Safe and secure model-driven design for embedded systems

Letitia Li

université PARIS-SACLAY

TELECOM ParisTech

NNT : 2018SACLT002

Thèse de doctorat

Approche Orientée Modèles pour la Sûreté et la Sécurité des Systèmes Embarqués

Thèse de doctorat de l'Université Paris-Saclay
préparée à Telecom ParisTech

Ecole doctorale n°580 Dénomination (STIC)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Biot, le 3 septèmbre 2018, par

LETITIA W. LI

Composition du Jury :

Prof. Philippe Collet Professeur, Université Côte d'Azur	Président
Prof. Guy Gogniat Professeur, Université de Bretagne Sud	Rapporteur
Prof. Maritta Heisel Professeur, University Duisburg-Essen	Rapporteur
Prof. Jean-Luc Danger Professeur, Telecom ParisTech	Examinateur
Dr. Patricia Guitton Ingénieur, Renault Software Labs	Examinateur
Dr. Ludovic Apvrille Maître de Conférences, Telecom ParisTech	Directeur de thèse
Dr. Annie Bracquemond Directeur de Recherche, Institut Vedecom	Co-directeur de thèse

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Approche Orientée Modèles pour la Sûreté et la Sécurité des Systèmes Embarqués

Mots clés: systèmes embarques, véhicules autonomes, sûreté de fonctionnement, sécurité, exploration d'architecture

Résumé: La présence de systèmes et d'objets embarqués communicants dans notre vie quotidienne a apporté une myriade d'avantages, allant de l'ajout de commodités et de divertissements à l'amélioration de la sûreté des déplacements et des soins de santé. Cependant, les défauts et les vulnérabilités de ces systèmes exposent leurs utilisateurs à des risques de dommages matériels, de pertes monétaires, et même de dommages corporels. Par exemple, certains véhicules commercialisés, qu'ils soient connectés ou conventionnels, ont déjà souffert d'une variété de défauts de conception entraînant des victimes. Dans le même temps, alors que les véhicules sont de plus en plus connectés (et dans un avenir proche, autonomes), les chercheurs ont démontré la possibilité de piratage de leurs capteurs ou de leurs systèmes de contrôle interne. Cette thèse s'intéresse à la sécurité et la sûreté des systèmes embarqués, dans le contexte du véhicule autonome de l'Institut Vedecom. Notre approche repose sur une nouvelle méthode de modélisation pour concevoir des systèmes embarqués sûrs et sécurisés, basée sur la méthodologie SysML-Sec, et impliquant de nouvelles stratégies de modélisation et de vérification formelle.

Title: Safe and Secure Model-Driven Design for Embedded Systems

Keywords: embedded systems, autonomous vehicles, safe comportment, security, design space exploration

Abstract: The presence of communicating embedded systems/IoTs in our daily lives have brought a myriad of benefits, from adding conveniences and entertainment, to improving the safety of our commutes and health care. However, the flaws and vulnerabilities in these devices expose their users to risks of property damage, monetary losses, and personal injury. For example, consumer vehicles, both connected and conventional, have succumbed to a variety of design flaws resulting in injuries, and in some cases, death. At the same time, as vehicles become increasingly connected (and in the near future, autonomous), researchers have demonstrated possible hacks on their sensors or internal control systems. This thesis discusses how to ensure the safety and security of embedded systems, in the context of Institut Vedecom's autonomous vehicle. Our approach involves a new model-based methodology for safe and secure design, which involve new modeling and verification methods.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Acknowledgments

The research presented in this thesis was sponsored by Institut Vedecom of Versailles, France and took place at Lab System-on-Chip of Telecom ParisTech in Sophia Antipolis, France.

With the conclusion of my PhD, I would like to recognize everyone who made the completion of this thesis possible: through technical support, encouragement, assisting with my post-thesis career plans, and helping me find a life in France.

I would like to thank my advisors, Professor Ludovic Apvrille and Dr. Annie Bracquemond, who have guided me through this thesis. They have taught me all about modeling, verification, safety, security, and embedded systems, and it is their suggestions and corrections who have shaped this thesis into a coherent scientific work. I have been exceptionally fortunate for their availability to aid me in more than just my research, for I could not have navigated the administrative procedures or language difficulties without them, and I am also grateful for their willingness to show me the cultural heritage of the country, such as raclette.

