing a successful allocation (when an allocable node exists, as stated on line 90). Postconditions on lines 92–93 ensure the tracking index is incremented by one, and that the returned pointer points to the first allocable block. While this fact is sufficient to deduce the validity clause on line 94, we keep the latter as well (and it is actually expected for any allocator). In the same way, lines 96–97 specify that the nodes of the bank other than the newly allocated block have not been modified[3].

In the Frama-C version we used in this section, Wp did not offer a memory model capable of handling byte-level assignments in C objects.

[3]This property is partly redundant with the assigns clause on line 78 but its presence facilitates the verification.

```
62 #define _alloc_max 100
63 static NODE_T _rsrc_bank[_alloc_max];  // bank used by the static allocator
64 static int _alloc_idx = 0;  // index of the next rsrc node to be allocated
65 /*@
66   predicate valid_rsrc_mem_bank{L} = 0 ≤ _alloc_idx ≤ _alloc_max;
67   predicate list_sep_from_allocables{L}(\list<NODE_T*> ll) =
68     ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒
69                          ptr_sep_from_list{L}(&_rsrc_bank[i], ll);
70   predicate ptr_sep_from_allocables{L}(NODE_T* node) =
71     ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒ \separated(node, &_rsrc_bank[i]);
72   predicate dptr_sep_from_allocables{L}(NODE_T** p_node) =
73     ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒ \separated(p_node, &_rsrc_bank[i]);
74 */

76 /*@
77   requires valid_rsrc_mem_bank;
78   assigns _alloc_idx, _rsrc_bank[_alloc_idx];
79   ensures valid_rsrc_mem_bank;
   ...
89   behavior allocable:
90     assumes 0 ≤ _alloc_idx < _alloc_max;
91
92     ensures _alloc_idx == \old(_alloc_idx) + 1;
93     ensures \result == &(_rsrc_bank[ _alloc_idx - 1]);
94     ensures \valid(\result);
95     ensures zero_rsrc_node_t( *(\result) );
96     ensures ∀ int i; 0 ≤ i < _alloc_max ∧ i ≠  \old(_alloc_idx) ⇒
97             _rsrc_bank[i] == \old(_rsrc_bank[i]);
98   disjoint behaviors; complete behaviors;
99 */
100 NODE_T *calloc_NODE_T()
101 {
102   static const RESOURCE empty_RESOURCE;
103   if(_alloc_idx < _alloc_max) {
104     _rsrc_bank[_alloc_idx].handle = (uint32_t) 0;
105     _rsrc_bank[_alloc_idx].rsrc = empty_RESOURCE;
106     _rsrc_bank[_alloc_idx].next = NULL;
107     _alloc_idx += 1;
108     return &_rsrc_bank[_alloc_idx - 1];
109   }
110   return NULL;
111 }
```

Figure 4.3 – *Allocation bank and static allocator.*

To represent as closely as possible the fact that allocated memory is initialized to zero by a call to `calloc` in the real-life code, we initialize each field

of the allocated node to zero with the C code on lines 104–106 and the post-condition on line 95 (equivalent to the postcondition lines `g-h` of `calloc` in Fig.4.2).

4.2.4 Contexts, Separation Predicates and Freshness

As mentioned in Section 2.1, by design, the TSS does not use any global state variable. Therefore, in the TPM2-TSS library and in our illustrative example, pointers to nodes are not passed directly as function arguments, but stored in a context variable (in this case, an equivalent of an `ESYS_CONTEXT` as we are working with the ESAPI layer) and a pointer to the context is passed as a function argument. Lines 113–116 of Fig. 4.4 define a simplified context structure, comprised of an **`int`** and a **`NODE_T`** pointer to the head of a linked list of resources.

We define additional predicates to handle memory separation and memory freshness are defined on lines 118–132. In particular, the **`ctx_sep_from_list`** predicate on lines 118–119 specifies memory separation between a **`CONTEXT`** pointer and a logic list of nodes. Lines 120–121 define separation between such a pointer and allocable nodes in the bank.

In C, a successful dynamic allocation of a memory block implies its *freshness*, that is, the separation between the newly allocated block (typically located on the heap) and all pre-existing memory locations on the heap, stack or static storages (equivalent to line `d` in Fig.4.2). As this notion of freshness is currently not supported by FRAMA-C/WP, we have to simulate it in another way. Our allocator returns a cell in a static array, so other global variables — as well as local variables declared within the scope of a function — will be separated from the node bank. To obtain a complete freshness within the scope of a function, we need to maintain separation between the allocable part of the bank and any other known memory locations accessible through pointers. In our illustrative example, pointers come from arguments including a pointer to a **`CONTEXT`** object (and pointers accessible from it) and a double pointer to a **`NODE_T`** node. This allows us to define a predicate to handle freshness in both function contracts.

```
113 typedef struct CONTEXT {
114   int placeholder_int;
115   NODE_T *rsrc_list;
116 } CONTEXT;
117 /*@
118   predicate ctx_sep_from_list(CONTEXT *ctx, \list<NODE_T*> ll) =
119     ∀ ℤ i; 0 ≤ i < \length(ll) ⇒ \separated(\nth(ll, i), ctx);
120   predicate ctx_sep_from_allocables(CONTEXT *ctx) =
121     ∀ int i; _alloc_idx ≤ i < _alloc_max ⇒ \separated(ctx, &_rsrc_bank[i]);
122
123   predicate freshness(CONTEXT * ctx, NODE_T ** node) =
124     ctx_sep_from_allocables(ctx)
125     ∧ list_sep_from_allocables(to_ll(ctx->rsrc_list, NULL))
126     ∧ ptr_sep_from_allocables(ctx->rsrc_list)
127     ∧ ptr_sep_from_allocables(*node)
128     ∧ dptr_sep_from_allocables(node);
129
130   predicate sep_from_list{L}(CONTEXT * ctx, NODE_T ** node) =
131     ctx_sep_from_list(ctx, to_ll{L}(ctx->rsrc_list, NULL))
132     ∧ dptr_sep_from_list(node, to_ll{L}(ctx->rsrc_list, NULL));
133 */
```

Figure 4.4 – *Context and predicates to handle separation from a list and memory freshness.*

The **freshness** predicate on lines 123–128 of Fig. 4.4 specifies memory separation between known pointers within the scope of our functions and the allocable part of the bank, using separation predicates previously defined on lines 120–121, and on lines 67–73 of Fig 4.3. This predicate will become unnecessary as soon as dynamic allocation is fully supported by Frama-C/Wp.

In the meanwhile, a static allocator with an additional separation predicate simulating freshness provides a reasonable solution to verify the target library. Since no specific constraint is assumed in our contracts on the position of previously allocated list nodes already added to the list, the verification uses a specific position in the bank only for the newly allocated node. The fact that the newly allocated node does not become valid during the allocation (technically, being part of the bank, it was valid in the sense of ACSL already before) is compensated in our contracts by the **freshness** predicate statin g that the new node — as one of the allocable nodes — was not used in the list before the allocation (cf. line 310 in Fig. 4.6). We expect that the migration from our specific allocator to a real-life dynamic allocator — with a more general contract

— will be very easy to perform, as soon as necessary features are supported by Frama-C.

Similarly, the **sep_from_list** predicate on lines 130–132 specifies separation between the context's linked list and known pointers, using predicates on lines 118–119, and on lines 28–29 of Fig 4.1.

4.2.5 Handling Separation: Frame Conditions

In ACSL, frame conditions are expressed at the level of function contracts, using the **assigns** clause to define which non-local memory locations *can* be modified by the function. When such a clause is declared, Wp tries to prove it at the scale of the function. More specifically, it tries to prove that all non-local memory locations that may be modified by the function are included in the set of memory locations listed in the **assigns** clause. The set of memory locations that may be modified is composed of all the locations that may be written to in the code of the function, as well as all the locations provided by the **assigns** clauses of its callees, to provide Wp information about the callees' side effects.

For example, line 78 of Fig. 4.3 specifies that only the allocation index `_alloc_idx` and the next allocable node `_rsrc_bank[_alloc_idx]` can be modified by our custom allocator. Because the node creation function `createNode` in Fig. 4.5 calls the custom allocator on line 186 (in replacement to the original call to `calloc` commented on line 185), the **assigns** clause we have to write for the function contract of `createNode` (cf. line 145) must include the locations specified in the **assigns** clause of `calloc_NODE_T` (that is to say, `_alloc_idx` and `_rsrc_bank[_alloc_idx]`). It must also include `ctx->rsrc_list`, as it is modified on lines 197 and 216, as well as `*out_node`, which is modified on line 232. We do not need to (nor can we) add `new_head` to the clause, as it is defined only locally. We do not need to add `*new_head` either, while the corresponding memory location is not defined locally, because Wp knows from the function contract of `calloc_NODE_T`, with lines 85 and 93, that within the scope of the function, `new_head` is either `NULL`, or `*new_head` is the freshly allocated node (on line 186) from `rsrc_bank` — that is, `_rsrc_bank[_alloc_idx]`), — which is already present in the list. In

```
135 /*@
136   requires valid_rsrc_mem_bank ∧ freshness(ctx, out_node);
145   assigns _alloc_idx, _rsrc_bank[_alloc_idx], ctx->rsrc_list, *out_node;
146   ensures valid_rsrc_mem_bank ∧ freshness(ctx, out_node);
147   ensures sep_from_list(ctx, out_node);
161   behavior allocated:
162     assumes 0 ≤ _alloc_idx < _alloc_max;
167     ensures ctx->rsrc_list == &_rsrc_bank[_alloc_idx - 1];
175   disjoint behaviors; complete behaviors;
176 */
177 int createNode(CONTEXT * ctx, uint32_t esys_handle, NODE_T ** out_node){
178 //@ ghost pre_calloc:;
185   // NODE_T *new_head = calloc(1, sizeof(NODE_T));
    /*library version*/
186   NODE_T *new_head = calloc_NODE_T();
187   /*@ assert unchanged_ll{pre_calloc, Here}(
188               to_ll{pre_calloc}(ctx->rsrc_list, NULL)); */
190   if (new_head == NULL){return 833;}
195   if (ctx->rsrc_list == NULL) {
196     /* The first object of the list will be added */
197     ctx->rsrc_list = new_head;
201     new_head->next = NULL;
202     /*@ assert to_ll(new_head, NULL) == [|new_head|]; */
203   }
204   else {
205     /* The new object will become the first element of the list */
208     new_head->next = ctx->rsrc_list;
216     ctx->rsrc_list = new_head;
223   }
232   *out_node = new_head;
240   return 1610;
241 }
```

Figure 4.5 – *The node creation function, where some annotations are removed.*

the same way, callers of the node creation function need to include its modified memory locations in their respective **assigns** clauses.

Specifying frame conditions is in practice mandatory in order to ensure validity and separation properties. In the general case, if no **assigns** clause is given for a specific function, it means that the function potentially modifies every known memory location, making it practically impossible to prove anything in its callers (where Wp would not have any precise information on the callee's side effects).

Thus, in order to ensure separation properties (as well as functional properties such as the well-formedness of the list), precise **assigns** clauses are required. In our case, in order to propagate to callers separation of a newly allocated node with the list, we need to preserve throughout function contracts the **freshness** predicate, as well as information to associate an allocated node with the allocation bank. In particular, the property on line 93 ensuring that the pointer returned by the allocator (in case of a successful allocation) points to the `_alloc_idx`-th element of array `_rsrc_bank` is required. This is necessary in order to propagate to the node creation function that, given the freshness, the returned allocated node on line 186 of `createNode` is separated from the list. The callee's **assigns** clause helps prove the assertion line 187-188, which indicates that the structure of the list was not changed (as the location potentially written by the call on line 186 was separated from the list). Similarly, we need to ensure the same equality in postcondition of the node creation function with line 167, in order to provide the necessary information to specify and prove **assigns** clauses in callers, and to help propagate to the callers (in this case, the search function of Figs. 4.6 and 4.7) that the newly allocated node is separated from the previous list.

4.3 Memory Management

This section presents how we use the definitions introduced in Sec. 4.2 to prove selected ESAPI functions involving linked lists. We also identify separation issues related to limitations of the Typed memory model of Wp, as well as a way to manage memory to overcome such issues. In this section, we present some parts of Fig. 4.5 and Figs. 4.6–4.10.

4.3.1 The Resource Node Search Function

Figures 4.6 (contract) and 4.7 (body) provide the search operation `getNode` with a partial contract illustrating functional and memory safety properties we aim to verify and judge necessary for the proof at a larger scale. Some

proof-guiding annotations (assertions, loop contracts) have been skipped for readability, but the code is preserved (mostly with the same line numbers). The arguments include a context, a handle to search and a double pointer for the returned node.

In Fig. 4.7, lines 380–416 perform the search of a node by its handle: variable `tmp_node` iterates over the nodes of the resource list, and the node is returned if its handle is equal to the searched one (in which case, the function returns 616 for success).

Lines 420–430 convert the resource handle to a TPM one, call the creation function to allocate a new node and add it to the list as its new head with the given handle if the allocation was successful (and return `833` if not). The new node is returned by `createNode` in `tmp_node_2` (again via a double pointer).

Lines 435–462 perform some modifications on the content of the newly allocated node, without affecting the structure of the list. An error code is returned in case of a failure, and 1611 (with the allocated node in `*node`) otherwise. Lines 450–451, 453–454 and 461 provide some assertions to propagate information to the last return clause of the function, attained in case of the successful addition of the new element to the list.

Compared to the real-life code, we have introduced anonymous blocks on lines 380–416 and 422–452 (which are not semantically necessary and were not present in the original code), as well as two local variables `tmp_node` and `tmp_node2` instead of only one. We explain these code adaptations below.

Contract of the Search Function

Figure 4.6 provides a partial function contract, illustrating two behaviors of `getNode`: if the element was found by its handle in the list (cf. lines 325–326), and if the element was not found at first, but was then successfully allocated and added (cf. lines 355–359), for each of them specific postconditions are stated. For instance, for the latter behavior, lines 369–370 ensure that if a new node was successfully allocated and added to the list, the old head becomes the second element of the list, while line 372 ensures the separation of

```
309 /*@
310   requires valid_rsrc_mem_bank{Pre} ∧ freshness(ctx, node);
313   requires linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));
317   requires sep_from_list(ctx, node) ∧ \separated(node, ctx);
...
321   ensures valid_rsrc_mem_bank ∧ freshness(ctx, node);
...
325   behavior handle_in_list:
326     assumes in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));
...
332     ensures unchanged_ll{Pre, Post}(to_ll(ctx->rsrc_list, NULL));
333     ensures \result == 616;
...
355   behavior handle_not_in_list_and_node_allocated:
356     assumes ¬(in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL)));
357     assumes rsrc_handle ≤ 31U ∨ (rsrc_handle ∈ {0x10AU, 0x10BU})
358             ∨ (0x120U ≤ rsrc_handle ≤ 0x12FU);
359     assumes 0 ≤ _alloc_idx < _alloc_max;
...
369     ensures \old(ctx->rsrc_list) ≠ NULL ⇒
370             \nth(to_ll(ctx->rsrc_list, NULL), 1) ==
    \old(ctx->rsrc_list);
371     ensures linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));
372     ensures sep_from_list(ctx, node);
373     ensures \result == 1611;
374   disjoint behaviors; complete behaviors;
375 */
```

Figure 4.6 – *The search function contract, where some annotations are removed.*

known pointers from the new list. We specify that the complete list of provided behaviors must be complete and disjoint (line 374).

As global preconditions, we notably require for the list to be well-formed (through the use of the linking predicate, cf. line 313), and the validity of our bank and freshness of allocable nodes with respect to function arguments and global variables (cf. line 310). Line 317 requires memory separation of known pointers from the list of resources using the **sep_from_list** predicate, and separation among known pointers using the **\separated** predicate.

As a global postcondition, we require that our bank stays valid, and that freshness of the (remaining) allocable nodes relatively to function arguments and global variables is maintained (cf. line 321). However, properties regarding the list itself — such as the preservation of the list when it is not modified (line

```
376 int getNode(CONTEXT *ctx, uint32_t rsrc_handle, NODE_T ** node) {
377   /*@ assert linked_ll(ctx->rsrc_list, NULL, to_ll(ctx->rsrc_list, NULL));*/
378   int r;
379   uint32_t tpm_handle;
380   { /* Block added to circumvent issues with the WP memory model */
381     NODE_T *tmp_node;
...
401     for (tmp_node = ctx->rsrc_list; tmp_node ≠  NULL;
402         tmp_node = tmp_node->next) {
...
405       if (tmp_node->handle == rsrc_handle){*node = tmp_node; return 616;}
...
415     }
416   }
...
420   r = iesys_handle_to_tpm_handle(rsrc_handle, &tpm_handle);
...
422   { /* Block added to circumvent issues with the WP memory model */
423     NODE_T *tmp_node_2 = NULL;
...
428     r = createNode(ctx, rsrc_handle, &tmp_node_2);
429     /*@ assert sep_from_list(ctx, node);*/
430     if (r == 833) {return r;};
...
435     tmp_node_2->rsrc.handle = tpm_handle;
436     tmp_node_2->rsrc.rsrcType = 0;
437     size_t offset = 0;
...
440     r = uint32_Marshal(tpm_handle, &tmp_node_2->rsrc.name.name[0],
441                        sizeof(tmp_node_2->rsrc.name.name),&offset);
...
443     if (r ≠ 0) {return r;};
444     tmp_node_2->rsrc.name.size = offset;
...
449     *node = tmp_node_2;
450     /*@ assert unchanged_ll{Pre, Here}(
451                 to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
452   }
453   /*@ assert unchanged_ll{Pre, Here}(
454               to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
...
461   /*@ assert sep_from_list(ctx, node);*/
462   return 1611;
463 }
```

Figure 4.7 – *The (slightly rewritten) search function body.*

332), or ensuring it remains well-formed after being modified (line 371) — have to be issued to ACSL behaviors to be proved, due to the way how local variables are handled in the memory model of Wp. The logic list properties are much more difficult for solvers to handle in global behaviors.

4.3.2 Memory Model Limitation: an Unprovable Property

Consider the assertion on line 377 of Fig. 4.7. Despite the presence of the same property as a precondition of the function (line 313), currently this assertion cannot be proved by Wp at the entry point for the real-life version of the function. Basically, the real-life version can be obtained[4] from Fig. 4.7 by removing the curly braces on lines 380, 416, 422, 452. This issue is due to a limitation of the Wp memory model.

Indeed, for such an assertion (as in general for any annotation to be proved), the proof obligation generated by Wp includes a representation of the current state of the program memory. In particular, pointers such as the resource list `ctx->rsrc_list` (and by extension, any reachable node of the list) will be considered part of the heap. To handle the existence of a variable in memory — should it be the heap, the stack or the static segments — Wp uses an allocation table to express when memory blocks are used or freed, which is where the issue lies. For instance, on line 428 of Fig. 4.7, the `tmp_node_2` pointer has its address taken, and is considered as used locally due to **`requires`** involving it in our function contract for `createNode`. It is consequently transferred to the memory model, where it has to be allocated.

Currently, the memory model of Wp does not provide separated allocation tables for the heap, stack and static segments. Using `tmp_node_2` the way it is used on line 428 changes the modification status of the allocation table, which is then considered as modified as a whole. This affects the status of other "allocated" (relatively to the memory model) variables as well, including (but not limited to) any reachable node of the list.

Therefore, the call to `createNode` line 428 of Fig. 4.7 in the real-life code

[4] another difference — removing variable `tmp_node2` declared on line 423 and using `tmp_node` instead — can be ignored in this context.

that uses the address of a local pointer as a third argument is sufficient to affect the status of the resource list on the scale of the entire function. As a result, the assertion on line 377 is not proved.

```
a int getNode(..., NODE_T ** node){
b   // list properties unprovable
c   int r;
d
e   NODE_T *tmp_node;
f   ... // iterate over the list
g
h
i
j   r = createNode(..., &tmp_node);
k   ...
l   *node = tmp_node;
m
n   return 1611;
o }
```

```
a int getNode(..., NODE_T ** node){
b   // list properties proved
c   int r;
d   {
e     NODE_T *tmp_node;
f     ... // iterate over the list
g   }
h   {
i     NODE_T *tmp_node_2 = NULL;
j     r = createNode(..., &tmp_node_2);
k     ...
l     *node = tmp_node_2;
m   }
n   return 1611;
o }
```

Figure 4.8 – *Comparison of the real-life code of* `getNode` *(on the left) and its rewriting with additional blocks (on the right) for proving list properties.*

A Workaround

As a workaround (found thanks to an indication of the WP team) to the aforementioned issue, we use additional blocks and variable declarations. Figure 4.8 presents those minor rewrites (with line numbers in alphabetical style to avoid confusion with the illustrative example). The left side illustrates the structure of the original C code, where the address of `tmp_node` is taken and used in the `createNode` call on line `j`, and the same pointer is used to iterate on the list. On the right, we add additional blocks and a new pointer `tmp_node_2`, initialized to `NULL` to match the previous iteration over the list. Each block defines a new scope, outside of which the pointer used by `createNode` will not exist and side-effect-prone allocations will not happen. It solves the issue.

4.3.3 Additional Proof-Guiding Annotations

Additional annotations (mostly omitted in Fig. 4.7) include, as usual, loop contracts and a few assertions. Assertions can help the tool to establish necessary intermediate properties or activate the application of relevant lemmas. For instance, assertions of lines 450–451 and 453–454 help propagate information over the structure of the linked list (by its logic list representation) outside of each block, and finally to postconditions. Assertions on lines 429 and 461 help propagate separations from the list through the function and its anonymous blocks. Some other intermediate assertions are needed to prove the unchanged nature of the list. Such additional assertions can be tricky to find in some cases and need some experience.

Additional intermediate annotations can be required in order to help propagate separations from the list, and the well-formedness of the list through the introduced anonymous blocks, and from the scope of the function to its postconditions. For instance, assertions in lines 455–461 of Fig. 4.9 are used to deduce and prove, outside the anonymous blocks, functional properties on the list expressed in postcondition.

More specifically, the assertions on lines 460 and 461 (which serve to preserve the freshness and the separation of the context pointer to any reachable node of the list) are proved in this instance using information provided by assertions on line 453–459. Lines 456–458 in particular help prove lines 460–461 by describing the list at this program point as the list provided in entry of the function, to which a new node was added (as the new head of the list) using `createNode`. In the same vein, lines 453–454 that establish that the structure of the previously known list has not changed between the start of the function and the current program point help prove properties of lines 456–458. This is however only achievable at this scale, but not in postcondition of the function. Indeed, given a specific postcondition (e.g. line 371 of Figure 4.6), the corresponding proof obligation generated by Wp will "only" contain information on the program memory (be it from the ACSL annotations or the C code itself) after return clauses but "before postconditions", hence after any locally defined variable has been freed.

```
422   { /* Block added to circumvent issues with the WP memory model */
423     NODE_T *tmp_node_2 = NULL;
424     /*@ assert dptr_sep_from_list(&tmp_node_2,
425                                   to_ll{post_loop}(ctx->rsrc_list, NULL));*/
426     /*@ assert unchanged_ll{Pre, Here}(to_ll{Pre}(ctx->rsrc_list, NULL));*/
427     /*@ assert \separated(node, &tmp_node_2);*/
428     r = createNode(ctx, rsrc_handle, &tmp_node_2);
429     /*@ assert sep_from_list(ctx, node);*/
430     if (r == 833) {/*@ assert sep_from_list(ctx, node);*/ return r;};
431 //@ ghost post_alloc:;
...
437     size_t offset = 0;
...
440     r = uint32_Marshal(tpm_handle, &tmp_node_2->rsrc.name.name[0],
441                        sizeof(tmp_node_2->rsrc.name.name),&offset);
...
444     tmp_node_2->rsrc.name.size = offset;
445     /*@ assert unchanged_ll{post_alloc, Here}(to_ll(ctx->rsrc_list, NULL));*/
446     /*@ assert dptr_sep_from_list(node, to_ll(ctx->rsrc_list, NULL));*/
447     /*@ assert dptr_sep_from_list(node,
448                 to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
449     *node = tmp_node_2;
450     /*@ assert unchanged_ll{Pre, Here}(
451                 to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
452   }
453   /*@ assert unchanged_ll{Pre, Here}(
454               to_ll{Pre}(\at(ctx->rsrc_list, Pre), NULL));*/
455   /*@ assert in_list_handle(rsrc_handle, to_ll(ctx->rsrc_list, NULL));*/
456   /*@ assert ctx->rsrc_list->next == \at(ctx->rsrc_list, Pre);*/
457   /*@ assert \at(ctx->rsrc_list, Pre) ≠ \null ⇒
458       ctx->rsrc_list->next == \nth(to_ll(ctx->rsrc_list, NULL), 1);*/
459   /*@ assert ctx->rsrc_list->handle == rsrc_handle;*/
460   /*@ assert freshness(ctx, node);*/
461   /*@ assert sep_from_list(ctx, node);*/
462   return 1611;
```

Figure 4.9 – *Examples of supplementary annotations needed to propagate properties with additional blocks.*

In this instance, the proof obligation does not contain any information pertaining to other **ensures** clause, but without anonymous blocks, would still contain the variable "allocations" (relatively to the memory model) such as that of `tmp_node_2` that would modify the modification status of the list as a whole in the scope of the function. The list would be considered "modified" again after any **return** clause, as `tmp_node_2` would be freed (relatively to the memory model). From there, most postconditions would be very difficult

to prove. Anonymous blocks and the described assertions allow for the proof of all postconditions, and thus a complete functional proof of the function.

4.3.4 Handling Pointer Casts

Another memory manipulation issue we have encountered comes from the function call on line 440 of Figure 4.9 in `getNode`: after having been added to the resource list, the newly allocated node must have its name (or more precisely, the name of its resource) set from its TPM handle `tpm_handle` (derived from the handle of the node by the function call on line 420). This is done through marshaling using the `uint32_Marshal` function, partially shown on lines 298–306 of Fig. 4.10, whose role is to store a 4-byte unsigned int (in this case, our TPM handle) in a flexible array of bytes (the name of the resource). The function calls `memcpy` on (commented) line 302, which is the source of our issue (a correct endianness being ensured by a previous byte swap in `in`).

```
271 /*@
272   requires \valid(src) ∧ \valid(dest + (0 .. sizeof(*src)-1));
...
279 */
280 void memcpy_custom(uint8_t *dest, uint32_t *src) {
281   dest[3] = (uint8_t)(*src & 0xFF);
282   dest[2] = (uint8_t)((*src >> 8) & 0xFF);
283   dest[1] = (uint8_t)((*src >> 16) & 0xFF);
284   dest[0] = (uint8_t)((*src >> 24) & 0xFF);
285 }
...
298 int uint32_Marshal(uint32_t in,uint8_t buff[],size_t buff_size,size_t *offset){
299   size_t local_offset = 0;
...
302   // memcpy(&buff[local_offset], &in, sizeof (in));
303   memcpy_custom(&buff[local_offset], &in);
...
306 }
```

Figure 4.10 – *Definition for `memcpy` replacement in marshal.*

For most functions of the standard libraries, Frama-C provides basic ACSL contracts to handle their use. However, for memory manipulation functions like `memcpy`, such contracts rely on pointer casts, whose support in Wp is currently limited. To circumvent this issue, we define our own memory copy

function on lines 280–285: instead of directly copying the 4-byte unsigned int pointed by `src` byte per byte using pointer casts using `memcpy`, we extract one-byte chunks using byte shifts and bitmasks (cf. lines 281–284, 303) without casts. Line 272 requires that both source and destination locations are valid, also without casts. This version is fully handled by WP. Current contracts are sufficient for the currently considered functional properties and the absence of runtime errors (and we expect they will be easy to extend for more precise properties if needed).

4.4 Lemmas

When SMT solvers become inefficient (e.g. for inductive definitions), it can be necessary to add lemmas to facilitate the proof. These lemmas can then be directly instantiated by solvers, but proving them often requires to reason by induction, with an interactive proof assistant.

The previous work using logic lists [22] defined and proved several lemmas using the Coq proof assistant. We have added two new useful lemmas (defined in Fig. 4.11) and used twelve of the previous ones to verify both the illustrative example and the subset of real-life functions. However, because the formalization of the memory models and various aspects of ACSL changed between the version of Frama-C used in the previous work and the one we use in this chapter, we could not reuse the proofs of these lemmas. While older Frama-C versions directly generated Coq specifications, more recent Frama-C versions let Why3 generate them. Even if the new translation is close to the previous one, the way logic lists are handled was modified significantly.

In the past, Frama-C logic lists were translated into the lists Coq offers in its standard library: an inductively defined type as usually found in functional programming languages such as OCaml and Haskell. Such types come with an induction principle that allows to reason by induction. Without reasoning inductively, it also offers the possibility to reason by case on lists: a list is defined either as empty, or as built with the `cons` constructor. In recent versions of Frama-C, ACSL logic lists are axiomatized as follows: two functions `nil`

```
lemma in_next_not_bound_in{L}: ∀ NODE_T *bl, *el, *item, \list<NODE_T*> ll;
  linked_ll(bl, el, ll) ⇒ in_list(item, ll) ⇒ item->next ≠  el ⇒
    in_list(item->next, ll);
lemma linked_ll_split_variant{L}:
  ∀ NODE_T *bl, *bound, *el, \list<NODE_T*> l1, l2;
  linked_ll(bl, el, l1 ^ l2) ⇒ l2 ≠  \Nil ⇒
  bound == \nth(l1 ^ l2, \length(l1 ^ l2) - \length(l2)) ⇒
    linked_ll(bl, bound, l1) ∧ linked_ll(bound, el, l2);
```

Figure 4.11 – *New lemmas proved in our verification work (in addition to those in [22]).*

and `cons` are declared, as well as a few other functions on lists, including the length of a list (`length`), the concatenation of two lists (`concat`), and getting an element from a list given its position (`nth`). However, there is no induction principle to reason by induction on lists, and because `nil` and `cons` are not constructors, it is not possible to reason by case on lists in this formalization. It is possible to test if a list is empty, but if not, we do not know that it is built with `cons`. Writing new recursive functions on such lists is also very difficult. Indeed, we only have `nth` to observe a list, while the usual way to program functions on lists uses the head and the tail of a list for writing the recursive case.

Interestingly, when the hypotheses of our lemmas include a fact expressed using **`linked_ll`**, it is still possible to reason by case, because this inductive predicate is translated into Coq as an inductive predicate, and thus provides an induction principle. Consequently, there are only two possible cases for the logic list: either it is empty, or it is built with `cons`. When such a hypothesis is missing, we axiomatized a `tail` function, and a decomposition principle stating that a list is either `nil` or `cons`. These axioms are quite classic and can be implemented using a list type defined by induction. We did not need an inductive principle on logic lists as either the lemmas did not require a proof by induction, or we reasoned inductively on the inductive predicate **`linked_ll`**. However, we proved such an induction principle using only the axioms we added. It is thus available to prove some other lemmas provided in [22] — not needed yet in our current work — that were proved by induction on lists.

Because of these changes, to prove all lemmas we need, we had to adapt all previous proof scripts, and in a few cases significantly. The largest proof scripts

are about 100 lines long excluding our axioms, and the shortest takes a dozen lines. We suggest that the next versions of Frama-C come back to a concrete representation of lists. Thanks to our approach, we expect that the required changes in our proofs of lemmas will remain minimal: we will only have to prove the axioms introduced on `tail` and our decomposition principle.

4.5 Evaluation

4.5.1 Research Questions

RQ 1: Ability to verify properties on real-life code To what extent does this case study aids the expression and verification of safety and functional properties on real-life code not designed with verification in mind?
RQ 2: Efficiency of the approach Are the proposed solutions and workarounds efficient for the verification of safety and functional properties on such code?
RQ 3: Effectiveness of the approach Is the adopted approach effective and sound for the verification of safety and functional properties on such code?
RQ 4: Identification of necessary tool improvements To what extent does this case study help highlight some tool limitations, and improvements necessary to fully verify in the future real-life code not designed with verification in mind?

4.5.2 Verification Results

Verification environment

Proof results, presented in Fig. 4.12, were obtained by running Frama-C 26.1 (Iron) on a desktop computer running Ubuntu 20.04.4 LTS, with an Intel(R) Core(TM) i5-6600 CPU @ 3.30 GHz, featuring 4 cores and 4 threads, with 16GB RAM. We ran Frama-C with options `-wp-par 3` and `-wp-timeout 30`. We used the Alt-Ergo v2.4.3 and CVC4 v1.8 solvers, via Why3 v1.5.1.

RQ 1: Ability to verify properties on real-life code

The verification of functional properties and the absence of runtime errors in real-life code like the TPM2-TSS library with FRAMA-C/WP currently faces several limitations. The tool faces several issues related to its capacity to reason about complex data structures, dynamic memory allocation, linked lists and their separation from other data. We have managed to overcome these limitations after minor simplifications and adaptations of the code.

Both functional properties and the absence of runtime errors (RTE) were proved. Assertions to ensure the absence of runtime errors are automatically generated by the RTE plugin of FRAMA-C (using the `-wp-rte` option). Functional properties include usual properties such as the fact that the well-formedness of the list is preserved, that a new resource has been successfully added to the resource list, that the searched element is correctly found if present, that memory separations between pointers (including, but not limited to, context variables and the list) are preserved, etc.

While the work carried out in this chapter provides solutions and workarounds for several issues encountered in the code of the TPM2-TSS library, such problems (while they may be encountered) are not necessarily representative of all the challenges one may run into when verifying real-life security critical code. However, we believe our work constitutes a good starting point for such verification projects.

RQ 2: Efficiency

In our illustrative example, 282 goals were proved in a total time of 5min13s with 56% proved by SMT solvers, and the rest by the internal simplifier engine Qed of WP and one WP script. The maximum time to prove a goal was 20s.

We also applied the solutions to memory manipulation problems presented in this chapter to our subset of functions of TPM2-TSS. The subset is comprised of over 10 different functions (excluding macros, and interfaces without code whose behaviors needed to be modeled), related to linked list manipulations and to some internal ESAPI feasibility checks and operations (cryptographic operations excluded). On that part of the code, over 659 goals proved in a total

		User-provided ACSL	RTE	Total	
Code subset	Prover	#Goals	#Goals	#Goals	Time
Illustrative example	Qed	105	18	123 (43.62%)	
	Script	1	0	1 (0.35%)	
	SMT	137	21	158 (56.03%)	
	All	243 (86.17%)	39 (13.83%)	**282**	**5m13s**
Library code subset	Qed	274	38	312 (47.34%)	
	Script	5	0	5 (0.76%)	
	SMT	311	31	342 (51.90%)	
	All	590 (89.53%)	69 (10.47%)	**659**	**18m07s**

Figure 4.12 – *Proof results for the illustrative example and the real-life code.*

of 18m07s, 52% were proved by SMT solvers and 47% by Qed. The maximum time to prove a goal was 1min50s.

As for the 14 lemmas we used, 11 were proved by Coq using our scripts, and the remaining 3 directly by Alt-Ergo. Their proof takes 6 seconds in our configuration, with the maximum time to prove a goal being 650ms.

RQ3: Effectiveness and soundness

In our approach, only one Wp proof script was required to conduct the proof on our illustrative example (to help prove and reduce proof times for the post-condition line 96-97 of Figure 4.3).

On the subset of functions of tpm2-tss, only 5 Wp proof scripts were used, when automatic proof either failed or was too slow to conclude in reasonable times (less than 10-15 min). These scripts were generated manually using the GUI of Frama-C (generating such scripts is only needed once, cf Chapter 2). This shows a high level of automation achieved in our project, in particular, thanks to carefully chosen predicates and lemmas. Moreover, while such lemmas are usually tricky to find for the first time, they can be reused and may be useful in other similar projects.

We also used smoke-tests to detect unexpected dead code or possible inconsistencies in the specification, and manually checked that no unexpected cases of those were detected.

RQ4: Identification of necessary tool improvements

The library code is very complex and challenging for verification tools, using complex data structure and relying on a more dynamic management of the memory than in other verification projects. Consequently, the work achieved in this chapter allowed us to identify some tool limitations, and some of the desired tool improvements required in order to achieve a full formal verification of the library.

First and foremost, these limitations include the support of dynamic allocations and related ACSL clauses (notably to express memory allocability and memory freshness, which we do here using the `_alloc_idx` tracking index and the **`freshness`** predicate) required to handle them. Secondly, the implementation of a memory model that works at byte level could enable reasoning with byte accesses, handling zero initialization of data types, and possibly handling pointer casting from complex custom types (such as context variables) to basic C types. Last but not least, a clearer separation of modification statuses of variables between the heap, the stack, and static segments of the memory should solve certain proof issues linked to the modelization of the memory with Wp (which we technically bypass using anonymous blocks), as well as reduce the general proof effort required whenever memory separations between pointers are involved.

While the current real-life code cannot be verified without adaptations, we expect that it will become provable as soon as those improvements of the tool are implemented.

Limitations of the approach

The work presented in this chapter, while encouraging for the prospect of fully formally verifying real-life code not designed with formal verification in mind, has several main limitations.

Indeed, while our approach allows to deal with codes using dynamic allocations, we do not currently have any way to handle freeing of dynamically allocated memory in specifications. Moreover, while it is not too specification intensive in our minimal example, the verification effort and the number of

required ACSL annotations may grow considerably with the size of the code, and more importantly with the number of pointers "visible" on the scale of the verified function. Additionally, the work accomplished in this chapter required some code rewriting which, while minimal and helpful to handle byte accesses for basic C types, will not easily scale to handling interactions with lower layers.

Runtime Verification for Security 5

Contents

The work carried out in the previous chapter on the deductive verification of functional and safety properties for the TPM2-TSS library faced multiple challenges and highlighted several issues which may impair the proof of security requirements on the library. Thus, the purpose of this chapter is to propose an alternative approach to deductive verification for the verification of high-level security requirements, in real-life security-critical code, using the E-ACSL plugin of Frama-C for runtime verification, and the MetAcsl plugin.

The contributions of this chapter are based on the "Runtime Verification for High-Level Security Properties: Case Study on the TPM Software Stack" [119] article, appeared at the *18th International Conference on Tests and Proofs* (TAP

2024). These contributions are associated with a companion artifact [116] available at `https://doi.org/10.5281/zenodo.12773113`.

Function subset.

To construct our target subset of functions, we simplified an integration test of the library for the `Esys_Create` function, to call the `TPM2_Create` command which can be used to import an object onto the TPM without using parameter encryption[1].

To do so, we remove from the TSS dependencies to other function calls to TPM commands for authorizations, parameter encryption (and thus dependencies to an external library such as OpenSSL), and we replace the command transmission interface with a dummy that receives the command buffer sent to the TPM, but replies to higher layers of the TSS with an empty response buffer. We do not change anything else, and keep each dynamic memory allocation, freeing, and memory manipulation operation as is. None of these modifications actually alter the behavior of this subset of functions of TPM2-TSS solely focused on the import of an object onto the TPM.

Thus, we consider the import of a single object acting as sensitive data, that is statically allocated in the test function. In this scenario, the call to `Esys_Create` will succeed in sending the TPM command to the TCTI, and will fail at the very end when receiving the invalid TPM response buffer. Regardless of the failure or success of the ESAPI call to the `TPM2_Create` command, it is expected that the TSS should not retain any sensitive data at the end of the integration test.

Chapter overview

The rest of the chapter will be structured as follows. In Section 5.1, we will introduce our target subset of functions to present limitations we have encountered with runtime verification for high-level requirements on real-life code using Frama-C plugins, E-ACSL and MetAcsl. In Section 5.2 and Section 5.4 we present our approach to model sensitive data for the definition of proper-

[1] More accurately, the `TPM2_Create` command creates a new object in the TPM, to which can be incorporated user defined data.

ties with MetAcsl, and how to define and verify high-level security properties using our representation. Section 5.5 presents the verification methodology we have designed, broken down in a few steps using a small toy example. Finally, in Section 5.6 we present our evaluation and verification results on our subset of functions, and some of the limitations of our work.

5.1 Examples overview

To illustrate our verification approach, we provide our modeling of the memory for sensitive data with Figs. 5.1–5.2. We isolate with Figs. 5.3–5.6 a few small examples of code and structure definitions from our subset representative of the manipulation of the imported sensitive information. Fig. 5.7 provides the security properties we verify in this work. Figs. 5.1–5.2 will be described step-by-step in Section 5.2, and Figs. 5.3–5.7 will be explained in Section 5.4.