Next, I would like to thank all of my jury members for their time in participating in my defense, and all of the insights and the expertise they brought forth. I thank my reviewers: Professor Guy Gogniat and Professor Maritta Heisel for taking time this summer to read my thesis and offer feedback vital to its improvement, and the examiners of my jury: Professor Jean-Luc Danger, Professor Philippe Collet, and Dr. Patricia Guitton, for their interest in my research.

I would also like to thank all of the developers of TTool, fellow members of LabSoc, and co-authors, for the working environment conducive to research, and all of our shared triumphs and disasters, especially: Professor Renaud Pacalet, who originally welcomed me to LabSoc for an internship and inspired me to start a career in research, and taught me the meaning of research and the search for knowledge and understanding, Professor Tullio Tanzi, who provided our case studies on his research on drones and rovers for disaster relief, and has taught me so much about radar and to find the courage to face the rest of the world so different than what I have experienced, Professor Rabea Ameur-Boulifa, who has been an excellent source of knowledge on formalization and verification, Dr. Florian Lugou, who developed so much of TTool's security verification capabilities upon which my research is based, Matteo Bertolino, my officemate and fellow PhD student, who I am sure will forge his own career in academia, Dr. Dominique Blouin, whose work has greatly facilitated TTool development, Professor Daniela Genius of LIP6, our co-author who inspired most of our work on performance and relating between levels of abstraction, and Dr. Andrea Enrici of Nokia, the first of my fellow PhD students to graduate, who has been a great source of information regarding post-thesis life. I also thank the other researchers at Institut Vedecom, Laurent Bonic, Professor Féthi ben Ouezdou, and many others for helping with the modeling and testing on the company side.

I thank my dearest and longtime friends Dr. Sharon Chou, Adam Fagan, Clement Pit-Claudel, Andrea Wang, and Stephie Wu, for their encouragement to see this thesis through, and staying a part of my life from afar. I am grateful for these friendships who have endured time and distance, and that they have ensured that we can meet again in those brief, precious times that we are on the same side of the ocean.

And lastly, I thank my parents, who have always supported me no matter where my education and career take me. They have set an example to strive for education, knowledge, integrity, and scientific achievement, which, in my own way, I have attempted to live up to.

Abstract

The presence of communicating embedded systems/IoTs in our daily lives have brought a myriad of benefits, from adding conveniences and entertainment, to improving the safety of our commutes and health care. However, the flaws and vulnerabilities in these devices expose their users to risks of property damage, monetary losses, and personal injury. For example, consumer vehicles, both connected and conventional, have succumbed to a variety of design flaws resulting in injuries, and in some cases, death. At the same time, as vehicles become increasingly connected (and in the near future, autonomous), researchers have demonstrated possible hacks on their sensors or internal control systems, including direct injection of messages on the CAN bus.

Ensuring the safety of users or bystanders involves considering multiple factors. Conventional safety suggests that a system should not contain software and hardware flaws which can prevent it from correct function. 'Safety of the Intended Function' involves avoiding the situations which the system or its components cannot handle, such as adverse extreme environmental conditions. Timing can be critical for certain real-time systems, as the system will need to respond to certain events, such as obstacle avoidance, within a set period to avoid dangerous situations. Finally, the safety of a system depends on its security. An attacker who can send custom commands or modify the software of the system may change its behavior and send it into various unsafe situations.

Various safety and security countermeasures for embedded systems, especially connected vehicles, have been proposed. To place these countermeasures correctly requires methods of analyzing and verifying that the system meets all safety, security, and performance requirements, preferably at the early design phases to minimize costly re-work after production. This thesis discusses the safety and security considerations for embedded systems, in the context of Institut Vedecom's autonomous vehicle. Among the proposed approaches to ensure safety and security in embedded systems, Model-Driven Engineering is one such approach that covers the full design process, from elicitation of requirements, design of hardware and software, simulation/formal verification, and final code generation. This thesis proposes a modeling-based methodology for safe and secure design, based on the SysML-Sec Methodology, which involve new modeling and verification methods.

Security modeling is generally performed in the last phases of design. However, security impacts the early architecture/mapping and HW/SW partitioning decisions should be made based on the ability of the architecture to satisfy security requirements. This thesis proposes how to model the security mechanisms and the impact of an attacker as relevant to the HW/SW Partitioning phase. As security protocols negatively impact performance, it becomes important to measure both the usage of hardware components and response times of the system. Overcharged components can result in unpredictable performance and undesired delays. This thesis also discusses latency measurements of safety-critical events, focusing on one critical to autonomous vehicles: braking as after obstacle detection. Together, these additions support the safe and secure design of embedded systems.

Abstract

La présence de systèmes et d'objets embarqués communicants dans notre vie quotidienne a apporté une myriade d'avantages, allant de l'ajout de commodités et de divertissements à l'amélioration de la sûreté des déplacements et des soins de santé. Cependant, les défauts et les vulnérabilités de ces systèmes exposent leurs utilisateurs à des risques de dommages matériels, de pertes financières, et même de dommages corporels. Par exemple, certains véhicules commercialisés, qu'ils soient connectés ou conventionnels, ont déjà souffert d'une variété de défauts de conception entraînant des victimes. Dans le même temps, alors que les véhicules sont de plus en plus connectés (et dans un avenir proche, autonomes), les chercheurs ont démontré la possibilité de piratage de leurs capteurs ou de leurs systèmes de contrôle interne, y compris l'injection directe de messages sur le bus CAN.

Pour assurer la sûreté des utilisateurs et des passants, il faut considérer plusieurs facteurs. La sûreté conventionnelle suggère qu'un système ne devrait pas contenir de défauts logiciels et matériels qui peuvent l'empêcher de fonctionner correctement. La "sûreté de la fonction attendue" consiste à éviter les situations que le système ou ses composants ne peuvent pas gérer, comme des conditions environnementales extrêmes. Le timing peut être critique pour certains systèmes en temps réel, car afin d'éviter des situations dangereuses, le système devra réagir à certains événements, comme l'évitement d'obstacles, dans un délai déterminé. Enfin, la sûreté d'un système dépend de sa sécurité. Un attaquant qui peut envoyer des commandes fausses ou modifier le logiciel du système peut changer son comportement et le mettre dans diverses situations dangereuses.

Diverses contre-mesures de sécurité et de sûreté pour les systèmes embarqués, en particulier les véhicules connectés, ont été proposées. Pour mettre en oeuvre correctement ces contre-mesures, il faut analyser et vérifier que le système répond à toutes les exigences de sûreté, de sécurité et de performance, et les faire la plus tôt possible dans les premières phases de conception afin de réduire le temps de mise sur le marché, et éviter les reprises. Cette thèse s'intéresse à la sécurité et la sûreté des les systèmes embarqués, dans le contexte du véhicule autonome de l'Institut Vedecom. Parmi les approches proposées pour assurer la sûreté et la sécurité des les systèmes embarqués, l'ingénierie dirigée par modèle est l'une de ces approches qui couvre l'ensemble du processus de conception, depuis la définition des exigences, la conception du matériel et des logiciels, la simulation/vérification formelle et la génération du code final. Cette thèse propose une nouvelle méthod de modélisation pour une conception sûre et sécurisée, basée sur la méthodologie SysML-Sec, et impliquant de nouvelles stratégies de modélisation et de vérification.

La modélisation de la sécurité est généralement effectuée dans les dernières phases de la conception. Cependant, la sécurité a un impact sur l'architecture/allocation; les décisions de partitionnement logiciel/-matériel devraient être prises en fonction de la capacité de l'architecture à satisfaire aux exigences de sécurité. Cette thèse propose comment modéliser les mécanismes de sécurité et l'impact d'un attaquant dans la phase de partitionnement logiciel/matériel. Comme les protocoles de sécurité ont un impact négatif sur le performance d'un système, c'est important de mesurer l'utilisation des composants matériels et les temps de réponse du système. Des composants surchargés peuvent entraîner des performances imprévisibles et des retards indésirables. Cette thèse traite aussi des mesures de latence des événements critiques pour la sécurité, en se concentrant sur un exemple critique pour les véhicules autonomes : le freinage/réponse après la détection d'obstacles. Ainsi, nos contributions soutiennent la conception sûre et sécurisée des systèmes embarqués.