5.2 Companion Memory Model for Sensitive Data

Memory representation

Lines 1–4 of Fig. 5.1 show our companion memory representation for sensitive data defined as global variables. Each sensitive data shall be uniquely represented by its memory location and its size. We define this representation statically as part of the code of the subset. We do not use `ghost` statements, as we have identified (and reported to developers) errors in ghost code instrumentation with E-ACSL.

Thus, we define `_all_sens` on line 3 as an array of `char *` pointers to be used to store the addresses of all the possible representations and copies of the imported object and other pieces of sensitive data throughout the layers of the TSS. With the `_len_sens` array of `int` on line 2, we couple each of these addresses with the size (in bytes) of the associated sensitive data. For convenience, the macro on line 1 defines `MAX_SENS`, a maximum number of sensitive data to

```
1  #define MAX_SENS 100
2  int _len_sens[MAX_SENS];
3  char * _all_sens[MAX_SENS];
4  int _nb_sens;
5
6  // define _sens_##vartype##_##usage##_idx here
7  int _sens_inSensitive_esys_Create_idx = -1;
8  int _sens_inSensitiveData_esys_Create_ctx_idx = -1;
9  int _sens_outPrivate_esys_Create_idx = -1;
10 int _sens_inSens_sys_Create_cmdbuff_idx = -1;
11
12 bool is_sens(void *ptr, int size, int idx)
13 {
14   if (0 > idx ∨ idx ≥ MAX_SENS){return false;}
15   else {
16     return(_all_sens[idx] == ptr ∧ _len_sens[idx] == size);
17   }
18 }
19
20 void remove_sens(int idx){_len_sens[idx] = 0;}
21
22 int add_as_sens(void *ptr, int size)
23 {
24   int idx;
25   if(_nb_sens ≥ MAX_SENS ∨ _nb_sens < 0){idx =-1;}
26   else {
27     _all_sens[_nb_sens] = (char*) (ptr);
28     _len_sens[_nb_sens] = size;
29     idx = _nb_sens++;
30   }
31   return idx;
32 }
```

Figure 5.1 – *Companion memory representation for sensitive data*

be modeled. The **_nb_sens** variable on line 4 serves to determine whether it is possible to add another piece of sensitive data to our representation: it stores the number of stored pointers (which gives also the index of the next available one). Lines 7–10 define indices we use to record where each type of sensitive information is located in our model. We define `_sens_A_B_idx` as the index of `A` or information `A` in location `B`, where `A` is the name of the original variable referred to by **_all_sens**`[_sens_A_B_idx]`, and `B` describes whether the original is a local variable in a specific layer, or stored in the ESAPI context, or in a command buffer.

Consequently, we consider that `_sens_A_B_idx` refers to sensitive data if we have `0≤_sens_A_B_idx<MAX_SENS` and `0<_len_sens[_sens_A_B_idx]`.

The corresponding memory block is assumed to be safely readable and writable, that is to say, `\valid(_all_sens[_sens_A_B_idx] + (0.._len_sens[_sens_A_B_idx]-1))` is supposed to be true in this case (it will be verified in the global invariant given in Fig. 5.2, discussed below).

As well as the model itself, we also define three helper functions to manage sensitive data in the model. In particular, we define **`is_sens`** in lines 12–18, **`remove_sens`** in line 20, and **`add_as_sens`** in lines 22–32:

- Given a pointer `ptr`, a size `size` and an index `idx`, **`is_sens`** is used to determine whether the memory block of size `size` at address `ptr` corresponds to the sensitive data at index `idx` of the representation, returns `true` in this case, and `false` otherwise. The **`is_sens`** function shall be used as a way to determine whether a piece of sensitive data exists in the representation before removing it with **`remove_sens`**, thus to determine whether **`remove_sens`** is safe to call.

- The **`remove_sens`** function in line 20 sets to `0` the size **`_len_sens`**`[idx]` of the sensitive data stored at index `idx` of our model, to express that `idx` is not considered to refer to sensitive data anymore[2]. **`remove_sens`** is to be typically used, for instance, when dynamically allocated sensitive data is freed, overwritten with non-sensitive data, or when we exit the scope of automatically allocated variables (such as local variables) with sensitive data.

- The **`add_as_sens`** function defined in lines 22–32 serves to add a new piece of sensitive data to the representation, given a pointer `ptr` and a size `size`. **`add_as_sens`** first checks with line 25 if **`_nb_sens`** is between `0` and `MAX_SENS`, that is to say if it is possible to add new data to the representation. If it is, `ptr` (resp. `size`) is added to the **`_all_sens`**

[2]For simplicity, when an element is removed, we do not shift the following array elements to the left. This feature can be easily added and particularly useful for long-running programs, where pieces of sensitive data are frequently added and removed.

array (resp. `_len_sens` array) at index `_nb_sens` (our tracking index), which is then incremented by 1. The new value of `_nb_sens` is then returned (to be used to set the indices defined in lines 7-10 of Figure 5.1).

Validity invariant

In order to ensure that all sensitive data added to our representation is always safe for the TSS (as well as for our helper functions) to read and write, we define as a MetAcsl meta-property a global validity invariant `valid_sensitive` in Fig. 5.2.

```
1 /*@ meta \prop, \name(valid_sensitive),
2   \targets(\ALL),
3   \context(\strong_invariant),
4   ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 < _nb_sens ≤ MAX_SENS ⇒
5     0 < _len_sens[i] ⇒
6     \valid(_all_sens[i] + (0 .. _len_sens[i] - 1)); */
```

Figure 5.2 – *Global invariant for the validity of memory accesses of modeled sensitive data*

As per the `\targets` clause in line 2, the invariant applies to all the defined functions in the representative subset, including the three helper functions of the companion model. As the `\context(\strong_invariant)` clause line 3, the property must hold at every program point in each target function. Subsequently, with lines 4–6, we express that for any index `i` for which sensitive data is defined — that is to say, `∀ i ∈ [0, _nb_sens - 1]` (with `_nb_sens ∈ [0, MAX_SENS]`), such that `_len_sens[i]>0` — we must have `\valid(_all_sens[i] + (0.._len_sens[i] - 1))`. Notice `_all_sens` is defined as an array of pointers of type `char` in line 3 of Fig. 5.1; as mentioned in Section 2.2.1, the validity of a pointer is related to its type, therefore so is the size of the pointed regions. The use of type `char` here allows to measure the pointed memory in multiples of `sizeof(char)`, thus in bytes.

In other words, it should always be possible to dereference any piece of sensitive data in the model, as long as they are part of it.

This property is expected to always be verified. Any violation of this invariant would imply that sensitive data was deleted or freed from the program

before it was removed from our representation, which should only happen in one of two situations: either we neglected to remove sensitive data from the model when it should have been, or such data is freed earlier than the code should have.

5.3 Determining and Handling the Lifetime of Sensitive Data

This section presents how we use the definitions introduced in Sec. 5.2 to define and verify high-level security properties such as the integrity and the confidentiality of sensitive data (in our case, the object sent to the TPM). We also identify issues related to the limited support of ACSL clauses in E-ACSL related to logic labels, function pointers, and which currently prevent some of the possible extensions of the pool of verifiable security properties. In this section, we detail Figs. 5.3–5.7.

Adding sensitive data on higher-level layers

Figure 5.3 provides an example of addition of sensitive data from the ESAPI layer to our representation. More specifically, we focus on the `store_input_parameters` function in charge of storing the data from `inSensitive` inside the ESAPI context `esysContext`, before it can be sent to the lower layers. The ESAPI context as defined in TPM2-TSS has specific fields to store the sensitive data to be imported, and its address.

As its name suggests, the `inSensitive` parameter is supposed to contain sensitive data (in our case, the encryption key, as well as some authorization value) to be sent for import to the TPM. Let us assume here that this sensitive information, passed as argument of `store_input_parameters`, has already been added to the model in previously called functions or in callers. The function checks with line 7 whether `inSensitive` is a `NULL` pointer, and sets the corresponding field in the context to `NULL` if such is the case. Otherwise, with lines 9–11, it adds a copy of the pointed memory — that is to say, the sensitive

```
1  /** Store sensitive data inside the ESYS_CONTEXT */
2  static void store_input_parameters (
3      ESYS_CONTEXT *esysContext,
4      const TPM2B_SENSITIVE_CREATE *inSensitive)
5  {
6          _read_sens[_sens_inSensitive_esys_Create_idx] = 1;
7    if (inSensitive == NULL) {esysContext->in.Create.inSensitive = NULL;}
8    else {
9      esysContext->in.Create.inSensitiveData = *inSensitive;
10     esysContext->in.Create.inSensitive =
11       &esysContext->in.Create.inSensitiveData;
12   /*For E-ACSL*/
13     /* We add the part of context containing the imported sensitive
14        object to our representation of sensitive data. */
15
16     _sens_inSensitiveData_esys_ctx_idx =
17       add_as_sens(esysContext->in.Create.inSensitive,
18              (int) sizeof(esysContext->in.Create.inSensitiveData));
19   /*end E-ACSL*/
20         _read_sens[_sens_inSensitive_esys_Create_idx] = 0;
21   }
22 }
```

Figure 5.3 – *Adding sensitive data from ESAPI to the representation*

data itself — to the context (in the `in.Create.inSensitiveData` field), and sets its address within the context (in the `in.Create.inSensitive` field).

This new instance of the imported data is different from the one passed as argument of the function. As it refers to a memory location different from that of `inSensitive`, we need to add it to our representation as well. In lines 15–19, we add the new instance of the sensitive data, using its address given by the `in.Create.inSensitive` field, and its size defined as the **`sizeof`** of the `in.Create.inSensitiveData` field. The reason we can use **`sizeof`** is because the **`TPM2B_SENSITIVE_CREATE`** structure only contains fields of fixed size, as shown in Fig. 5.4. More specifically, this type is defined in lines 6–9 with two fields, an **`int`**-like field `size`, and a `sensitive` field of type **`TPMS_SENSITIVE_CREATE`** defined in lines 2–5 and comprised solely of two subfields `userAuth` and `data`, each defined with an **`int`**-like field and a fixed-size array, following the template in line 1.

While knowing the precise type definition is not paramount to handling sensitive data on high-level layers, knowing it is essential in understanding

how such data is transferred to lower-level layers, and how to handle sensitive data on SAPI and TCTI. As the latter acts as a transmission layer to send the command buffer formed on the SAPI to the TPM (resp. to send the response buffer from the TPM to the SAPI), sensitive data is stored in the same way in both layers.

```
1 typedef struct {UINT16 size;BYTE buffer[TYPE_MAX_SIZE];} TPM2B_TYPE;
2 typedef struct  {
3   TPM2B_AUTH userAuth;             //TPM2B_TYPE-like type
4   TPM2B_SENSITIVE_DATA data;       //TPM2B_TYPE-like type
5 } TPMS_SENSITIVE_CREATE;
6 typedef struct  {
7   UINT16 size;     // ignored by marshal according to the TCG specifications
8   TPMS_SENSITIVE_CREATE sensitive;
9 } TPM2B_SENSITIVE_CREATE;
```

Figure 5.4 – *Partial type definition used for* `inSensitive`

Adding sensitive data on low-level layers

Figure 5.5 presents an example of addition of sensitive data, to our model, in a function of SAPI, a lower-level layer of the library. We focus here on the `Tss2_Sys_Create_Prepare` function on line 1. This library function is the `Prepare` variant of the SAPI call to the command, which partially constructs the command buffer by copying and saving data passed as argument to the `cmdBuffer` of the `sysContext` context. It is typically used to transfer information such as `inSensitive` (used as argument in line 4) from higher layers to lower layers.

The snippet of code in lines 28–38 is in charge of such an operation. In particular, line 30 checks whether the `inSensitive` pointer is `NULL` or not. If such is the case, the value `0` is copied byte per byte to the command buffer to indicate that the command will not have an imported object. Otherwise, it will copy the relevant content of `inSensitive` to the `ctx->cmdBuffer` buffer, starting at the `ctx->nextData`-th byte.

More specifically, the function call in lines 34–35 is used to read the subfields of `inSensitive`, and write their content byte per byte in the command buffer, starting at the `ctx->nextData`-nth byte. The function writes,

```
1  TSS2_RC Tss2_Sys_Create_Prepare(
2    TSS2_SYS_CONTEXT *sysContext,
3    TPMI_DH_OBJECT parentHandle,
4    const TPM2B_SENSITIVE_CREATE *inSensitive,
   ...
8  {
   ...
28   size_t inSens_offset = ctx->nextData;
29         _read_sens[_sens_inSensitiveData_esys_Create_ctx_idx] = 1;
30   if (¬inSensitive) {
31     rval = Tss2_MU_UINT16_Marshal(0, ctx->cmdBuffer,
32                     ctx->maxCmdSize,&ctx->nextData);
33   } else {
34     rval = Tss2_MU_TPM2B_SENSITIVE_CREATE_Marshal(
35       inSensitive, ctx->cmdBuffer, ctx->maxCmdSize, &ctx->nextData);
36   }
37   if (rval)
38     return rval;
39   /* We add the part of the command buffer containing the imported
40     sensitive object to our representation of sensitive data. */
41
42   int sys_sensitive_size = (int) sizeof(UINT16) +
43                 (int) sizeof(inSensitive->sensitive.userAuth.size) +
44                 (int) inSensitive->sensitive.userAuth.size +
45                 (int) sizeof(inSensitive->sensitive.data.size)+
46                 (int) inSensitive->sensitive.data.size;
47         _read_sens[_sens_inSensitiveData_esys_Create_ctx_idx] = 0;
48   _sens_inSens_sys_Create_cmdbuff_idx =
49       add_as_sens(ctx->cmdBuffer + inSens_offset, sys_sensitive_size);
   ...
98 }
```

Figure 5.5 – *Adding sensitive data from SAPI to the representation*

in this order, (and as defined in Fig. 5.4), the size `userAuth.size`, the first `userAuth.size` bytes of the `userAuth.buffer` array, the size `data.size`, and the first `data.size` bytes of the `data.buffer` array.

This function is part of the Marshal API, and is used to "translate" objects of type **`TPM2B_SENSITIVE_CREATE`** into a buffer of bytes fit for data transmission. This marshal function does not write `inSensitive->size` inside the command, but computes the size in bytes of the written area, and writes it. We define the total size in bytes of the written area in the command buffer with the `sys_sensitive_size` variable (defined in lines 42–46). We

base this definition on the behavior described in the official TSS SAPI Specification, rather than the behavior of this marshaling function as it is implemented in the TPM2-TSS library. With line 28, we insert some code to store in the `inSens_offset` variable the offset where the marshaling stores the transformed sensitive data in the command buffer. We then add the SAPI command buffer instance of `inSensitive` to our model , using its address `ctx->cmdBuffer + inSens_offset` and its size `sys_sensitive_size`, with lines 48–49.

Removing sensitive data and handling its lifetime.

In order to express high-level security properties on sensitive data in this subset, it is necessary to determine the lifetime of such data, so as to ascertain when such properties need to be verified. Thus, if we take for example the integrity of the imported key, for which we need to ensure it is only written or modified when it is supposed to be (as will be explained later on with Fig. 5.7), we need to precisely define for every instance of the key when they are freed or overwritten, and thus when their representation should be removed from the companion model.

```
392   /* Allocate memory for response parameters */
393   if (outPrivate ≠ NULL) {
394     *outPrivate = calloc(sizeof(TPM2B_PRIVATE), 1);
395     if (*outPrivate == NULL) {
396       return_error(TSS2_ESYS_RC_MEMORY, "Out␣of␣memory");
397     }
398     _sens_outPrivate_esys_Create_idx =
399       add_as_sens(*outPrivate, (int) sizeof(TPM2B_PRIVATE));
400   }
...
505   if (outPrivate ≠ NULL){
506     if((*outPrivate) ≠ NULL) {
507       /*remove the sensitive data from the model before freeing*/
508       if(is_sens(*outPrivate, size, (int) sizeof(TPM2B_PRIVATE)))
509         remove_sens(_sens_outPrivate_esys_Create_idx);
510       free((void*) (*outPrivate));
511       (*outPrivate)=NULL;
512     }
513   }
```

Figure 5.6 – *Usage example for* **`remove_sens`** *on* ESAPI

We illustrate this with the two snippets of code in Fig. 5.6 (lines 392–400 and lines 505–513 refer to the same file[3] in the considered subset of code). Lines 389–392 allocate memory to be used to store sensitive data `outPrivate` returned by the TPM about the imported key, before the TSS receives the response buffer (which is empty in our example). Similarly to what we presented in Figs. 5.3 and 5.5, we add it to the companion model with lines 393–394. Lines 505–513 provide a typical usage by the library of dynamic deallocation of memory previously allocated (with line 389) for high-level representations of command or response parameters. The **`if`** statements in lines 505 and 506 correspond to checks to ensure the call to `free` line 510 is safe, before setting the corresponding pointer to `NULL` with line 511.

5.4 Defining and Verifying High-level Security Properties

Defining security properties

To define high-level security properties over specific pieces of sensitive data (in our case, the key to be imported), we first need to add such data to the companion model as previously described in this section. Consequently, because our representation relies on both the address and the size in bytes to represent and describe sensitive data, we can verify high-level properties over all relevant memory locations by specifying and verifying the properties in question using their companion representation. Moreover, using this intermediary model is essential for two main reasons:

- Using a common global view of sensitive information leads to a smaller specification effort compared to using each instance of sensitive data separately, provided they all are added to the model when they are defined, and removed when they are freed or overwritten.

[3] `./artifact/code/src/tss2-esys/api/Esys_Create.c` in the companion artifact

- The MetAcsl plug-in is very efficient at expressing properties over data that is visible at a global level, and as such, it mostly relies on global variables to express such properties. However, by design and per the official TCG specification, the library avoids the usage of global state variables[4], and thus global variables to store sensitive information one would want to send to the TPM. Although it *is* possible to use the **\formal** operator to refer to arguments of target functions, it is limited in that it is not possible to use it to refer to existing memory locations unrelated to function arguments.

```
1  int _write_sens[MAX_SENS];
2  int _read_sens[MAX_SENS];
3
4  /*@ meta \prop, \name(all_sens_data_integrity),
5    \targets(
6      \diff(\ALL,
7        \union({tcti_fake_init, doLog, doLogBlob, getLogFile, // F-C crash
8          tcti_proxy_transmit_fake_tpm, tcti_fake_receive,
9          tcti_proxy_receive_fake_tpm })), //function pointers support
10   \context(\writing),
11   ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 ≤ _nb_sens ≤ MAX_SENS ⇒
   //idx within bounds
12     0 < _len_sens[i] ⇒ //sensitive data exists
13     _write_sens[i] ≠ 1 ⇒ //sensitive data should not be writable
14     \separated(\written, (char*)_all_sens[i]+(0..(size_t)(_len_sens[i]-1)));*/
15
16 /*@ meta \prop, \name(all_sens_data_confidentiality),
17   \targets(
18     \diff(\ALL,
19       \union({tcti_fake_init, doLog, doLogBlob, getLogFile, // F-C crash
   ...
27         tcti_proxy_receive_fake_tpm})})),
28   \context(\reading),
29   ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 ≤ _nb_sens ≤ MAX_SENS ⇒
30     0 < _len_sens[i] ⇒
31     _read_sens[i] ≠ 1 ⇒ //sensitive data should not be readable
32     \separated(\read, (char*)_all_sens[i]+(0..(size_t)(_len_sens[i]-1)));*/
```

Figure 5.7 – *Meta-property for the integrity and confidentiality of sensitive data*

Figure 5.7 shows the high-level requirements expressing the integrity and

[4] in practice, any TSS implementation following the specification should not use any global state variable either

the confidentiality of sensitive data represented by the companion model, as well as two additional arrays with lines 1–2, `_write_sens` and `_read_sens`, that we use in the properties defined in lines 4–32 to help determine whether a sensitive data should be read or written. We define the integrity of sensitive data as such:

- We name our high-level requirement `all_sens_data_integrity` with line 4.
- In lines 6–9, we define the set of functions where the property should hold, that is to say the set of functions in which the property shall be verified. We define it as our full subset of functions, from which we exclude the subset defined in lines 7–9. We exclude the functions listed in line 7 because of parsing issues (leading to crashes or the use of unsupported features of E-ACSL) with MetAcsl, and the functions lines 8–9 are removed from the target because they are called using function pointers, whose support is limited in E-ACSL.
- Line 10 establishes a writing context with `\context(\writing)`, so the property defined in lines 11–14 must hold whenever a variable or a memory location is written, where `\written` refers to the written location.
- The predicate lines 11–14 states that, provided the tracking index `_nb_sens` is within bounds, for any index `i` referring to existing sensitive data, *if* the data stored at index `i` is not supposed to be writable (line 13), then the written location must be separated from any sensitive data.
- While MetAcsl does provide a clause to locally relax strong invariants, it is currently not natively possible to locally relax other types of high-level requirements without excluding entire functions. In order to do so, for specific data stored at index `idx` in the model, we manually set `_write_sens[idx]` to `1` in parts of code where the data should be written, before it is written, and we set it back to `0` afterwards.

Similarly, we may define the confidentiality of sensitive data, with lines 16–32, using the **`\reading`** context, and by stating any read location must be separated from existing sensitive data in the target set of functions defined with lines 19–27. Where needed, in parts of the code where we need to locally relax the confidentiality, we manually set **`_read_sens`**`[idx]` for data of index `idx` to `1`: we do it for instance in functions like `store_input_parameters` of Fig. 5.3 (on lines 6 and 20) and `Tss2_Sys_Create_Prepare` in Fig. 5.5 (on lines 29 and 47), as they can read already existing sensitive data to create new instances and copies of it.

MetAcsl can then be used to translate these high-level requirements (and the validity invariant defined in Fig. 5.2) into local ACSL assertions at each relevant program point. For example, for the integrity, an assertion of the defined separation will be added to the code before each writing operation, where **`\written`** will be replaced by the written location. For the validity invariant, `valid_sensitive`, an assertion is generated at every program point of the target functions. Once generated, the resulting annotated code can be given as input to E-ACSL to be instrumented (as we have already excluded unsupported parts of the code), and to convert the logical ACSL assertions into executable assertions. E-ACSL can then use GCC to compile this version of the code. The high-level requirements are verified if all the generated executable assertions are verified at runtime.

Handling standard memory manipulation functions

By default, Frama-C uses its own C standard library, providing a number of its functions with ACSL contracts to specify their behaviors. It is however possible for an analyzed program to use libc functions that have not been defined (yet) in the Frama-C libc, for some of the defined types to mismatch those of the system libc, or even for other plug-ins to generate code that may be unusable by or incompatible with E-ACSL. In order to instrument and compile a program with E-ACSL, it can in turn be necessary to force the usage of the standard library of the system. Our subset of functions (and tpm2-tss in general) represents a case of such obligation.

However, some of these contracts include information about which parts of the memory may be modified by the corresponding function, through the use of the **assigns** clauses. In particular, **assigns** A, B **\from** C, D; specifies that the memory locations referred by variables A and B *can* be modified by the corresponding function, which reads variables C and D to do so. While E-ACSL does not currently support this clause, MetAcsl can rely on such annotations to generate assertions for high-level requirements using the **\writing** or the **\reading** contexts[5]. More specifically, if we consider our integrity (resp. confidentiality) meta-property applied to a function f specified with a minimal **assigns** A **\from** C clause, whenever f is called, MetAcsl will generate an assertion so that the written (resp. read) memory location referred by A is separated from any defined sensitive data. Without these specifications, MetAcsl will not be able to cover all situations where some high-level requirements should be checked.

```
1 #include <string.h>
2 /*@ assigns ((char*)dest)[0..n - 1] \from ((char*)src)[0..n-1];*/
3 void *memcpy_e_acsl (void * dest, const void *src, size_t n)
4 {return memcpy(dest, src, n);}
```

Figure 5.8 – *memcpy wrapper function*

Thus, for functions of the libc such as memcpy that may read or write (non-local) variables, and whose contracts and implementations are unavailable to be used by MetAcsl for compatibility reasons with E-ACSL, we define wrapper functions with minimal contracts to describe such memory manipulation as shown in Fig. 5.8.

5.5 Verification Methodology

Our approach has allowed us to suggest a methodology for the verification of high-level security properties (such as integrity and confidentiality) of sensitive data on real-life code. We believe it to be relatively easy to apply for a

[5]In practice, the presence of an **assigns** clause in the contract of any function foo is only used by MetAcsl to generate assertions in callers whenever foo is called *only* if foo is declared but not defined

user who does not possess in-depth prior knowledge of the target code. Figures 5.9–5.15 will be used in this section to summarize the verification methodology over a small toy example of code.

```
1  void swap(int* a, int* b){
2      int tmp = *a;
3
4      *a = *b;
5
6      *b = tmp;
7  }
8
9  int main(){
10     int some_local_var;
11     int a = 42;
12
13     int b = 37;
14     swap(&a, &b);
15
16
17     some_local_var = 1;
18     return 0;
19 }
```

Figure 5.9 – *Toy example program for the verification of high-level security properties over sensitive data*

The methodology can be broken down into the six main steps we will describe using the toy example in Fig. 5.9. This example consists of two main functions, a `swap` function defined in lines 1–7, and a `main` entry point defined in lines 9–19. `swap` takes two pointers `a` and `b` of type **`int`** as argument and swaps their stored values. The `main` function defines two local **`int`** variables `a` and `b`, swaps them with a call to `swap` line 14, and then modifies a local variable `some_local_var` before returning `0`.

In this exemple, let us assume that the `a` variable defined in line 11 and initialized to `42` is a piece of sensitive data that should not be illegally modified, regardless of whether or not it is ever read or copied.

The steps of our methodology are as follows:

1. **Define the memory representation** to be used for sensitive data, as shown in Fig. 5.1.

2. **Identify** at a high level of abstraction the pieces of **sensitive data** whose security has to be ensured, and **add them into the model**.

```
9  int main(){
10     int some_local_var;
11     int a = 42;
12     _sens_exData1_idx = add_as_sens(&a, sizeof(a)); // because 42 is sensitive
13     int b = 37;
14     swap(&a, &b);
15     some_local_var = 1;
16
17
18     return 0;
19 }
```

Figure 5.10 – *Step 2: Identifying the pieces of sensitive data*

In our example, our target sensitive data is variable `a` introduced in line 11, which we add to the companion model with the call to **`add_as_sens`** we added in line 12 of Fig. 5.10.

3. **Define security properties** over sensitive data. Defining the **integrity** and the **confidentiality** as shown in Fig. 5.7 makes for a good **starting point**, and should help refine the approach in the following steps.

```
12 /*@ meta \prop,
13   \name(integrity),
14   \targets(\ALL),
15   \context(\writing),
16   ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 ≤ _nb_sens ≤ MAX_SENS ⇒
17                 0 < _len_sens[i] ⇒
18     _write_sens[i] ≠ 1 ⇒
19     \separated(\written, (char *)_all_sens[i] + (0 .. (_len_sens[i] - 1)));
20 */
```

Figure 5.11 – *Step 3: Defining high-level security properties over sensitive data*

In our example, we only aim to verify the integrity of the given sensitive data, which we define in Fig. 5.11 similarly to what we did in Fig. 5.7.

4. Parse, instrument, compile and execute the target code with the defined properties:

(a) If the code lacks an entry point, it is up to the user to define a `main` function and to ensure the code can be compiled.

(b) Parse the code and the properties with MetAcsl, instrument the resulting annotated code and compile the output with E-ACSL, and execute.

5. If possible, **use the output** of the executed code to **refine the previous definitions**:

(a) A detected violation of the validity invariant should indicate that a piece of sensitive data should not be considered as sensitive at the reported program point

(b) A detected violation of confidentiality should indicate either an "illegal" read of sensitive data, or a "legal" read of sensitive data not yet handled by the current definitions. In the latter case, the user should locally relax the property by setting the corresponding element of **`_read_sens`** to `1` before the read, and setting it back to `0` after data has been read.

(c) Similarly, a detected violation of integrity should indicate either an "illegal" write of sensitive data, or a "legal" write of sensitive data not yet handled by the current definitions. In the latter case, the user should locally relax the property by setting the corresponding element of **`_write_sens`** to `1` before the write, and setting it back to `0` after data has been written.

In our example, we detect two different violations of properties: a breach of integrity in lines 1–4 of Fig. 5.12, reporting the issue takes place at line `258` of the `swap` function in the code parsed with MetAcsl, as well as a violation of the validity invariant in lines 21–25 of Fig. 5.12, reporting the issue takes place at line `305` of the `swap` function in the code parsed with MetAcsl.

We chose to focus on the validity issue first: the ouput indicates the issue occurs in postcondition of the `main` function, indicating some memory

```
1  ./meta_swap_add.c: In function 'swap'
2  ./meta_swap_add.c:258: Error: Assertion failed:
3          The failing predicate is:
4          integrity/meta:
   ...
21 ./meta_swap_add.c: In function 'main'
22 ./meta_swap_add.c:305: Error: Postcondition failed:
23         The failing predicate is:
24         valid_sensitive/meta:
25                     int i;
26   0 ≤ i < _nb_sens ⇒
27   0 < _nb_sens ≤ 100 ⇒
28   0 < _len_sens[i] ⇒ \valid(_all_sens[i] + (0 .. _len_sens[i] - 1)).
29         With values at failure point:
30         - valid_sensitive: meta:
31   ∀ int i;
32     0 ≤ i < _nb_sens ⇒
33     0 < _nb_sens ≤ 100 ⇒
34     0 < _len_sens[i] ⇒ \valid(_all_sens[i] + (0 .. _len_sens[i] -
          1)): 0
```

Figure 5.12 – *Step 5: Detected violations of validity and integrity*

```
9  int main(){
10     int some_local_var;
11     int a = 42;
12     _sens_exData1_idx = add_as_sens(&a, sizeof(a));
13     int b = 37;
14     swap(&a, &b);
15     some_local_var = 1;
16     if(is_sens(&a, sizeof(a), _sens_exData1_idx))
17         remove_sens(_sens_exData1_idx);
18     return 0;
19 }
```

Figure 5.13 – *Step 5: Refining the handling of the lifetime of sensitive data*

which only exists within the scope of `main` is still part of our model when exiting the function. Because we never remove the sensitive data from representation, we know that all that is left to do is to remove it before exiting the function, which we do by adding the necessary function calls in lines 16-17 of Fig. 5.13.

6. **Repeat steps 4 and 5** until either no violation of properties are detected,

or all reported violations correspond to breaches of integrity or confidentiality.

```
1  ./meta_swap_remove.c: In function 'swap'
2  ./meta_swap_remove.c:258: Error: Assertion failed:
3          The failing predicate is:
4          integrity/meta:
5                  ∀ int i;
6    0 ≤ i < _nb_sens ⇒
7    0 ≤ _nb_sens ≤ 100 ⇒
8    0 < _len_sens[i] ⇒
9    _write_sens[i] ≠ 1 ⇒
10   \separated(a, (char *)_all_sens[i] + (0 .. _len_sens[i] - 1)).
11         With values at failure point:
12         - integrity: meta:
13   ∀ int i;
14     0 ≤ i < _nb_sens ⇒
15     0 ≤ _nb_sens ≤ 100 ⇒
16     0 < _len_sens[i] ⇒
17     _write_sens[i] ≠ 1 ⇒
18     \separated(a, (char *)_all_sens[i] + (0 .. _len_sens[i] - 1)): 0
```

Figure 5.14 – *Step 6: Remaining detected violation of integrity*

By running the verification once more after modifying the code as shown in Fig. 5.13, the newly generated output in Fig. 5.14 shows the breach of integrity previously found in the swap function is still present. We compare the code parsed with MetAcsl with the original one (in Fig. 5.9), and from lines 10 and 18 of the output, we can trace back the issue to the writing of `*a` in line 4 of the `swap` function.

```
1 void swap(int* a, int* b){
2     int tmp = *a;
3         _write_sens[_sens_exData1_idx]= 1; //locally relaxed
4     *a = *b;
5         _write_sens[_sens_exData1_idx]= 0; //end local relaxation
6     *b = tmp;
7 }
```

Figure 5.15 – *Step 6: Refining the handling of the integrity of sensitive data*

We assume here that we have some form of software specification, according to which the writing line 4 is normal. As shown in Fig. 5.15, we thus modify the code to locally relax our constraint of integrity by manually

setting the corresponding index in `_write_sens` to `0` before the write (in line 3) and back to `0` after the write (in line 5). Running the verification once more after that yields no other error. Note that in this example, we only focus on the integrity and not the confidentiality of sensitive data. Because of that, we do not handle copies of the defined sensitive data. However, defining and verifying the confidentiality as well allows for the detection of any other instance of the provided sensitive data in the code.

5.6 Evaluation

5.6.1 Research Questions

The evaluation of our approach addresses the following research questions:

- **RQ1 (Expressiveness)**: To what extent does this verification approach allow for the expression and the verification of high-level security properties on real-life code?
- **RQ2 (Effectiveness)**: To what extent is this verification approach *effective* for the verification of high-level security properties on real-life code?
- **RQ3 (Efficiency)**: To what extent is this verification approach *efficient* for the verification of high-level security properties on real-life code?

Verification target.

To answer the proposed research questions, we use a subset of 86 functions of the tpm2-tss library, of which 50 consist in marshal functions for different data types. The remaining 36 functions correspond to unique TSS operations involved in the import operation (without parameter encryption) defined by the `Esys_Create` function/the `TPM2_Create` command. The entire subset consists of approximately 20k lines of code, about 12k of which corresponding to interfaces and the remaining 8k corresponding to actual function implementations, with marshal functions defined as non-unfolded macros.

We construct the target using one of the integration tests of the library, and we remove from the subset cryptographic operations and their dependencies, and dependencies to libraries other than the C standard library. We also implement a few different versions of the code where we introduce faults designed to locally break the integrity and the confidentiality as they are defined in Fig. 5.7.

Tools Used.

We use Frama-C v28.1(Nickel) with the corresponding version of E-ACSL, and MetAcsl v0.6. We use MetAcsl to parse the defined high-level security properties and generate corresponding low-level ACSL assertions on each version of the code (with and without introduced errors). We then instrument the generated annotated code with E-ACSL to translate the annotations into C code to allow their verification at runtime. The ouput code is compiled by GCC through E-ACSL.

Verification Environment.

For conducting our evaluation, we parse, instrument, compile and run each version of the code. All evaluations are performed on a desktop computer running Ubuntu 20.04.6 LTS, with an Intel(R) Core(TM) i5-6600 CPU @ 3.30 GHz, featuring 4 cores and 4 threads, with 16GB RAM.

RQ1: Expressiveness

The MetAcsl plug-in allows users to define and express properties over data that are visible at a global level, thus mostly relying on global variables as targets. However, software specifications such as the TSS — and therefore library API code such as that of the tpm2-tss library — may require to avoid the usage of global state variables. Our approach allows rendering such data visible at a global level, and thus usable as target variables for the definition of high-level properties with MetAcsl, to be verified with other Frama-C plug-ins such as

Wp or E-ACSL. In this way, it *extends* the capacities of MetAcsl, whose usage to handle data not visible at a global level is very challenging.

In our work, we focus on two high-level security properties, integrity and confidentiality of sensitive data, verified at runtime with E-ACSL. We do not define properties such as invariants relying directly on the values stored in modeled data between two program points, as their verification can appear difficult due to the current limitations of E-ACSL, and since lightweight runtime verification of security properties is an important goal by itself. However, such properties should be provable with Wp, but would require a bigger specification effort.

RQ2: Effectiveness

In our approach, we use the E-ACSL plug-in to verify at runtime the assertions generated by MetAcsl for each security property. Using this method requires a much smaller specification effort on such real-life code than that of deductive verification for properties defined in ACSL.

Indeed, we define the integrity and the confidentiality of sensitive data by using the `\separated` clause to express memory separations, which are typically hard to prove with Frama-C on C codes relying on more dynamic management of the memory, as the Wp plug-in does not currently rely on any aspect of separation logic. Proving assertions about memory separations can require a lot of intermediary specifications, as highlighted in [81],[118] and Chapter 4.

To ensure the integrity of sensitive data as defined in Fig. 5.7, MetAcsl generates about 800 low-level assertions on our subset of functions, and approximately 4200 low-level assertions for the confidentiality.

In our `main` function, we randomized the size and content of the imported key as target sensitive data, to make it possible to run the verification multiple times in a row without needing to instrument and recompile the code between runs. We did not detect any breach of integrity or confidentiality of sensitive data in our subset "as is". To ensure our approach is effective, we applied it to a few variations of the code in which we introduced errors to detect.

For instance, we modified a few marshal functions to wrongly report where sensitive data was written. We also imagined a scenario in which a command buffer sent to the TPM could have been temporarily stored in a global variable for reissuing. We detected the resulting breaches of integrity in the former, and of confidentiality in the latter. In addition, we tested and detected situations where errors in specifications lead to sensitive data being added several times to the model, and the RTE safeguard checks performed by E-ACSL for each separation clause allowed for the detection of situations when sensitive data was wrongly removed from the representation after it had been deleted in the code.

RQ3: Efficiency

Parsing, instrumentation, compilation and execution times were similar for the main subset and its variations. Generating low-level assertions from high-level properties with MetAcsl took only about 8 seconds. Instrumentation with E-ACSL and compilation with GCC took much longer, reaching up to 15 minutes for the instrumentation, and up to 5 minutes for the compilation. Running a produced executable takes less than a second. Hence, the execution time remains compatible with executing numerous test cases on the same version of the code, as it should be typically required for a rigorous dynamic verification.

On average, the processing and evaluation of a version of the code takes approximately 19 minutes. We believe that this is considerably shorter than what would probably be required for deductive verification with Wp, and requires much less time and effort for specification (from our experience with deductive verification). This should be confirmed with a more rigorous study focusing on comparing directly the usage of Wp and E-ACSL for verifying the proposed properties. Additionally, the complete processing of a single version of the code is single-threaded, and thus can be further accelerated by running in parallel (for various versions of the code, or for different test cases).

5.6.2 Threats to Validity

Internal validity

To verify the low-level specifications generated by MetAcsl from the high-level properties, we used E-ACSL to check them at runtime. However, while our verification approach should not modify the behavior of the original code as is, we can not guarantee it to be on the same level as if using ghost code. Indeed, as previously shown in Section 2.2.1, in ACSL, ghost variables and statements are declared for specification purposes only. While ghost statements can read the content of ghost and non-ghost variables, they can only modify ghost variables, and cannot be used in the original code and outside specifications, and cannot modify the control flow of the original program either. In our approach, as we have identified (and reported to developers) errors in ghost code instrumentation with E-ACSL, we define our companion memory model with regular C code.

External validity

The conclusions about the expressiveness, effectiveness and efficiency of our approach might not hold for a complete TSS call to a TPM command, other programs using linked data structures, or with dependencies to external libraries. However, it should still be possible to conduct at least a partial verification by excluding problematic parts of the code.

5.6.3 Tool Limitations

While the work presented in this chapter allowed us to suggest a methodology for the verification of high-level security properties, encouraging in the prospect of fully formally verifying real-life code such as tpm2-tss, it also permitted us to identify various tool limitations related to E-ACSL, or its combined usage with MetAcsl. We briefly present some of them here, and a summary of these limitations is available in Figure 5.16.

For instance, tool limitations with E-ACSL include the usage of `ghost`

statements, whose support by E-ACSL is currently limited, leading to incorrect instrumentation and wrong verification results. Other partially supported features leading to incorrect instrumentation or even crashes include the support of function pointers, variadic functions, the direct comparison between two `struct` or two `union` types, and low-level accesses to bitfields.

Limitation	Relevant Plug-in	Location	Issue	Temporary workaround
`ghost` code	E-ACSL	Companion memory model annotations	Wrong instrumentation misplacing annotations	Companion model defined statically
Function pointers	MetAcsl	`tcti_fake_init`	Crash during parsing by MetAcsl	Function excluded from target set of HILAREs
Logic definitions	E-ACSL	Linked lists of resources	Warnings for annotations ignored by E-ACSL	Removed functions from subset
`assigns` clauses	MetAcsl for E-ACSL		For `\writing` contexts but not checked by E-ACSL	Proof with Wp of helper functions companion memory model
Variadic functions	E-ACSL	`doLog` log functions	Ignored annotations	Functions excluded from target sets of HILARE
		calls to `vsnprintf`	limited support	
Comparisons between `struct` or `union`	E-ACSL for MetAcsl		Limited support	Avoid properties and invariants over complex datatypes
Predicates with labels	E-ACSL for MetAcsl	HILAREs with labels and/or named predicates	Warnings for annotations ignored by E-ACSL	Avoid named predicates and more complex meta-properties
Bitfield pointers	E-ACSL	`_Prepare` functions and other low-levels accesses to bitfields	Warnings for annotations ignored by E-ACSL	Temporarily ignored for now
External dependencies	E-ACSL for MetAcsl	Crypto functions and functions calling external libraries		Removed functions from subset

Figure 5.16 – *Summary of tool limitations with the combinations of* MetAcsl *with* E-ACSL.