Contents

List of Figures

List of Tables

List of Abbreviations

AADL	Architecture Analysis & Design Language
AES	Advanced Encryption Standard
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AVATAR	Automated Verification of reAl Time softwARe.
BDMP	Boolean Logic Driven Markov Processes
CAN	Controller Area Network
CFT	Component Fault Tree
CPU	Central Processing Unit
DIPLODOCUS	DesIgn sPace exLoration based on fOrmal Description teChniques, Uml and SystemC
DSE	Design Space Exploration
ECC	Elliptic-Curve Cryptography
ECU	Electronic Control Unit
EVITA	E-safety vehicle intrusion protected applications (project)
FIFO	First In First Out
FMVEA	Failure Mode, Vulnerabilities and Effect Analysis
FPGA	Field-Programmable Gate Array
GPS	Global Positioning System
HAZOP	HAZard and OPerability
HSM	Hardware Security Module
HW/SW	Hardware/Software
IoT	Internet of Things
ISO	International Organization for Standardization
LIDAR	Light Detection and Ranging
MABX	Micro Auto Box
MAC	Message Authentication Code
MARTE	Modeling and Analysis of Real Time and Embedded Systems
MCS	Minimum Cut Set
MDE	Model-Driven Engineering
OBD	On-board Diagnostics
NOP	No operation
OCL	Object Constraint Language
RSA	Rivest Shamir Adleman (algorithm)
SMOLES	Simple Modeling Language for Embedded Systems
SOTIF	Safety of the Intended Function
STPA	Systems Theoretic Process Analysis

SysML	Systems Modeling Language
TCU	Telematic Control Unit
TDMA	Time-Division Multiple Access
TPM	Trusted Platform Module
UML	Unified Modeling Language
VANET	Vehicular Ad hoc NETworks
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle
V2X	Vehicle-to-Everything
VHDL	VHSIC Hardware Description Language
VIN	Vehicle Identification Number

List of Publications

International Journals

- Letitia W. Li, Daniela Genius, and Ludovic Apvrille. **Formal and Virtual Multi-level Design Space Exploration.** *International Conference on Model-Driven Engineering and Software Development*. Springer, 2017. (Accepted)

Book Chapters

- [16] Ludovic Apvrille and Letitia W. Li. **Safe and Secure Support for Public Safety Networks.** *Wireless Public Safety Networks 3*, Elsevier, ed. Daniel Camara and Navid Nikaein, pp 185 - 210, 2017

National Journals

- Ludovic Apvrille and Letitia Li. **Security Concerns of Connected and/or Autonomous Vehicles.** *MISC (87)* (Sep/Oct 2016) (in French)

Conference Papers

- [203] Letitia W. Li, Florian Lugou, and Ludovic Apvrille. **Evolving Attacker Perspectives for Secure Embedded System Design.** *Conference on Model-Driven Engineering and Software Development (Modelsward'2018).* Funchal, Portugal (Jan 2018)

- [119] Daniela Genius, Letitia W. Li, and Ludovic Apvrille. **Multi-level Latency Evaluation with an MDE Approach.** *Conference on Model-Driven Engineering and Software Development (Modelsward'2018).* Funchal, Portugal (Jan 2018)

- [202] Letitia W. Li, Florian Lugou, and Ludovic Apvrille. **Security Modeling for Embedded System Design.** *Fourth International Workshop on Graphical Models for Security*. Santa Barbara, CA, USA (Aug 2017).

- [200] Letitia W. Li, Ludovic Apvrille, and Annie Bracquemond. **Design and Verification of Secure Autonomous Vehicles.** *Intelligent Transport Systems.* Strasbourg, France (June 2017).