As for the combination of MetAcsl with E-ACSL, a notable limitation would be the lack of support for **assigns** clauses by E-ACSL. Indeed, such clauses can be used by MetAcsl to generate annotations. In particular, for an HILARE using the **\writing** context and whose target includes a `foo` function, if `foo` has a function contract, then MetAcsl may use its **assigns** clause to generate low-level assertions whenever `foo` is called.

Combining Approaches

6

Contents

The work carried out in Chapter 4 on the deductive verification (of functional and safety properties) on tpm2-tss highlighted several issues to conduct automatic proofs on code with recursive data structures and relying on dynamic memory allocations. While deductive verification tools face such issues, abstract interpretation tools based on separation logic and shape analysis can efficiently reason about such structures, but cannot deal with large classes of properties handled by deductive verification.

In a similar fashion, the work presented in Chapter 5 on the runtime verification of high-level security requirements, in real-life security critical code,

highlighted limitations of the E-ACSL plugin related to the support of various features of ACSL (including but not limited to labels), which may in turns be handleable by the deductive verification provided by Wp.

Thus, in this chapter we aim to propose preliminary work on combining deductive verification with other validation techniques on real-life code not designed with verification in mind. We focus here on two combinations: Frama-C/Wp and MemCAD [104], a shape analysis tool, for the verification of low-level safety and functional properties, and Wp with E-ACSL, for the verification of high-level security properties.

Function subsets and chapter overview

We use the same subset of functions as that of Chapter 4 to explore the combination between Wp and MemCAD for low-level properties in Section 6.1, and the same subset as that of Chapter 5 for the combination between Wp and E-ACSL for high-level requirements in Section 6.2.

The chapter will be structured as follows. In Section 6.1, we will present our verification approach combining the deductive verification provided by Wp with the shape analysis used by MemCAD, and apply it to the subset of functions of Chapter 4 on linked lists in the tpm2-tss library. In Section 6.2, we take inspiration from Section 6.1 to introduce deductive verification to support the runtime verification of high-level security properties of Chapter 5, extending the work presented in that chapter.

6.1 Deductive Verification and Shape Analysis for Safety and Functional Properties

Motivation

The main idea of our approach is to prove structural and separation properties in MemCAD, and to then assume them in Frama-C/Wp.

The aim is to increase the level of automation of deductive verification with Wp, and overcome some of its limitations.

We apply our approach to a simplified version of the subset of functions on linked lists we used in Chapter 4, where we demonstrated that deductive verification of the library functions manipulating linked lists was relatively hard, and required many additional lemmas and assertions.

6.1.1 Specifications for Deductive Verification with Frama-C/Wp

We illustrate our verification approach on the running example[1] of Fig. 6.1–6.6, where Fig. 6.1–6.2 and Fig. 6.4–6.6 present the C code and ACSL specifications for our example, and Fig. 6.3 shows MemCAD definitions which will be explained later on.

The main predicate of Fig. 6.1 is the inductively defined predicate **`linked_ll`** (Lines 10–19) stating that a linked list (segment) of **`int`** values (defined on Lines 1–2) from pointer `bl` to pointer `el` (excluded) is a well-formed list represented by an ACSL logical list `ll`. The lists and predicates used here are simplified versions of the ones from Section 4.2.1, obtained by removing the TPM resources from the list. In other words, `ll` contains the pointers to the cells of that list segment (or the whole list if `el` is `NULL`). In the inductive case (`linked_ll_cons`) overlapping list cells (or cyclic lists) are avoided by requiring that the first cell `bl` is separated from all the other cells in the list including `el`, so the list is well-formed. Using C labels and the predefined **`\at`** predicate, **`unchanged_ll`** states that a logic list does not change between two program points (Lines 20–24). Finally, Lines 25–36 define an axiomatic function **`to_ll`** that constructs a logic list from a C linked list. While writing "**`requires`** ∃ **`\list`**`<cell>ll; ` **`linked_ll`**`(*pl, NULL, ll);`" instead of Line 72 of Figure 6.2 would have been possible, the scope of the existential quantifier is just this line. Therefore, `ll` cannot be used in the postconditions, hence the need for **`to_ll`**. Lines 69–85 of Fig. 6.2 provide an ACSL contract for function `list_push` (detailed below) that adds a new value into a linked list (cf. Lines 1–2 of Fig. 6.1), allocating a new cell. We focus here on smaller functions like

[1] Available in a companion artifact on `http://doi.org/10.5281/zenodo.10497923`

```
1  typedef struct cell_s {struct cell_s* next; int data;} cell;
2  typedef cell* list;
3  /*@
4    predicate ptr_sep_from_list{L}(cell* c, \list<cell*> ll) =
5      ∀ ℤ n; 0 ≤ n < \length(ll) ⇒ \separated(c, \nth(ll, n));
6    predicate dptr_sep_from_list{L}(cell** c, \list<cell*> ll) =
7      ∀ ℤ n; 0 ≤ n < \length(ll) ⇒ \separated(c, \nth(ll, n));
8    predicate in_list{L}(cell* c, \list<cell*> ll) =
9      ∃ ℤ n; 0 ≤ n < \length(ll) ∧ \nth(ll, n) == c;
10   inductive linked_ll{L}(cell *bl,cell *el,\list<cell*> ll) {
11     case linked_ll_nil{L}:
12       ∀ cell *el; linked_ll{L}(el, el, \Nil);
13     case linked_ll_cons{L}:
14       ∀ cell *bl, *el, \list<cell*> tail;
15         (\separated(bl, el) ∧ \valid(bl) ∧
16         linked_ll{L}(bl->next, el, tail) ∧
17         ptr_sep_from_list(bl, tail)) ⇒
18           linked_ll{L}(bl, el, \Cons(bl, tail));
19   }
20   predicate unchanged_ll{L1, L2}(\list<cell*> ll) =
21     ∀ ℤ n; 0 ≤ n < \length(ll) ⇒
22       \valid{L1}(\nth(ll,n)) ∧ \valid{L2}(\nth(ll,n)) ∧
23       \at((\nth(ll,n))->next, L1) == \at((\nth(ll,n))->next, L2) ∧
24       \at((\nth(ll,n))->data, L1) == \at((\nth(ll,n))->data, L2);
25   axiomatic cell_to_ll {
26     logic \list<cell*> to_ll{L}(cell* beg, cell* end)
27       reads {node->next | cell* node;
28              \valid(node) ∧ in_list(node, to_ll(beg, end))};
29     axiom to_ll_nil{L}: ∀ cell *node;
30       to_ll{L}(node, node) == \Nil;
31     axiom to_ll_cons{L}: ∀ cell *beg, *end;
32       (\separated(beg, end) ∧ \valid{L}(beg) ∧
33       ptr_sep_from_list{L}(beg, to_ll{L}(beg->next, end))) ⇒
34         to_ll{L}(beg, end) ==
35         \Cons(beg, to_ll{L}(beg->next, end));
36   }
37 */
38 #include "lemmas_min.h"
```

Figure 6.1 – *Types and* ACSL *predicates for linked lists.*

`list_push` to illustrate our approach, before applying it to the illustrative example we used in Chapter 4.

Let us now detail the contract of `list_push` (its code is detailed below). The preconditions state that `pl` is a valid pointer to a list (Line 71), separated from every element in the list (Line 72), and refers to a linked list verifying

```
70 /*@
71   requires \valid(pl);
72   requires dptr_sep_from_list(pl,to_ll(*pl, NULL));
73   requires linked_ll(*pl, NULL, to_ll(*pl, NULL));
74   assigns *pl, \at(**pl, Post);
75   ensures dptr_sep_from_list(pl, to_ll(*pl, NULL));
76   ensures linked_ll(*pl, NULL, to_ll(*pl, NULL));
77   ensures \result ∈ {0, 1};
78   ensures \result == 0 ⇒
79     unchanged_ll{Pre, Post}(to_ll(*pl, NULL));
80   ensures \result == 0 ⇒ *pl == \old(*pl);
81   ensures \result == 1 ⇒
82     to_ll(*pl, NULL) == ([|*pl|] ^ to_ll(\old(*pl), NULL));
83   ensures \result == 1 ⇒
84     unchanged_ll{Pre, Post}(to_ll(\old(*pl), NULL));
85   ensures \result == 1 ⇒ (*pl)->next == \old(*pl);
86   ensures \result == 1 ⇒ (*pl)->data == data;
87 */
88 int list_push(list* pl, int data) {
89   cell* c = calloc_cell();
90   if (¬c) return 0;
91   mc_chk_sep_cell_plist(c, pl);
92   //@ admit ptr_sep_from_list(c,to_ll(*pl,NULL));
93   //@ admit \separated(pl, c);
94   //@ ghost Alloc:;
95   c->next = *pl;
96   //@ assert unchanged_ll{Alloc,Here}(to_ll{Alloc}(*pl,NULL));
97   c->data = data;
98   //@ ghost Link:;
99   *pl = c;
100  /*@ assert unchanged_ll{Link,Here}(
101    to_ll{Link}(\at(*pl,Pre),NULL)); */
102  mc_chk_plist(pl);
103  //@ admit dptr_sep_from_list(pl,to_ll(*pl,NULL));
104  //@ admit linked_ll(*pl,NULL,to_ll(*pl,NULL));
105  return 1;
106 }
```

Figure 6.2 – *Function* `list_push` *with contract.*

the inductive predicate **`linked_ll`** (Line 73). Line 74 specifies that the only locations the function is allowed to modify are `*pl`, the head pointer of the list, and **`\at`**`(**pl, Post)`, the first element of the list at the exit point, i.e. the freshly allocated cell. We cannot reference the new list cell at the entry point because it is not allocated yet. In postconditions, the returned value indicates

whether or not the allocation is successful (Line 77). Regardless of the success, we expect the list invariants to hold (Lines 75–76). In case the allocation fails, we expect the pointer `*pl` and the list contents to be unchanged (Lines 78–80). If it succeeds, we expect the list to be composed of the new cell followed by the old list (Lines 81–82), the old list being unchanged (Lines 83–84), and the fields of the new cell, `next` and `data`, resp., to point to the old list (Line 85) and to contain the expected value (Line 86).

6.1.2 Specifications of linked lists for Shape Analysis with MemCAD

```
a ll_cell<0,0> :=
b | [0]
c   - emp
d   - this = 0
e | [2 addr int]
f   - this->0 |-> $0 * $0.ll_cell()
        *
g     this->4 |-> $1
h   - alloc(this, 8) & this ≠ 0.
i
j plist<0, 0> :=
k | [1 addr]
l   - this->0 |-> $0 * $0.ll_cell()
m   - alloc(this, 4) & this ≠ 0.
```

```
n cell<0,0> :=
o | [0]
p   - emp
q   - this = 0
r | [2 addr int]
s   - this->0 |-> $0 * this->4 |->
        $1
t   - alloc(this, 8) & this ≠ 0.
u
v cell_plist<0,0> :=
w | [2 addr addr]
x   - this->0 |-> $0 * $0.cell() *
y     this->4 |-> $1 * $1.plist()
z   - alloc(this, 8) & this ≠ 0.
```

Figure 6.3 – *Inductive predicates for* MemCAD.

The purpose of MemCAD [104] is to automatically infer precise invariants about programs manipulating complex data structures. It is based on shape analysis [49], a static code analysis technique that discovers and verifies properties of recursive, dynamically allocated data structures. It relies on separation logic and abstract interpretation. Unlike in Wp, the analysis is global.

To use MemCAD on linked lists defined on Lines 1–2 of Fig. 6.1, we must first define an inductive predicate expressing a structural invariant of a well-formed linked list, such as predicate `ll_cell` on Lines a–h of Fig. 6.3. A list, i.e. a pointer to a list cell, satisfies the predicate in two cases. Each case defines a memory separation formula and additional constraints. In the first

case, the pointer is null (Line d) and no specific memory separation is required (Line c). This case has no additional arguments (cf. `[0]` on Line b). The second case has two (implicitly existentially quantified) arguments: an address and an integer (Line e), denoted, resp., by `$0` and `$1` in the rest of the case. The pointer is non null and refers to a valid memory block of 8 bytes (Line h), assuming a 32-bit system. Lines f–g define the values of the fields `next` and `data` (at offsets 0 and 4) as `$0` and `$1`, and require separation between those fields and the rest of the list. The separation is expressed by the separating conjunction "`*`" [104]. Notice that "`...*$0.ll_cell()*...`" on Line f specifies separation recursively, for all list cells reached by the predicate via the inductive case. The user can insert the instruction `_memcad("add_inductive(l,ll_cell)");` to *assume* that list `l` respects predicate `ll_cell`, or `_memcad("check_inductive(l,ll_cell)");` to *check* the same property in MemCAD.

Predicate `cell` on Lines n–t is very close to predicate `ll_cell` except that it only defines one list cell without recursion. Predicate `plist` on Lines j–m expresses that a double pointer to a list cell (i.e. of type `list*`) is valid, refers to a well-formed list and is separated from its cells. Predicate `cell_plist` is explained in Section 6.1.3.

6.1.3 Combined Approach

Shape Analysis Assisted Verification

To prove complex memory-related annotations with Wp on real-life code the user typically has to manually annotate the code with many additional carefully chosen assertions establishing structural invariants and separation properties at several intermediate program points, and to add numerous lemmas to facilitate reasoning about them (whose proof must usually be done manually in Coq, an interactive proof assistant). Our approach proposes to let MemCAD deal with the structural invariants of recursive data structures and separation properties, and to admit them in Wp at some key points.

In order to use both tools in this way, we first need to informally establish

the equivalence between MemCAD and Wp inductive predicates. For MemCAD, predicate `ll_cell` (Lines a–h of Fig. 6.3) specifies that each element of the list is a valid cell, is separated from every other cell of the list and the list is null-terminated. This is equivalent to the `linked_ll` predicate for Wp (Lines 10–19 of Fig. 6.1) when we consider the whole list. Indeed, when `el` is `NULL`, this predicate also means that every list cell is valid and separated from any other list cell, and the list is null-terminated. Explicit separation conditions in the ACSL predicate for Wp are expressed by the separating conjunction in the MemCAD counterpart. (Notice that separation of `bl` with `NULL` on Line 15 is trivial.) The sequence of list elements, expressed by a logic list in ACSL and used to prove functional properties about the contents of the list (cf. Lines 80–81 of Figure 6.2) in Wp, does not need to be specified for MemCAD, which we only use to reason about structural properties.

```
40 /*@ assigns \nothing;*/
41 void mc_chk_plist(list* pl) {
42   _memcad("check_inductive(pl,plist)");
43 }
44
45 typedef struct {cell* c; list* pl;} cell_plist;
46
47 /*@ assigns \nothing;*/
48 void mc_chk_sep_cell_plist(cell* c, list* pl) {
49   cell_plist tmp;
50   tmp.c = c; tmp.pl = pl;
51   cell_plist* ptmp = &tmp;
52   _memcad("check_inductive(ptmp,cell_plist)");
53 }
```

Figure 6.4 – *Auxiliary* MemCAD *checks for linked lists.*

To check if invariants hold in MemCAD, we define check functions shown in Fig. 6.4. These functions are specified to be side-effect-free (cf. Lines 40, 47) to prevent interference with the proof in Wp.

The first function, `mc_chk_plist` (Lines 41–43), checks that `pl` respects the `plist` predicate, i.e. is a valid pointer to a well-formed list from which it is separated (Line 42, see also Lines j–m of Fig. 6.3).

The goal of the second function, `mc_chk_sep_cell_plist`, is to check that `c` refers to a list cell, `pl` respects the `plist` predicate, and the correspond-

ing pointer and the list cells are separated from the cell referred to by `c`. To do that in MemCAD, we introduce an ad-hoc structure `cell_plist` with both pointers (Line 45). The function initializes a local structure (Lines 49–50) and takes its address (Line 51) in order to express the required check (Line 52). This check relies on the predicate `cell_plist` (Lines v–z of Fig. 6.3) stating that the given pointer is non-null and refers to a structure with two pointers at offsets 0 and 4, denoted `$0` and `$1`, referring, resp., to a cell and to a double pointer to a well-formed list, which are separated (between them and from the list cells). Notice that "`...*$1.plist()`" on Line y specifies separation recursively, that is, from all locations considered in separation constraints reached via `plist` (and hence via `ll_cell`).

```
59 /*@
60   assigns \nothing;
61   ensures \result ≠  NULL ⇒ (\valid(\result) ∧
62     \result->next == NULL ∧ \result->data == 0);
63 */
64 cell* calloc_cell() {
65   cell* c = malloc(sizeof(cell));
66   if(c){c->next = NULL; c->data = 0;}
67   return c;
68 }
```

Figure 6.5 – *Function `calloc_cell` contract.*

An important benefit of using MemCAD is its capacity to automatically handle dynamic memory allocation, which is not yet supported in Wp. Thus, we define a custom allocator that simulates the behavior of `calloc` for list cells on Lines 59–68 of Fig. 6.5. Wp uses its contract, which is simple but currently unprovable by Wp since dynamic allocation is not supported (it should become provable when this support is added into Wp).

6.1.4 Proof of Function `list_push`

We illustrate our approach on function `list_push` of Fig. 6.2. It tries to allocate a new cell (Lines 89–90), and, in case of success, puts it on top of the list with the given data (Lines 95, 97, 99). Lines 94, 98 define *ghost* labels (that is, labels used only in annotations).

Lines 91–93 show how we use MemCAD to verify that the new cell (referred to by `c`) is separated both from the list cells and the pointer referred to by `pl` (Line 91), and introduce these properties as assumptions for Wp (**`admit`** clauses on Lines 92–93). They help Wp to prove in an `assert` clause on Line 96 that the list remains unchanged since label `Alloc` (i.e. Line 94) despite writing into the new cell on Line 95, and a similar assertion for the old list on Lines 100–101 despite the assignment on Line 99.

Instead of reasoning about the modified list directly in Wp—which often presents another difficulty for deductive verification—we let MemCAD check the list invariants on Line 102 and admit them on Lines 103–104 for Wp to prove the postconditions. Thanks to those assumptions, Wp successfully proves this function. Notice that the check instruction for MemCAD and the admit instructions for Wp are placed (for the moment, manually) at the same program location to ensure the soundness of the global verification.

```
108 int mc_verify_list_push(void) {
109   list* pl; int i; _memcad("add_inductive(pl,plist)");
110   list_push(pl, i);
111 }
```

Figure 6.6 – *Wrapper to verify* `list_push` *in* MemCAD.

In order to have a full proof, we also need to run MemCAD to verify all the checks in `list_push`. For this purpose, we define a wrapper in Fig. 6.6 to analyze the call to `list_push` on Line 110 with an arbitrary list respecting the given preconditions (which correspond in MemCAD, as we explained above, to assuming predicate `plist` for `pl`, cf. Lines 70–72, 109). MemCAD also succeeds in its analysis, hence, we can conclude that our function respects its ACSL contract.

While the annotation step is done manually in the current work, it can be automated in the future. A coordinated generation of checks and assumptions for a given recursive data structure for both tools will facilitate the verification and the justification of soundness of the combined approach. An early idea consists in defining a domain-specific language for the description of the target recursive data structure that is then used for the generation of necessary

predicates for MemCAD and for Wp as well as necessary assumptions and checks. The investigation of this research direction is left for future work.

6.1.5 Case study on tpm2-tss

We tested our approach on the toy example we used in Chapter 4, focused on the linked list used to store and use TPM resources in the TSS. In that toy example, list nodes were dynamically allocated using the `calloc` function. We simplified fields with complex data structures in nodes of the list, replacing the resources with a single `int` type variable.

We consider the two `createNode` and `getNode` functions, to add an object to the list, and to search for an object in a list. We apply MemCAD to avoid the usage of our custom static allocator, `calloc_NODE_T`, and to verify separation properties for a newly allocated node that Wp is currently not able to deduce with the original call to `calloc`. As demonstrated in Chapter 4, deductive verification with Wp of these functions required many additional lemmas and assertions, as well as the replacement of the dynamic memory allocation by a static allocator. Interestingly, the difficulty to verify real-life code was not caused by complex operations on lists—these operations are in reality quite simple in the target code—but by the difficulty to reason about the recursive data structure itself.