- [201] Letitia W. Li, Florian Lugou, and Ludovic Apvrille. **Security-Aware Modeling and Analysis for HW/SW Partitioning.** *Conference on Model-Driven Engineering and Software Development (Modelsward'2017).* Porto, Portugal (Feb 2017)

- [118] Daniela Genius, Letitia W. Li, and Ludovic Apvrille. **Model-Driven Performance Evaluation and Formal Verification for Multi-level Embedded System Design.** *Conference on Model-Driven Engineering and Software Development (Modelsward'2017).* Porto, Portugal (Feb 2017)

- Letitia Li, Ludovic Apvrille, and Daniela Genius. **Virtual Prototyping of Automotive Systems: Towards Multi-level Design Space Exploration.** *Conference on Design and Architectures for Signal and Image Processing* (2016)

- Ludovic Apvrille, Letitia Li, and Yves Roudier. **Model-Driven Engineering for Designing Safe and Secure Embedded Systems.** *Architecture-Centric Virtual Integration (ACVI)*. pp. 4–7. IEEE (2016)

- [209] Florian Lugou, Letitia W. Li, and Ludovic Apvrille, Ameur-Boulifa, Rabea. **SysML Models and Model Transformation for Security.** *Conference on Model-Driven Engineering and Software Development (Modelsward'2016)*. Rome, Italy (Feb 2016)

Under Review

- Letitia W. Li, Florian Lugou, Ludovic Apvrille, and Annie Bracquemond. **Model-Driven Design for Safe and Secure Embedded Systems.** *Under review*

Chapter 1

Introduction

> “The best thing for being sad," replied Merlin, beginning to puff and blow, “is to learn something. That’s the only thing that never fails. You may grow old and trembling in your anatomies, you may lie awake at night listening to the disorder of your veins, you may miss your only love, you may see the world about you devastated by evil lunatics, or know your honour trampled in the sewers of baser minds. There is only one thing for it then - to learn. Learn why the world wags and what wags it. That is the only thing which the mind can never exhaust, never alienate, never be tortured by, never fear or distrust, and never dream of regretting. Learning is the only thing for you. Look what a lot of things there are to learn." –T.H. White, The Once and Future King

Communicating embedded systems/IoTs, are becoming increasingly prevalent in our daily lives [96]. These systems with both hardware and software components, contain a computer ‘embedded’ into the device, and are designed to perform a single dedicated function [25]. The omnipresent connectivity in these objects has improved our daily lives, from adding conveniences by allowing us to track appliances (such as fridges, detectors, etc) [263], adding conveniences and improving safety in our daily commutes (internet connectivity, car monitoring, emergency braking, etc) [342], monitoring our personal health (fitbits, glucose monitors) [77] or of children (Mimo, trackers) [277], to helping medical professionals with menial tasks, monitoring patients or administering treatment (delivery robot, insulin delivery, etc) [314]. For all the benefits due to these connected devices, malfunctions may lead to grave impacts on personal privacy or safety.

1.1 Safety and Security Concerns in IoTs/Embedded Systems

Radiation therapy machines should help treat cancer, but the software bugs within the Therac-25 machine caused malfunctions which sickened or killed 6 patients from 1985 - 1986 [198], and the Cobalt-60 radiation machine sickened or killed 28 patients in 2000 [121]. The Nest Protect smoke and carbon monoxide detector could be turned off easily and unintentionally, and possibly fail to warn users of dangerous situations [6]. Numerous products have been recalled due to the ability to catch fire, such as hoverboards, smartphones, and children’s fitness monitors [176]. Security flaws have also been found on medical appliances, such as the Hospira Symbiq drug pump [153]. Researchers have demonstrated how to gain control

of connected cars through cellular connectivity and wifi [67, 223], and drones through an insecure remote connection [268]. Attacks have also targeted important industrial systems, as demonstrated by the Stuxnet, Flame, and Duqu [219] attacks. All of these examples demonstrate the safety risks posed by flaws or vulnerabilities in connected embedded systems.

Safety is defined as avoidance of situations which can cause losses such as personal injury, illness, property damage, monetary damage, etc [197]. A system should be free of faults, which are defined as undesired system states due to either incorrect commands or the absence of the correct command, or a failure, which is defined as the inability of the system or system element to perform its intended function [62]. In our context, we divide system safety into multiple aspects: conventional safety involves avoiding malfunctions resulting in losses, such as the race conditions within the Therac-25 machine that could cause a fatal dose of radiation to be occasionally administered to patients, and safety of the intended functionality involves avoiding losses due to environmental conditions even in a system without faults, such as malfunctioning sensors in adverse weather conditions [31, 154, 198].

Real-time systems involve software continuously controlling physical components which operate within timing constraints for correct function [123]. In certain real-time systems, performance can also be critical to safety [64, 243]. Critical events should not be delayed due to bus or processor contention, as such delays can lead to potentially unsafe situations or inability to avoid possible damages to the system or users [9, 191].

Furthermore, the safety of a system depends on its security. According to security researcher Charlie Miller, who demonstrated the remote hack of a Jeep through the cellular network, “You cannot have safety without security" [128]. Even if a system was designed to be completely safe, should a hacker gain access to the system, he/she might be able to change the functionality entirely to one which ignores all safety controls.

1.2 Design of Embedded Systems

The design of safety-critical embedded systems is complicated by their many requirements, and the presence of both hardware and software components [145]. Not only must we assure that the system will always behave safely and is protected against attackers, we must also consider the real-time performance for timing-critical devices, memory, device lifetime, the cost and size of the architecture, reliability, and power consumption as many of these devices have limited battery life [25, 184].

Designing secure systems is complicated by the lack of security expertise in developers [20, 284, 306], and the fact that security mechanisms are often added as an afterthought [98]. At the same time, designing safe systems is complicated by the need to ensure both the software’s functional correctness in a variety of environments, and the ability of the hardware to support that software function [144].

Many solutions to ensuring safety and security have been proposed, such as modeling, testing, various methodologies, and following rigorously discussed industry standards. While many safety and security standards exist [287], they are written in text, and it is uncertain if the designer has taken them into account, where quantitative verification is more formalized and uses objective analysis [275]. Formal and semi-formal specifications can be less ambiguous, and better support finding errors and translations for simulation/analysis [28]. Tools can check if formal requirements are consistent and complete as an ensemble, and also automatically check the satisfaction of some types of requirements [238]. We propose

that we can directly analyze if each standard is fulfilled if they are written as requirements that are refined until they can be directly tested (i.e. latency < value). To allow for objective evaluation, we must test a mathematical statement rather than a high-level idea such as "safety should be a consideration".

One solution, systematic modeling and formal verification, can help detect flaws earlier, specify the system, and better analyze the overall system, which individual tests cannot do [290]. Fixing software flaws earlier in the design process costs less than after mass production [137]. Formal verification and simulation can be performed on the more abstract models to validate the design [91], as full systems can be too large to model and take too long to verify, often described as the state explosion problem [61]. Designers then iteratively refine abstract models until the models include all important details [340]. Final models can then be automatically translated into generated code [192, 331], which ensures that the system and models correlate, and eases the software development process [290].

Graphical modeling languages, such as UML are also found to be more 'human-friendly', and easier to understand [291]. A supporting modeling tool should ease modeling and verification efforts, save time by demonstrating good user interface design, and perform automatic verification to clearly present the flaws in the model to the designer [114, 235]. Poor user interface may fail to indicate problems to the user, or provide misleading information. For example, in 1988, the USS Vincennes mistakenly shot down a civilian plane it thought to be hostile partially due to receiving measurements from a different plane and failure to determine if the plane was descending towards the ship to attack, mistakes which can be blamed on a difficult-to-use and uninformative user interface [301]. In the 1989 airplane crash at Kegsworth, the pilots failed to determine which engine was malfunctioning, and then shut down the functioning engine, partially due to a difficult-to-read indicator [43]. While modeling software cannot cause such catastrophic damage, it is still important to ensure our tool provides the correct information in a clear manner to the user to best facilitate the generation of correct designs meeting desired specifications.

1.3 Problem Statement

My thesis, sponsored by Institut Vedecom, the French research institute for the development of sustainable and autonomous vehicles, investigates how to design safe and secure embedded systems. The design of their autonomous vehicle must take into account a multitude of requirements, especially the safety of the occupants and other bystanders. One of the critical stages of development is to decide on the high-level hardware and software, and the mapping of software to hardware (determining the hardware components where functions are executed) [217, 236, 316].