The proposed approach combining deductive verification with shape analysis allows us to perform a complete proof with less effort and without replacing the dynamic allocator by a static allocator. On the considered functions, the proof with Wp alone (cf. Section 4.5) required 14 lemmas, leading to the generation of 241 proof obligations, one of which required a manually created Wp script, and took 4m50s. Thanks to combining Wp and MemCAD in our work, we could remove 45 auxiliary ACSL annotations and 5 lemmas, so the proof required only 9 lemmas, leading to 194 proof obligations using no scripts, and took 1min47s in total for Wp and MemCAD (the latter taking less than 1 sec.).

6.2 Deductive Verification and Runtime Assertion Checking for Security Properties

With the work carried out in the previous section, we have presented an approach combining deductive verification using Frama-C/Wp and shape analysis using MemCAD. As MemCAD is more proficient at proving separation properties and structural invariants for linked data structures such as the list of TPM resources used in tpm2-tss, this approach allows us to assume such properties with Wp, which can use them to focus on other properties.

However, that approach has its own limitations, as the absence of a common higher-level specification mechanism between MemCAD and Frama-C makes it difficult to accurately translate properties from one to the other. Moreover, automating these translations is another issue which may prove to be problematic for MetAcsl-based approaches that rely heavily on the generation of low-level specifications for the verification of high-level properties. In this section, we take inspiration from the work of Section 6.1 to slightly improve upon the verification approach we proposed in Chapter 5, by introducing some deductive verification into it.

To illustrate our verification approach, we provide a description of how we filter properties for Wp and E-ACSL with Figures 6.7, 6.9, 6.10. We show how we define high-level properties for both Wp and E-ACSL with Figure 6.8. With Figure 6.11, we provide the Marshaling API macro defining the functions we use as target for the proof/Wp portion of the verification.

6.2.1 Filtering properties

Our verification approach, aiming to mix proof with runtime verification, relies on the idea we introduced in Section 6.1 to verify a property with one tool and assume it, if necessary, in the other tool. Because Wp and E-ACSL (as well as most Frama-C plugins in general) already use the ACSL specification language as a common specification mechanism, for our approach, all that is left is to ensure, firstly, what type of property can be or can not be verified with Wp (respectively E-ACSL), and secondly, given an annotated code, only

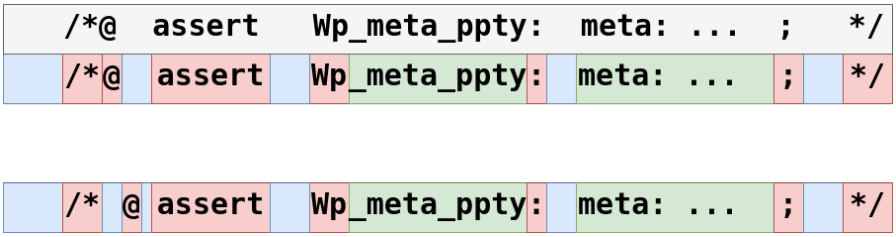
the properties written with Wp in mind are proved by Wp, and only those written with E-ACSL in mind are verified with E-ACSL.

We partially ensured the former thanks to our work in Chapter 4 and Chapter 5. As for the latter, we want to introduce some form of property filtering similarly to Section 6.1, so that ACSL properties defined for Wp are only visible to Wp but not E-ACSL, and vice-versa.

However, as of version 29.0, Frama-C does not provide any easy way to do that, and the intermediary AST and representations the Frama-C kernel generates when parsing C code are very difficult to access and to use. Instead, we choose to filter properties before any kind of processing performed by Frama-C.

```
1  (?P<spaces_1> *)
2  (?P<acsl_cmnt_start>\/\**)(?P<at>@)
3  (?P<spaces_2>\s*(\s|(//(?:(?!\*\/).*(\n)))\s*)*)
4  (?P<clause>assert|check|admit)
5  (?P<spaces_3>\s*(\s|(//(?:(?!\*\/).*(\n)))\s*)*)
6  (?P<tag_start>Wp)(?P<tag_end>(?:[^/@\s\-\:]*))(?P<tag_colon>\:)
7  (?P<spaces_4>\s*(\s|(//(?:(?!\*\/).*(\n)))\s*)*)
8  (?P<property>(?:[^/@][^@]+))
9  (?P<semicolon>;)
10 (?P<spaces_5>\s*(\s|(//(?:(?!\*\/).*(\n)))\s*)*)
11 (?P<acsl_cmnt_end>(\*+\/))
```

(a) *Example of regex with capturing groups used to filter out properties tagged for* Wp



(b) *Visual representation of the regular expression and its use to filter out properties.*

Figure 6.7 – *Example of regular expression used to filter out properties tagged for* Wp.

In order to do so, we rely on regular expressions, as defined in Figure 6.7. We assume we have already previously defined the `Wp_meta_ppty` meta-property, which has been used to generate low-level assertions. In Figure 6.7a,

we define a regular expression using named capturing groups, which we represent visually using such an assertion as an example in Figure 6.7b. The regular expression defined here is used to identify multi-line C comments corresponding to ACSL assertions tagged with a property name starting with `Wp`.

More specifically, the **`spaces_1`** group line `1` is used to detect any number of spaces in a line before the start of a comment, and the **`spaces_n`** capturing groups, for `n` between `2` and `5`, defined on lines 3, 5, 7 and 10, match any number of characters that do not cause the start or the end of an ACSL comment, and that do not contain any colon (used for property tags). In the visual example of Figure 6.7b, they correspond, before filtering, to the blue colored blocks.

The **`acsl_cmnt_start`** and **`at`** groups line 2 match with the `/*` and the `@` corresponding to the start of an ACSL multi-line comment, and the **`semicolon`** (line 9) and the **`acsl_cmnt_end`** (line 11) capturing groups are used to match with the semicolon and the end of a multi-line comment, modulo a certain number of spaces. In the visual example of Figure 6.7b, they correspond respectively to the first, second, second to last and last red colored blocks.

The **`clause`** capturing group (line 4) makes it so that the regular expression only matches the multi-line ACSL comments that use an **`assert`**, **`check`** or an **`admit`** (although in the work presented in this section, we only focus on filtering low-level assertions). In the visual example of Figure 6.7b, this corresponds to the red block with the `assert` clause.

The **`tag_start`**, **`tag_end`**, and **`tag_colon`** groups (line 6) are used to match with tag names that start with `Wp`. In the visual example of Figure 6.7b, this corresponds to the red and green blocks with the `Wp_meta_ppty` tag name, followed by the red semicolon block.

Finally, the **`property`** capturing group (line 8) matches with anything between the tag name of the property, to the semicolon marking the end of the ACSL property. We do not check if it defines a valid ACSL property, as the kernel of Frama-C already performs such checks when parsing annotated C codes. In the visual example of Figure 6.7b, this corresponds to the green block with the secondary `meta` tag name.

Thus, for any ACSL multi-line comment that matches this regular expression, all we need to do for every assertion we want to be verified in E-ACSL and to be ignored by Wp, is to introduce any number of spaces before the `@` symbol of the **`at`** capturing group for E-ACSL.

Similarly, for assertions we want to prove with Wp and admit for E-ACSL, all we need to replace the `assert` clause with an `admit` clause, if the **`clause`** group matches to an `assert` clause. On a code of 100k lines of code, with approximately 1400 of such assertions, this filtering only takes about 2 seconds. Similarly, using `eacsl` instead of `Wp` as a tag, we filter properties written with E-ACSL in mind.

Defining properties

As we now have a good idea of how to filter properties, the next step is to define and tag target high-level properties with regards to how their translation into low-level assertions with MetAcsl can be handled by Wp and E-ACSL.

For our approach, we focus here on the integrity of sensitive data as defined in Section 5.4, that is to say, as the separation between sensitive data and any written memory when relevant. We target the same subset of functions, but we focus here on proving integrity of sensitive data in functions of the Marshaling API.

We focus here on marshals for three main reasons:

- Marshals being used to translate and copy data between higher-level layers like ESAPI and lower-level layers like SAPI, it is important to be able to ensure integrity (and confidentiality) in these functions.

- Marshaling and unmarshaling functions in the tpm2-tss library are defined using macros for each category of data types. For example, there is one macro to define all marshaling functions (resp. one macro for all unmarshaling functions) for all 15 base types used in the library, and two marshaling macros for functions for all 30 `TPM2B` types.

  Such consistency means that there are not any variations of names for

arguments of functions of the same category, allowing for the use of the `\formal` keyword in meta-properties.

- Being able to access function parameters in meta-properties is extremely useful, as proving properties with Wp can require many additional annotations (as shown in Chapter 4), making MetAcsl more convenient to use, since we now have the ability to automatically generate (using additional meta-properties) intermediary assertions, and even preconditions and postconditions, using function parameters to help proving other properties (such as the low-level assertions relating to security properties.)

We demonstrate our approach on the category of marshals of tpm2-tss for base types, focusing on the 6 functions used in the subset of code of Chapter 5. We aim to prove the integrity of sensitive data in these functions, instead of "simply" verifying it at runtime. In order to filter out the low-level assertions generated for the integrity, for these 6 marshals, for E-ACSL, and those generated for the other functions, for Wp, we need to split the `all_sens_data_integrity` property previously defined in Figure 5.7 into two variants, presented in Figure 6.8.

We define in lines 1-9 of Figure 6.8 the integrity for functions verified with E-ACSL, and in lines 11-23 the one for the 6 marshal functions we target with Wp. The E-ACSL variant is identical to the `all_sens_data_integrity` property, except the `\targets` clause which we removes the 6 functions defined as target for the Wp variant in lines 13-15. Besides the names, which differ so that we can filter the correspondingly generated low-level asserts, the only difference between the two properties is the additional `\valid(\written)` clause line 20 for the Wp variant. This is not required for `eacsl_sens_integrity`, as the `\separated` clause E-ACSL already ensures validity checks at runtime for its arguments. For `Wp_sens_integrity`, we only check the validity of the written memory, as the validity of sensitive data is already ensured by the strong validity invariant for sensitive data of Figure 5.2.

```
/*@ meta \prop, \name(eacsl_sens_integrity),
  \targets(...),
  \context(\writing),
  ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 ≤ _nb_sens ≤ MAX_SENS ⇒
    0 < _len_sens[i] ⇒
    _write_sens[i] ≠ 1 ⇒
    \separated(\written,
         (char *)_all_sens[i] + (0 .. (size_t) (_len_sens[i] - 1)));
*/

/*@ meta \prop, \name(Wp_sens_integrity),
  \targets(
    \union({Tss2_MU_UINT8_Marshal, Tss2_MU_UINT16_Marshal,
            Tss2_MU_UINT32_Marshal,Tss2_MU_TPM2_ST_Marshal,
            Tss2_MU_TPMI_ALG_HASH_Marshal,Tss2_MU_BYTE_Marshal})),
  \context(\writing),
  ∀ int i; 0 ≤ i < _nb_sens ⇒ 0 ≤ _nb_sens ≤ MAX_SENS ⇒
    0 < _len_sens[i] ⇒
    _write_sens[i] ≠ 1 ⇒
    \valid(\written) ⇒ //equivalent RTE safeguard
    \separated(\written,
         (char *)_all_sens[i] + (0 .. (size_t) (_len_sens[i] - 1)));
*/
```

Figure 6.8 – *Meta-properties for the integrity of sensitive data for* Wp *and* E-ACSL

Parsing for verification

Similarly to defining security properties, parsing them for our approach requires slight adjustments with regards to the work we carried out in Chapter 5. Ideally, all we would need to do would be to simply parse the code and meta-properties with MetAcsl to generate the code with low-level annotations, filter properties with our regexes to generate two versions of the code, one for deductive verification, and one for runtime verification, as show in Figure 6.9.

However, we know that verification with E-ACSL can require functions of the C standard library not defined in Frama-C's own version of the libc (in fact, we know from Section 5.4 that it is the case for the tpm2-tss library and for our function subset), forcing the usage of the libc of the system. Similarly, proving with Wp requires the use of Frama-C's version of the standard library, as Wp may rely on its base predefined annotations, and as there can be mismatches

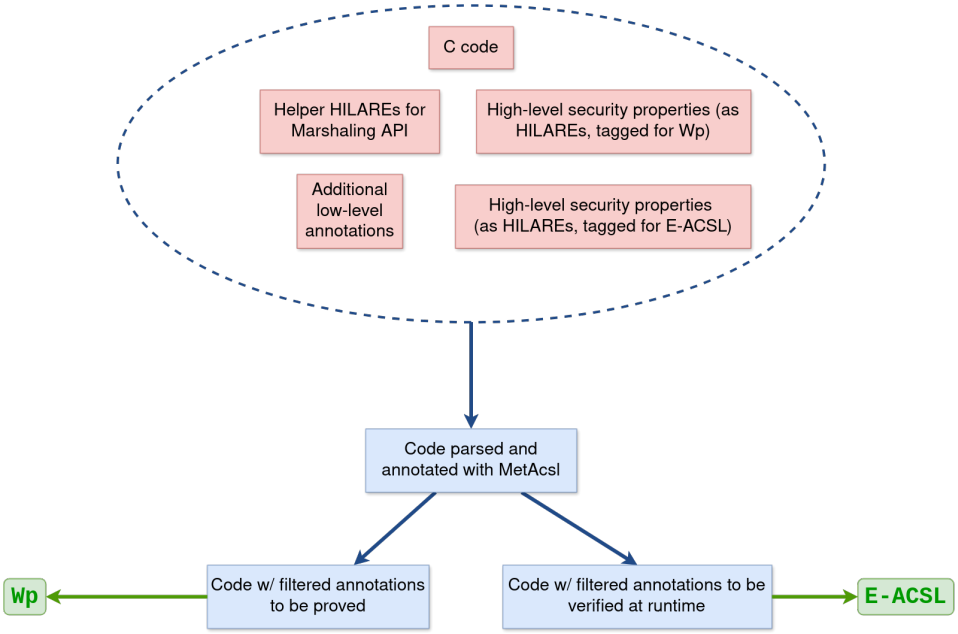


Figure 6.9 – *Expected verification approach*

between low-level type and function definitions between both versions of the libc.

Therefore, instead of directly sending the output given by MetAcsl to both verification plugins, we choose to parse the code with MetAcsl twice, pre property filtering: once using the Frama-C libc to output a version with Wp in mind, and once again for E-ACSL. Besides the standard library, as shown in Figure 6.10, both versions are generated using the same properties over sensitive data, the same intermediary assertions, and the same helper meta-properties that we will talk about in the next subsection.

6.2.2 Proof of marshal functions

We illustrate our approach by focusing on the 6 marshals for base types that we target with Wp. All 6 functions are defined with the `BASE_MARSHAL` macro, a simplified version of which is defined in Figure 6.11, which takes as argument the type **`type`** of the object to be marshaled, and defines the corresponding `Tss2_MU_##type##_Marshal` function.

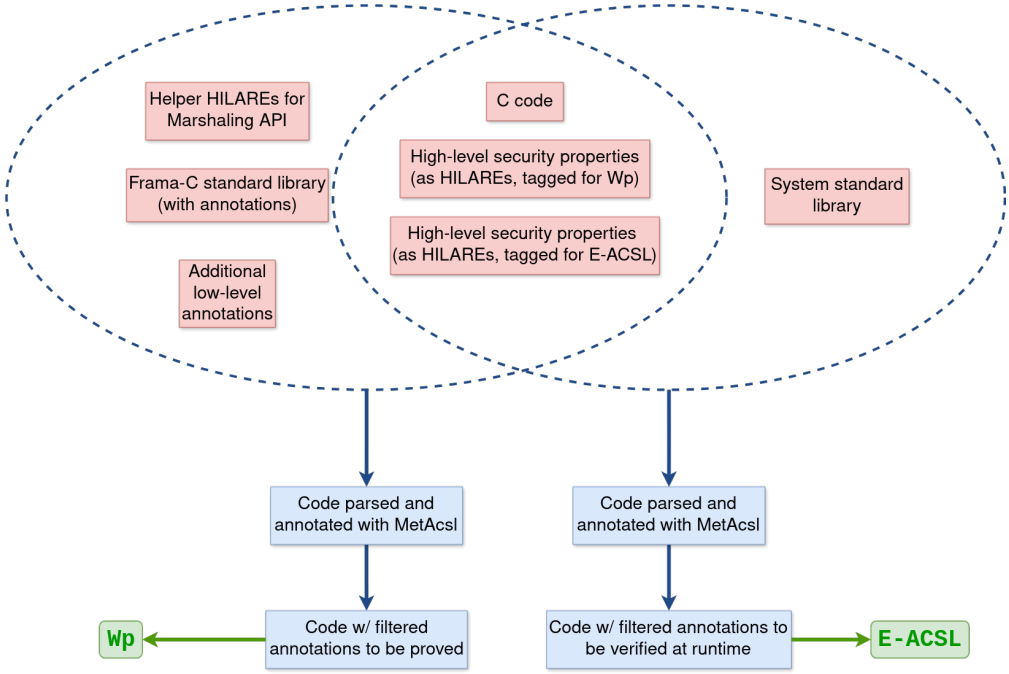


Figure 6.10 – *Verification approach with limitations due to incompatibilities of the libc in mind.*

The function copies argument `src` byte per byte, into the `buffer` array of bytes of size `buffer_size`, with a starting position for writing into `buffer` given by the `offset` pointer argument.

Lines 9-21 perform various checks on parameter validity and values, to ensure marshaling is possible. Lines 23-28 change the endianness of `src` to big endian, depending on its type, and line 30 ensures the copy of the data given by `src`, byte per byte, into the `buffer` array of bytes. Line 31 updates, when defined, the value of `*offset`, before the function returns a success code with line 32.

While verifying integrity at runtime was relatively light on such functions specification-wise, proving it is not as straightforward, as deductive verification with Wp often requires intermediate assertions. Moreover, due to the modular aspect of Wp, the verification method also requires us to provide at the very least minimal function contracts with preconditions, which E-ACSL did not need.

```
#define BASE_MARSHAL(type) \
TSS2_RC \
Tss2_MU_##type##_Marshal ( \
    type            src, \
    uint8_t         buffer [], \
    size_t          buffer_size, \
    size_t         *offset) \
{ \
    size_t  local_offset = 0; \
\
    if (offset ≠ NULL) {local_offset = *offset;} \
\
    if (buffer == NULL ∧ offset == NULL) { \
        return TSS2_MU_RC_BAD_REFERENCE; \
    } else if (buffer == NULL ∧ offset ≠ NULL) { \
        *offset += sizeof (src); \
        return TSS2_RC_SUCCESS; \
    } else if (buffer_size < local_offset ∨ \
               buffer_size - local_offset < sizeof (src)) { \
        return TSS2_MU_RC_INSUFFICIENT_BUFFER; \
    } \
\
    switch (sizeof (type)) { \
        case 1: break; \
        case 2: src = HOST_TO_BE_16(src); break; \
        case 4: src = HOST_TO_BE_32(src); break; \
        case 8: src = HOST_TO_BE_64(src); break; \
    } \
\
    memcpy(&buffer [local_offset], &src, sizeof (src));\
    if (offset ≠ NULL) { *offset = local_offset + sizeof (src);} \
    return TSS2_RC_SUCCESS; \
}
```

Figure 6.11 – *Simplified Marshal macro for base types.*

Fortunately, and as we have previously mentioned, the macro used here is convenient for us to automate the generation of such specifications with MetAcsl: each function defined with this macro uses the same variable names for its parameters (all base type marshals have `src`, `buffer`, `buffer_size`, `offset` as arguments), allowing us to use the **`\formal`** keyword to easily access them in meta-properties. We can also use the **`\precond`** context to automatically generate preconditions, and the **`\strong_invariant`** to generate

useful intermediary assertions to help guide the proof. Figure 6.12 provides 3 examples of such properties:

- lines 1-5 define `Wp_valid_offset_MU_precond`, used to generate preconditions ensuring that the `offset` pointer is either `NULL` at the entry point of all 6 functions.
- lines 7-13 define `Wp_valid_buffer_MU_precond`, used to generate preconditions ensuring that the `buffer` pointer is either `NULL`, or the memory of size `buffer_size` at address `buffer` is valid at the entry point of all 6 functions.
- lines 15-20 define `Wp_sep_buffer_offset_MU_strinv`, defining a strong invariant ensuring the separation between `offset` and the `buffer` array at any program point in all 6 functions.

```
1  /*@ meta \prop, \name("valid_offset_MU_precond"),
2    \targets(...),
3    \context(\precond),
4
5    \formal(offset) ≠  \null ⇒ \valid(\formal(offset)); */
6
7  /*@ meta \prop, \name("valid_buffer_MU_precond"),
8    \targets(...),
9    \context(\precond),
10
11   \formal(buffer) ≠  \null ⇒
12     \valid((\formal(buffer) +
13             (0 .. sizeof(\formal(buffer_size)) - 1)));*/
14
15 /*@ meta \prop, \name("Wp_sep_buffer_offset_MU_strinv"),
16   \targets(...),
17   \context(\strong_invariant),
18
19   \separated( \formal(offset),
20       \formal(buffer) + (0 .. (size_t) (\formal(buffer_size) - 1)));
          */
```

Figure 6.12 – *Examples of meta-properties used to guide the proof of integrity in marshals.*

Note that names for our meta-properties for preconditions do not start with `Wp` or `ecasl`. We do not want those to be filtered by our regular expressions, as

it allows us, for supported preconditions, to both verify them at runtime when marshals are called, and to use them to prove post conditions and assertions for those functions.

6.2.3 Verification Results

This proposed approach, combining deductive verification and runtime verification, allows us to verify properties with higher confidence in parts of the code where proof is accessible, while at the same time delegating properties relying on features with limited support for WP (such as complex pointer casts, complex memory separations, and dynamic allocations) to E-ACSL. While using runtime verification provides less confidence in verification results than tools based on more rigorous techniques such as MEMCAD with shape analysis, this approach provides a common specification mechanism in the form of METACSL and the ACSL language.

We tested our approach on the function subset used in the work of Chapter 5, focusing on the verification of the integrity of sensitive data, isolating 6 marshal functions with base types for the proof. Proof results were obtained using FRAMA-C 28.1 on a desktop computer running Ubuntu 20.04.4 LTS, with an Intel(R) Core(TM) i5-6600 CPU @ 3.30 GHz, featuring 4 cores and 4 threads, with 16GB RAM. We ran FRAMA-C/WP with options `-wp-par 3` and `-wp-timeout 35`. We used the Alt-Ergo v2.5.3, CVC4 v1.8 and Z3 v4.8.15 solvers, via WHY3 v1.7.0.

In addition to validity invariant over sensitive data and the version of the meta-property for integrity written with WP in mind, we defined 11 additional meta-properties to guide the proof of integrity for the 6 marshal functions, and help the potential proof in their callers: 4 properties with the **`\precond`** context, 3 with **`\postcond`**, and 4 strong invariants. For these 6 functions, a total of 364 low-level annotations were generated, 42 of which corresponding to the integrity, 24 for preconditions, 34 for postconditions, and the remaining 280 for intermediary assertions.

Parsing meta-properties with METACSL took less than a minute for WP, with the proof over all *6 functions* of the subset took about 8min16s.

Conclusion and Future Work 7

Contents

7.1 Summary

In this thesis, we have studied to what extent it is currently possible to formally (or semi-formally) verify real-life critical code not designed with verification in mind, nor developed with the assistance of formal verification to ensure its safety. We focused on three main types of properties: low-level safety properties and low-level functional properties (in Chapter 4 and Section 6.1), and high-level security properties (in Chapter 5 and Section 6.2). We have identified what is currently handleable for deductive verification and runtime assertion checking through the Frama-C verification platform, temporary workarounds and solutions to address the limitations we have encountered, as well as some necessary improvements to conduct exhaustive and complete verification efforts on such code in the future. Indeed, the library code is very different from more "traditional" industrial codes, very complex, and challenging for verification tools such as Frama-C.

In Chapter 4, we have presented our verification results for a subset of 10 functions of the ESAPI layer of TPM2-TSS that we verified with WP. The verified properties include both low-level functional properties and the absence of runtime errors (such as invalid pointers and buffer overflows, which often lead to security vulnerabilities). We have described current limitations and temporary solutions we used to address them. We have proved all necessary lemmas (extending those of a previous case study for linked lists [22]) in COQ. Finally, we identified desired tool improvements to achieve a full formal verification of the library: support of dynamic allocations and their related ACSL clauses, a memory model which can handle byte-level operations, and clearer separation of modification statuses of variables between different segments of the memory of the program.

In Chapter 5, we presented preliminary work on how to define and verify (at runtime) security properties over sensitive data in real-life code like the TPM2-TSS library. As the communication layer between the TPM and host platforms or applications, ensuring the security of sensitive data to be stored on the TPM is highly critical. It is highly desired to guarantee that such data can never be recovered without authorization while it is passing through the software stack. We have presented our verification result for a subset of 86 functions, on which we have verified, at runtime, that the target sensitive data (an object to be imported onto the TPM) is never modified or read when it is not supposed to. We have described some limitations of the tool and temporary solutions we used to address them. The subset was slightly simplified to remove dependencies to external libraries, but the logical behavior of the code corresponding to our subset of functions was maintained.

In Chapter 6, we proposed two preliminary approaches based on combining validation techniques to help alleviate some of the limitations highlighted by our work in the previous two chapters. We focused here on two combinations: in Section 6.1, the combination of FRAMA-C/WP with MEMCAD [104], a shape analysis tool, for the verification of low-level safety properties and low-level functional properties (extending the work presented in Chapter 4); in Section 6.2, the combination of WP with E-ACSL, for the verification of high-level security properties (extending the work presented in Chapter 5). In Section 6.1,

separation properties and structural invariants for linked data structures were more easily proved by MemCAD, and then used as assumptions in Wp, thus allowing it to focus on proving other properties. In Section 6.2, we adopted a similar approach, where we relied on Wp to prove security properties (as well as intermediary safety and functional properties) in functions where proofs are more easily automatable.

7.2 Future Work

This work opens up multiple opportunities for further extensions. In this section, we discuss some interesting research avenues that build upon the works presented in this thesis.

7.2.1 Deductive Verification of Safety and Functional Properties

The work presented in Chapter 4 opens the way towards a full verification of low-level properties on the tpm2-tss library (or similar security-critical library APIs [63, 114]).

Future work includes the verification of a larger subset of functions, including lower-level layers and operations. Specification and verification of specific security properties is another future work direction. Maintaining proofs between changing versions of tools is also an interesting research direction, in particular, by automating the generation of proof scripts (as recently proposed in [42]).

Finally, combining formally verified modules with modules which undergo a partial verification (e.g. limited to the absence of runtime errors, or runtime assertion checking of expected specifications on large test suites) can be another promising work direction to increase confidence in the security of the library.

7.2.2 Runtime Verification for Security

In the work presented in Chapter 5, we only verified, at runtime, two high-level security properties (integrity and confidentiality, defined as HILAREs) over (some, but not all) pieces of sensitive data, on a relatively simplified subset of the TPM2-TSS library.

In the future, an interesting research direction would be to perform a more complete verification work on the library, by reintroducing dependencies to external libraries and cryptographic capabilities of the TSS, and by running the verified code paired with a real TPM.

Extending the range of security properties and improving the automation of this approach is another point of improvement, as well as establishing a method to perform a more extensive evaluation of our verification methodology. It would be interesting and relevant for this evaluation to account for a careful comparison with deductive verification.

7.2.3 Combining Approaches

The work presented in Section 6.1 opens various research questions and perspectives. For instance, the automation of the proposed verification technique could be an interesting perspective. Such automation could include, for instance, a coordinated generation of checks of properties (in MemCAD) and assumptions (in Frama-C/Wp), as well as a proof of the soundness of this generation. Another interesting direction would be the design of a common (higher-level) specification mechanism, for recursive data structures such as the linked list structures used in TPM2-TSS, with automatic translation into suitable definitions for MemCAD and Frama-C. Here as well, establishing a more extensive evaluation of our verification methodology is another point of improvement, as is targeting other relevant case studies.

The work and approach we presented in Section 6.2 presents several points of amelioration and research perspectives as well. For instance, the filtering of properties we use currently only applies to low-level assertions, but extending to function contracts or even higher level properties would be an improve-

ment. Moreover, the current filtering relies on regular expressions, but using a sounder approach such as filtering properties directly at the level of the intermediary representations (such as the abstract syntax trees) Frama-C generates while parsing could be another interesting perspective.

Conversely, another research direction of interest could be to establish a methodology which focuses on applying more effective and sounder analyzers of the Frama-C platform (such as Wp or EVA) to verify such high-level requirements first, and to leave what could not be successfully verified to runtime verification. This approach would ideally be automatic and sequential: for instance, after defining high-level requirements, MetAcsl would be called to generate the corresponding low-level assertions; then the value analysis of the EVA plugin could be automatically applied to verify them, and would transfer the verification effort of properties that could not be verified to Wp; this plugin would in turns do the same and forward what deductive verification could not handle to the runtime verification provided by E-ACSL.

Another interesting research direction would be to establish a verification methodology loosely based on that of Chapter 5, which could consist on verifying high-level properties at runtime, and progressively introducing deductive verification into the verification effort.

7.2.4 Applying Artificial Intelligence to Support Verification

Recent advances in Artificial Intelligence (AI) and in particular in Large Language Models (LLMs) open the door to AI-assisted verification approaches. First efforts to evaluate the benefits of LLM-assisted techniques [14, 64, 72] demonstrate their potential and the need to further investigate and define an optimal usage of AI-assisted approaches for software verification. In particular, the authors of [72] have carried out encouraging preliminary work on the automated generation of ACSL loop invariants using LLMs.

This could also open up the possibility for automatically generating structural invariants of well-formed recursive data structures (such as the ones we manually define and use for linked lists in Chapters 4 and 6), for helping the

automation proposed as future work of Chapter 6, or to help automate the handling of sensitive data by the companion model given in Chapter 5.

Bibliography

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*, volume 8471 of *LNCS*, pages 55–71. Springer, 2014. doi: 10.1007/978-3-319-12154-3_4. 38

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification – The KeY Book: From Theory to Practice*. Springer International Publishing, Cham, 2016. doi: 10.1007/978-3-319-49812-6. 39

[3] Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. StaRVOOrS - episode II - strengthen and distribute the force. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, volume 9952 of *LNCS*, pages 402–415, 2016. doi: 10.1007/978-3-319-47166-2_28. 42

[4] ARM Limited. Building a Secure System using TrustZone Technology. `https://developer.arm.com/documentation/PRD29-GENC-009492/` (white paper), 2009. 37

[5] Will Arthur and David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, USA, 1st edition, 2015. ISBN 1430265833. 9

[6] Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. A model-based approach to combining static and dynamic verification techniques. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, volume 9952 of *LNCS*, pages 416–430, 2016. doi: 10.1007/978-3-319-47166-2_29. 42

[7] Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. Model-based static and runtime verification for Ethereum smart contracts. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *Model-Driven Engineering and Software Development - 8th International Conference, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020, Revised Selected Papers*, volume 1361 of *Communications in Computer and Information Science*, pages 323–348. Springer, 2020. doi: 10.1007/978-3-030-67445-8_14. 42

[8] Shaun Azzopardi, Joshua Ellul, Ryan Falzon, and Gordon J. Pace. Tainting in smart contracts: Combining static and runtime verification. In Thao Dang and Volker Stolz, editors, *Runtime Verification - 22nd International Conference, RV 2022, Tbilisi, Georgia, September 28-30, 2022, Proceedings*, volume 13498 of *LNCS*, pages 143–161. Springer, 2022. doi: 10.1007/978-3-031-17196-3_8. 42

[9] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011. doi: 10.1007/978-3-642-22110-1_14. iii, 2, 26

[10] David A. Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Symbolically analyzing security protocols using Tamarin. *ACM SIGLOG News*, 4(4): 19–30, 2017. doi: 10.1145/3157831.3157835. 35

[11] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché,

Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. URL `http://frama-c.com/acsl.html`. 17

[12] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. *WP Plugin Manual v1.0*, 2016. 26

[13] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM*, 64(8):56–68, 2021. ISSN 0001-0782. doi: 10.1145/3470569. vi, 4, 26

[14] Bernhard Beckert, Jonas Klamroth, Wolfram Pfeifer, Patrick Röper, and Samuel Teuber. Towards combining the cognitive abilities of large language models with the rigor of deductive progam verification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering Methodologies - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part IV*, volume 15222 of *LNCS*, pages 242–257. Springer, 2024. doi: 10.1007/978-3-031-75387-9_15. xv, 129

[15] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Annual international cryptology conference*, pages 232–249. Springer, 1993. 34

[16] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 230–239. IEEE Computer Society, 2004. doi: 10.1109/SEFM.2004.36. iii, 2, 37

[17] Téo Bernier, Yani Ziani, Nikolai Kosmatov, and Frédéric Loulergue. Combining deductive verification with shape analysis. In Dirk Beyer and Ana Cavalcanti, editors, *Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City,*

Luxembourg, April 6-11, 2024, Proceedings, volume 14573 of *LNCS*, pages 280–289. Springer, 2024. doi: 10.1007/978-3-031-57259-3_14. viii, 7

[18] Téo Bernier, Yani Ziani, Nikolai Kosmatov, and Frédéric Loulergue. Combining deductive verification with shape analysis: Companion artifact for the paper published at FASE 2024. 6

[19] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004. doi: 10.1007/978-3-662-07964-5. 26

[20] Dirk Beyer, Martin Spiessl, and Sven Umbricht. Cooperation between automatic and interactive software verifiers. In *20th Int. Conf. on Software Engineering and Formal Methods (SEFM 2022)*, volume 13550 of *LNCS*, pages 111–128. Springer, 2022. 43

[21] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. In *Proc. of the 10th NASA Formal Methods Symposium (NFM 2018)*, volume 10811 of *LNCS*, pages 37–53. Springer, 2018. 39

[22] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Logic against ghosts: Comparison of two proof approaches for a list module. In *Proc. of the 34th Annual ACM/SIGAPP Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT 2019)*, pages 2186–2195. ACM, 2019. doi: 10.1145/3297280.3297495. xi, xxii, 39, 49, 66, 67, 126

[23] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov. Towards Full Proof Automation in Frama-C using Auto-Active Verification. In *Proc. of the 11th NASA Formal Methods Symposium (NFM 2019)*, volume 11460 of *LNCS*, pages 88–105. Springer, 2019. doi: 10.1007/978-3-030-20652-9_6. 40

[24] Bruno Blanchet. Cryptoverif: A computationally-sound security protocol verifier. *Techical Report*, 2017. 34

[25] Stefan Blom and Marieke Huisman. The VerCors tool for verification of concurrent programs. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun,

editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014. doi: 10.1007/978-3-319-06410-9_9. 38

[26] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *Int. J. Softw. Tools Technol. Transf.*, 17(6):709–727, 2015. doi: 10.1007/S10009-014-0314-5. 26, 39

[27] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against AMD's Secure Encrypted Virtualization. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2875–2889. ACM, 2021. doi: 10.1145/3460120.3484779. 35

[28] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005. 17

[29] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *International conference on the theory and applications of cryptographic techniques*, pages 453–474. Springer, 2001. 34

[30] Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. Deductive verification of smart contracts with Dafny. In Jan Friso Groote and Marieke Huisman, editors, *Formal Methods for Industrial Critical Systems - 27th International Conference, FMICS 2022, Warsaw, Poland, September 14-15, 2022, Proceedings*, volume 13487 of *LNCS*, pages 50–66. Springer, 2022. doi: 10.1007/978-3-031-15008-1_5. 39

[31] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *Proc. of the 5th International Conference on Tests*

and Proofs (TAP 2011), volume 6706 of *LNCS*, pages 78–83. Springer, June 2011. ISBN 978-3-642-21767-8. doi: 10.1007/978-3-642-21768-5_7. 31

[32] Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. StaRVOOrS: A tool for combined static and runtime verification of Java. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *LNCS*, pages 297–305. Springer, 2015. doi: 10.1007/978-3-319-23820-3_21. 42

[33] Jesús Mauricio Chimento, Wolfgang Ahrendt, and Gerardo Schneider. Testing meets static and runtime verification. In Stefania Gnesi, Nico Plat, Paola Spoletini, and Patrizio Pelliccione, editors, *Proceedings of the 6th Conference on Formal Methods in Software Engineering, FormaliSE 2018, collocated with ICSE 2018, Gothenburg, Sweden, June 2, 2018*, pages 30–39. ACM, 2018. doi: 10.1145/3193992.3194000. 42

[34] Nathan Chong and Bart Jacobs. Formally verifying FreeRTOS' interprocess communication mechanism. 2021. URL `https://nchong.github.io/papers/ewc21.pdf`. 38

[35] Ştefan Ciobâcă and Diana-Elena Gratie. Implementing, specifying, and verifying the QOI format in Dafny: A case study. In Nikolai Kosmatov and Laura Kovács, editors, *Integrated Formal Methods - 19th International Conference, IFM 2024, Manchester, UK, November 13-15, 2024, Proceedings*, volume 15234 of *LNCS*, pages 35–52. Springer, 2024. doi: 10.1007/978-3-031-76554-4_3. 39

[36] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. 4

[37] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007. ISBN 978-3-540-71940-3. doi: 10.1007/978-3-540-71999-1. 37

[38] Guillaume Cluzel, Kyriakos Georgiou, Yannick Moy, and Clément Zeller. Layered formal verification of a TCP stack. In *Proc. of the IEEE Secure Development Conference (SecDev 2021)*, pages 86–93. IEEE, 2021. doi: 10.1109/SecDev51306.2021.00028. 42

[39] Christian Colombo and Gordon J. Pace. Runtime verification using LARVA. 3:55–63, 2017. doi: 10.29007/N7TD. 41, 42

[40] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In Dang Van Hung and Padmanabhan Krishnan, editors, *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 33–37. IEEE Computer Society, 2009. doi: 10.1109/SEFM.2009.13. 41, 42

[41] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018. URL `https://inria.hal.science/hal-01960203`. iii, 2, 26

[42] Loïc Correnson, Allan Blanchard, Adel Djoudi, and Nikolai Kosmatov. Automate where automation fails: Proof strategies for Frama-C/WP. In *Proc. of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2024)*, volume 14570 of *LNCS*, pages 331–339. Springer, April 2024. doi: 10.1007/978-3-031-57246-3_18. xiii, 127

[43] 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*, pages 238–252, 1977. iv, 3

[44] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a

simple ARM-based separation kernel. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM SIGSAC Conference on Computer and Communications Security, CCS, Berlin, Germany, November 4-8*, pages 223–234. ACM, 2013. doi: 10.1145/2508859.2516702. 37

[45] Ádám Darvas and Peter Müller. Proving consistency and completeness of model classes using theory interpretation. In *Proc. of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010)*, volume 6013 of *LNCS*, pages 218–232. Springer, 2010. 41

[46] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's Java.utils.Collection.sort() is broken: The good, the bad and the worst case. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *LNCS*, pages 273–289. Springer, 2015. doi: 10.1007/978-3-319-21690-4_16. 38

[47] Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying OpenJDK's sort method for generic collections. *J. Autom. Reason.*, 62(1):93–126, 2019. doi: 10.1007/S10817-017-9426-4. 38

[48] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *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*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. 26

[49] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006. 108

[50] Adel Djoudi, Martin Hána, and Nikolai Kosmatov. Formal Verification of a JavaCard Virtual Machine with Frama-C. In *Proc. of the 24th International Symposium on Formal Methods (FM 2021)*, volume 13047 of *LNCS*, pages 427–444, Cham, 2021. Springer. ISBN 978-3-030-90870-6. doi: 10.1007/978-3-030-90870-6_23. 41

[51] Adel Djoudi, Martin Hána, Nikolai Kosmatov, Milan Kříženecký, Franck Ohayon, Patricia Mouy, Arnaud Fontaine, and David Féliot. A bottom-up formal verification approach for Common Criteria certification: Application to JavaCard virtual machine. 2022. URL `https://hal.science/hal-03704287`. 39

[52] Frank Dordowsky. An experimental study using ACSL and Frama-C to formulate and verify low-level requirements from a DO-178C compliant avionics project. *Electronic Proceedings in Theoretical Computer Science*, 187: 28–41, 2015. ISSN 2075-2180. doi: 10.4204/EPTCS.187.3. 38

[53] Claire Dross and Johannes Kanig. Recursive data structures in SPARK. In Shuvendu K. Lahiri and Chao Wang, editors, *Proc. of the 32nd International Conference on Computer Aided Verification (CAV 2020)*, volume 12225 of *LNCS*, pages 178–189. Springer, 2020. doi: 10.1007/978-3-030-53291-8_11. 40

[54] Claire Dross and Yannick Moy. Auto-active proof of red-black trees in SPARK. In Clark W. Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, volume 10227 of *LNCS*, pages 68–83, 2017. doi: 10.1007/978-3-319-57288-8_5. 40

[55] Catherine Dubois and Alain Giorgetti. Tests and proofs for custom data generators. *Formal Aspects Comput.*, 30(6):659–684, 2018. doi: 10.1007/S00165-018-0459-1. 31

[56] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *Proc. of the 29th Annual IEEE Conference on Local Computer Networks*

(LCN 2004), pages 455–462. IEEE Computer Society, 2004. doi: 10.1109/LCN.2004.38. 40

[57] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. *Engineering dependable software systems*, pages 141–175, 2013. iv, 3

[58] Kevin Falzon and Gordon J. Pace. Combining testing and runtime verification techniques. In Ricardo Jorge Machado, Rita Suzana Pitangueira Maciel, Julia Rubin, and Goetz Botterweck, editors, *Model-Based Methodologies for Pervasive and Embedded Software, 8th International Workshop, MOMPES 2012, Essen, Germany, September 4, 2012. Revised Papers*, volume 7706 of *LNCS*, pages 38–57. Springer, 2012. doi: 10.1007/978-3-642-38209-3_3. 41

[59] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *Proc. of the 22nd European Symposium on Programming (ESOP 2013)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013. 26, 39

[60] International Organization for Standardization (ISO). *The ANSI C standard (C99)*, 2005. URL `https://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf`. 19

[61] Carlo A. Furia, Martin Nordio, Nadia Polikarpova, and Julian Tschannen. AutoProof: auto-active functional verification of object-oriented programs. *STTT*, 19(6):697–716, 2017. doi: 10.1007/s10009-016-0419-0. 41

[62] Christoph Gladisch and Shmuel S. Tyszberowicz. Specifying linked data structures in JML for combining formal verification and testing. *Sci. Comput. Program.*, 107-108:19–40, 2015. doi: 10.1016/j.scico.2015.02.005. 41

[63] GlobalPlatform. GlobalPlatform Technology TEE Internal Core API Specification Version 1.3.1. `https://globalplatform.org/specs-library/tee-internal-core-api-specification/`, 2021. 37, 127

[64] George Granberry, Wolfgang Ahrendt, and Moa Johansson. Specify what? Enhancing neural specification synthesis by symbolic methods. In Nikolai Kosmatov and Laura Kovács, editors, *Integrated Formal Methods - 19th International Conference, IFM 2024, Manchester, UK, November 13-15, 2024, Proceedings*, volume 15234 of *LNCS*, pages 307–325. Springer, 2024. doi: 10.1007/978-3-031-76554-4_19. xv, 129

[65] Reiner Hähnle and Marieke Huisman. Deductive software verification: From pen-and-paper proofs to industrial tools. In *Computing and Software Science – State of the Art and Perspectives*, volume 10000 of *LNCS*, pages 345–373. Springer, 2019. doi: 10.1007/978-3-319-91908-9_18. 38

[66] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, 2012. doi: 10.1145/2187671.2187678. 16

[67] Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Stefanescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. A type system for extracting functional specifications from memory-safe imperative programs. *Proc. ACM Program. Lang.*, 5:1–29, 2021. 42

[68] Philippe Herrmann and Julien Signoles. *RTE: Runtime Error Annotation Generation*, 2024. URL `https://frama-c.com/download/frama-c-rte-manual.pdf`. 27

[69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969. iii, 2

[70] Hans Niklas Jacob, Christian Werling, Robert Buhren, and Jean-Pierre Seifert. faultpm: Exposing AMD fTPMs' deepest secrets. In *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*, pages 1128–1142. IEEE, 2023. doi: 10.1109/EUROSP57164.2023.00069. 35

[71] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In Kazunori Ueda, editor, *Programming Languages and*

Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings, volume 6461 of *LNCS*, pages 304–311. Springer, 2010. doi: 10.1007/978-3-642-17164-2_21. 38

[72] Christian Janßen, Cedric Richter, and Heike Wehrheim. Can chatgpt support software verification? In Dirk Beyer and Ana Cavalcanti, editors, *Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings*, volume 14573 of *LNCS*, pages 266–279. Springer, 2024. doi: 10.1007/978-3-031-57259-3_13. xv, 129

[73] Nikolai Kosmatov and Julien Signoles. A lesson on runtime assertion checking with Frama-C. In *Runtime Verification (RV)*, volume 8174 of *LNCS*, pages 386–399, Berlin, Heidelberg, 2013. Springer. doi: 10.1007/978-3-642-40787-1_29. iv, vi, 3, 5, 28

[74] Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles. *Guide to Software Verification with Frama-C*. CSFAL. Springer Cham, 2024. vi, 4, 17

[75] Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles, editors. *Guide to Software Verification with Frama-C. Core Components, Usages, and Applications*. Computer Science Foundations and Applied Logic Book Series. Springer, 2024. ISBN 978-3-031-55607-4. doi: 10.1007/978-3-031-55608-1. 31, 39

[76] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. *J. Comput. Secur.*, 24(5):583–616, 2016. doi: 10.3233/JCS-160556. 35

[77] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, volume 523 of *The Kluwer International Series in Engineering and Computer Science*, pages 175–188. Springer, 1999. doi: 10.1007/978-1-4615-5229-1_12. 17

[78] Dirk Leinenbach and Thomas Santen. Verifying the microsoft Hyper-V hypervisor with VCC. In Ana Cavalcanti and Dennis Dams, editors, *Proc. of the Second World Congres on Formal Methods (FM 2009)*, volume 5850 of *LNCS*, pages 806–809. Springer, 2009. doi: 10.1007/978-3-642-05089-3_51. 38

[79] K. Rustan M. Leino and Valentin Wüstholz. The Dafny integrated development environment. In *F-IDE 2014*, pages 3–15. 39

[80] Rustan K. Leino. *Program Proofs*. MIT Press, 2024. 39

[81] Frédéric Loulergue, Allan Blanchard, and Nikolai Kosmatov. Ghosts for lists: from axiomatic to executable specifications. In *Proc. of the 12th International Conference on Tests and Proofs (TAP 2018)*, volume 10889 of *LNCS*, pages 177–184. Springer, 2018. doi: 10.1007/978-3-319-92994-1_11. 96

[82] Yuwei Ma, Qianying Zhang, Shijun Zhao, Guohui Wang, Ximeng Li, and Zhiping Shi. Formal verification of memory isolation for the TrustZone-based TEE. In *27th Asia-Pacific Software Engineering Conference, APSEC 2020, Singapore, December 1-4, 2020*, pages 149–158. IEEE, 2020. doi: 10.1109/APSEC51365.2020.00023. 37

[83] Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. A memory allocation module of Contiki formally verified with Frama-C. A case study. In *Proc. of the 11th International Conference on Risks and Security of Internet and Systems (CRiSIS 2016)*, volume 10158 of *LNCS*, pages 114–120. Springer, 2016. 51

[84] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with Spark*. Cambridge University Press, 2015. 39, 40

[85] Bertrand Meyer. Eiffel: A language and environment for software engineering. *J. Syst. Softw.*, 8(3):199–246, 1988. doi: 10.1016/0164-1212(88)90022-2. 16

[86] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1988. 16

[87] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2057–2073. USENIX Association, 2020. URL `https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-tpm`. 35

[88] Mathieu Montin, Amélie Ledein, and Catherine Dubois. LibNDT: Towards a formal library on spreadable properties over linked nested datatypes. In Jeremy Gibbons and Max S. New, editors, *Proceedings Ninth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2022, Munich, Germany, 2nd April 2022*, volume 360 of *EPTCS*, pages 27–44, 2022. doi: 10.4204/EPTCS.360.2. 40

[89] Zeinab Nehaï and François Bobot. Deductive proof of industrial smart contracts using Why3. In Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Creissac Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas, editors, *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part I*, volume 12232 of *LNCS*, pages 299–311. Springer, 2019. doi: 10.1007/978-3-030-54994-7_22. 39

[90] Matús Nemec, Marek Sýs, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith's attack: Practical factorization of widely used RSA moduli. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1631–1648. ACM, 2017. doi: 10.1145/3133956.3133969. 35

[91] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9. 37

[92] Wytse Oortwijn and Marieke Huisman. Formal verification of an industrial safety-critical traffic tunnel control system. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, *Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings*, volume 11918 of *LNCS*, pages 418–436. Springer, 2019. doi: 10.1007/978-3-030-34968-4_23. 38

[93] Dillon Pariente and Julien Signoles. Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, June 2017. 30

[94] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. How testing helps to diagnose proof failures. *Formal Asp. Comput.*, 30(6):629–657, 2018. ISSN 0934-5043. doi: 10.1007/s00165-018-0456-4. 31

[95] Ruzica Piskac, Thomas Wies, and Damien Zufferey. GRASShopper - complete heap verification with mixed specifications. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, pages 124–139. Springer, 2014. 42

[96] Nadia Polikarpova, Carlo A. Furia, and Bertrand Meyer. Specifying reusable components. In *Proc. of the 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2010)*, volume 6217 of *LNCS*, pages 127–141. Springer, 2010. 40

[97] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. *Formal Asp. Comput.*, 30(5):495–523, 2018. doi: 10.1007/s00165-017-0435-1. 41

[98] Siddharth Prakash Rao, Gabriela Limonta, and Janne Lindqvist. Usability and security of Trusted Platform Module (TPM) library APIs. In Sonia Chiasson and Apu Kapadia, editors, *Eighteenth Symposium on Usable Privacy and Security, SOUPS 2022, Boston, MA, USA, August 7-9, 2022*, pages

213–232. USENIX Association, 2022. URL https://www.usenix.org/conference/soups2022/presentation/rao. 36

[99] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. MetAcsl: Specification and verification of high-level properties. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019)*, volume 11427 of *LNCS*, pages 358–364. Springer, 2019. doi: 10.1007/978-3-030-17462-0_22. vi, 5, 29

[100] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. Tame your annotations with MetAcsl: Specifying, testing and proving high-level properties. In *Proc. of the 13th International Conference on Tests and Proofs (TAP 2019), Held as Part of the Third World Congress on Formal Methods (FM 2019)*, volume 11823 of *LNCS*, pages 167–185. Springer, 2019. doi: 10.1007/978-3-030-31157-5_11. 41

[101] Jianxiong Shao, Yu Qin, and Dengguo Feng. Formal analysis of HMAC authorisation in the TPM2.0 specification. *IET Inf. Secur.*, 12(2):133–140, 2018. doi: 10.1049/iet-ifs.2016.0005. 35

[102] Carlton Shepherd and Konstantinos Markantonakis. *Trusted Execution Environments*. Springer, 2024. ISBN 978-3-031-55560-2. doi: 10.1007/978-3-031-55561-9. 35, 37

[103] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008. doi: 10.1007/978-3-540-71067-7_6. 37

[104] Pascal Sotin and Xavier Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *10th Asian Symposium on Programming Languages and Systems (APLAS'12)*, volume 7705 of *LNCS*, pages 131–147. Springer, 2012. vi, xii, 5, 8, 104, 108, 109, 126

[105] Petr Svenda, Antonin Dufka, Milan Broz, Roman Lacko, Tomas Jaros, Daniel Zatovic, and Josef Pospisil. TPMScan: A wide-scale study of security-relevant properties of TPM 2.0 chips. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):714–734, 2024. doi: 10.46586/TCHES.V2024.I2.714-734. 36

[106] Philip Tasche, Raúl E. Monti, Stefanie Eva Drerup, Pauline Blohm, Paula Herber, and Marieke Huisman. Deductive verification of parameterized embedded systems modeled in SystemC. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II*, volume 14500 of *LNCS*, pages 187–209. Springer, 2024. doi: 10.1007/978-3-031-50521-8_9. 38

[107] Andy S. Tatman, Hans-Dieter A. Hiep, and Stijn de Gouw. Analysis and formal specification of OpenJDK's BitSet. In Paula Herber and Anton Wijs, editors, *iFM 2023 - 18th International Conference, iFM 2023, Leiden, The Netherlands, November 13-15, 2023, Proceedings*, volume 14300 of *LNCS*, pages 134–152. Springer, 2023. doi: 10.1007/978-3-031-47705-8_8. 39

[108] The Coq Development Team. The Coq proof assistant. `http://coq.inria.fr`,. iii, 2

[109] Trusted Computing Group. Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.59 – November. `https://trustedcomputinggroup.org/work-groups/trusted-platform-module/`, 2019. last accessed: May 2023. 9

[110] Trusted Computing Group. TCG TSS 2.0 Overview and Common Structures Specification. `https://trustedcomputinggroup.org/work-groups/software-stack/`, 2021. 9

[111] Weijin Wang, Yu Qin, and Dengguo Feng. Automated proof for authorization protocols of TPM 2.0 in computational model. In *Information Security Practice and Experience: 10th International Conference, ISPEC 2014,*

Fuzhou, China, May 5-8, 2014. Proceedings 10, pages 144–158. Springer, 2014. 34

[112] Weijin Wang, Yu Qin, Bo Yang, Yingjun Zhang, and Dengguo Feng. Automated security proof of cryptographic support commands in TPM 2.0. In *Proc. of the 18th International Conference on Information and Communications Security (ICICS 2016)*, volume 9977 of *LNCS*, pages 431–441, Cham, 2016. Springer. doi: 10.1007/978-3-319-50011-9_33. 34

[113] Nicky Williams and Nikolai Kosmatov. *Chapter 6: Test Generation with PathCrawler*, pages 305–338. Computer Science Foundations and Applied Logic Book Series. Springer, 2024. ISBN 978-3-031-55607-4. doi: 10.1007/978-3-031-55608-1_6. 31

[114] Geunyeol Yu, Seunghyun Chae, Kyungmin Bae, and Sungkun Moon. Formal specification of Trusted Execution Environment APIs. In *Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings*, volume 14573 of *LNCS*, pages 101–121. Springer, 2024. doi: 10.1007/978-3-031-57259-3_5. 37, 127

[115] Qianying Zhang and Shijun Zhao. A comprehensive formal security analysis and revision of the two-phase key exchange primitive of TPM 2.0. *Comput. Networks*, 179, 2020. doi: 10.1016/j.comnet.2020.107369. 34

[116] Yani Ziani, Nikolai Kosmatov, Frédéric Loulergue, and Daniel Gracia Pérez. Runtime verification for high-level security properties: Case study on the tpm software stack: Companion artifact for the paper published at TAP 2024, . 5, 74

[117] Yani Ziani, Nikolai Kosmatov, Frédéric Loulergue, Daniel Gracia Pérez, and Téo Bernier. Towards formal verification of a tpm software stack: Companion artifact for the paper published at iFM 2023, . URL `https://doi.org/10.5281/zenodo.13693011`. 5, 46

[118] Yani Ziani, Nikolai Kosmatov, Frédéric Loulergue, Daniel Gracia Pérez, and Téo Bernier. Towards formal verification of a TPM software stack. In Paula Herber and Anton Wijs, editors, *iFM 2023 - 18th International Conference, iFM 2023, Leiden, The Netherlands, November 13-15, 2023, Proceedings*, volume 14300 of *LNCS*, pages 93–112. Springer, 2023. doi: 10.1007/978-3-031-47705-8_6. viii, 6, 46, 96

[119] Yani Ziani, Nikolai Kosmatov, Frédéric Loulergue, and Daniel Gracia Pérez. Runtime verification for high-level security properties: Case study on the TPM software stack. In Marieke Huisman and Falk Howar, editors, *Tests and Proofs - 18th International Conference, TAP 2024, Milan, Italy, September 9-10, 2024, Proceedings*, volume 15153 of *LNCS*, pages 87–106. Springer, 2024. doi: 10.1007/978-3-031-72044-4_5. viii, 6, 73

[120] Yani Ziani, Téo Bernier, Nikolai Kosmatov, Frédéric Loulergue, and Daniel Gracia Pérez. Towards formal verification of a TPM software stack: Achievements and opportunities. *Form. Asp. Comput.*, June 2025. ISSN 0934-5043. doi: 10.1145/3743153. Just Accepted. ix, 7, 46

Yani ZIANI

Vérification formelle de couches de confiance dans les logiciels : application à la TPM Software Stack

Résumé :
Le Trusted Platform Module (TPM) est un cryptoprocesseur conçu pour protéger l'intégrité et la sécurité des ordinateurs modernes. Les communications avec le TPM passent par la TPM Software Stack (TSS), qui joue le rôle de couche de confiance pour le TPM. La bibliothèque open-source tpm2-tss (écrite en C) est une implémentation largement utilisée de la TSS. Actuellement, la spécification et la vérification d'applications critiques pour la sécurité telles que la librairie tpm2-tss restent un défi important, en particulier lorsque le code n'est pas écrit dans l'optique d'être vérifié. L'objectif de cette thèse est d'explorer dans quelle mesure il est actuellement possible de spécifier et de vérifier des propriétés de sécurité, des propriétés fonctionnelles et des propriétés de sûreté sur un code critique qui n'a pas été pensé pour la vérification. La plateforme de vérification Frama-C et certains de ses principaux analyseurs (Wp, E-ACSL et MetAcsl) sont utilisés pour étudier l'efficacité de diverses techniques de vérification dans ce contexte.
En ce qui concerne la vérification déductive à l'aide de Wp, la spécification et la vérification de propriétés fonctionnelles et de l'absence d'erreurs à l'exécution sur un sous-ensemble représentatif de fonctions mettent en évidence plusieurs limitations et des solutions de contournement nécessaires, notamment lorsqu'il s'agit d'allocations dynamiques et de structures de données récursives. Par ailleurs, lorsque la vérification déductive s'avère difficile à appliquer à ce type de code, l'outil de vérification à l'exécution E-ACSL est utilisé pour vérifier les propriétés de sécurité de haut niveau écrites avec MetAcsl. Cette approche offre un moyen plus accessible d'évaluer des propriétés de haut niveau sur des programmes de grande taille, mais donne moins de confiance dans les résultats de la vérification. Enfin, nous explorons des approches combinées pour évaluer les propriétés de tels programmes : d'abord en combinant la vérification déductive avec l'analyse de forme fournie par l'outil MemCAD pour se concentrer sur les propriétés de bas niveau; ensuite en combinant Wp et E-ACSL pour la vérification d'exigences de haut niveau écrites avec MetAcsl.

Mots clés : methodes formelles, vérification déductive, vérification dynamique, Frama-C, TPM software stack

Formal Verification of Trusted Layers of Software: Application to the TPM Software Stack

Abstract :
The Trusted Platform Module (TPM) is a cryptoprocessor designed to protect integrity and security of modern computers. Communications with the TPM go through the TPM Software Stack (TSS), which assumes the role of a trusted layer for the TPM. The open-source library tpm2-tss (written in the C programming language) is a widely used implementation of the TSS. Currently, the specification and verification of security-critical applications such as tpm2-tss remains an important challenge, especially when the code is not written with an intention of verification in mind. The goal of this thesis is to explore to what extent it is currently possible to specify and verify safety, functional and security properties on critical code not designed for verification. The Frama-C verification platform and some of its main analyzers (Wp, E-ACSL and MetAcsl) are used to investigate the efficiency and effectiveness of various verification techniques in this context.
Regarding deductive verification using Wp, the specification and verification of functional properties and the absence of runtime errors on a representative subset of functions highlights several limitations and necessary workarounds, notably when dealing with dynamic allocations and recursive data structures. Alternatively, when deductive verification has proved to be a challenge for such code, the runtime verification tool E-ACSL is used to verify high-level security properties written with MetAcsl. This approach offers a more accessible way to assess high-level properties over large programs, but provides less confidence in verification results. Finally, we explore combined approaches to evaluate properties on such programs: first by combining deductive verification with the shape analysis provided by the MemCAD tool to focus on low-level properties; then by combining Wp and E-ACSL for the verification of high-level requirements written with MetAcsl.

Keywords : formal methods, deductive verification, runtime verification, Frama-C, TPM software stack

Laboratoire d'Informatique Fondamentale d'Orléans (LIFO) - UR 4022
Bâtiment IIIA - 6, rue Léonard de Vinci - BP 6749 - 45067 Orléans Cedex 2