As presented in this thesis, there exist various design methodologies and tools, each which focus on certain aspects of design or specific domains, but none that support all of the necessary verification tools and handle the security modeling required, especially during selection of an architecture and mapping. We therefore evolve the SysML-Sec Methodology and supporting toolkit TTool to better address these needs [13, 15]. As an advantage of our approach, keeping the entire modeling within a single toolkit better ensures that there is only one set of models, and minimizes the amount of rework at each change [157]. At the same time, we ensure the ease-of-use of our toolkit, an essential quality to ensuring its adoption by designers [235]. We discuss our efforts for clear presentation of verification results, to save the designer time by automatically identifying which requirements a model fails to meet.

1.4 Contribution of this Thesis

The contribution of this thesis involves methods to ensure safe and secure design of connecting embedded systems, ultimately cumulating in new modeling and verification methodology. More specifically, the contributions include:

1.4.1 Security modeling and verification in the mapping phase

As to be discussed in Chapter 3, most security modeling describes the detailed implementation of security protocols, and takes place in the last phases of design. However, the choice of architecture should depend on its ability to support security features. When designing an architecture and mapping the functions to architecture, the selection of an optimal mapping relies on correct approximations of execution times of functions [141]. Therefore, the time to execute these security protocols should be considered when selecting an architecture/mapping.

However, placement of security functions depends on which data should be secured, which in turn depends on both the capabilities of the attacker to access the architecture, and which data can be accessed from those architectural locations. While previous works can take into account the performance overhead due to execution of security protocols, they lack the ability to check if an attacker can access or modify critical data [14, 288]. To be certain that security protocols are correctly placed and all security properties are satisfied, a formal security verification process should be used. To assist the designer, it could also be helpful to add security mechanisms automatically based on the verification results, and thus generate a new model fixing all the security flaws.

This thesis describes an approach on modeling attacker capabilities affecting the security of an architecture, and how to abstractly model the functional and architectural security mechanisms for secure communication, including encryption, Hardware Security Modules, and Firewalls [201]. Furthermore, we discuss how to translate mapping models to a formal specification to be analyzed by the security verifier ProVerif. Based on the verification results, the security flaws in the model could be fixed through the addition of various security mechanisms. To actualize these ideas, the security modeling and verification steps are implemented in our toolkit.

1.4.2 Proof of Correctness of Model Transformation for Formal Security Verification

While some formal verification tools conveniently operate directly on program code or graphical models, most use a mathematical specification language that is complex, difficult to read, and unusable for actual design [41]. As we rely on graphical models for their ease of use, to formally analyze our models requires a model transformation process. The correctness of verification results, however, relies as much on correctness of the model transformation as it does the correctness of the model and formal verification tools [220, 328].

We prove the correctness of the model transformation to a formal specification that can be analyzed with a security prover, leveraging in part the proof and transformation process described in [208].

1.4.3 Attacker Modeling

While attack steps can be modeled in Attack Trees, the description of each attack is usually limited to a few words, thus limiting their usage to documentation and not to formal evaluation. Simulations on attack trees can show possible sequences of attack steps, but they do not operate on the actual system, and therefore, the success of a simulated attack does not imply that the attack will succeed in real life. Penetration testing, on the other hand, operates directly on the actual production system, but at the final stages of the engineering process, and after most design has been completed. While successful attacks more definitively imply that security flaws exist in the system, more rework must be done than if the flaws had been detected in the design process, and additional hardware may need to be purchased if the repair requires a modification of the architecture.

We present attacker modeling at a level of abstraction in between attack trees and penetration testing, where the attacker actions can be directly executed on a system model. This thesis describes 'Attacker Scenarios', which explicitly model the attacker actions affecting a system [203]. Simulation examines the effect on the system, including the security property Availability, which could not be previously studied.

1.4.4 Latency analysis

As previously mentioned, timing can be critical in real-time systems. Real-time systems may need to respond to external or system events within a set time frame [218]. For example, a sudden rise in core temperature should be controlled before components are damaged, and a moving system should change course before it collides with nearby objects. Even if a system will respond correctly in terms of function, it is insufficient if the response cannot be executed in time.

Therefore, the latencies between safety-critical events should be analyzed to determine if a system can behave safely [119]. We describe how our approach can automatically measure latencies, and how they can be related across the Mapping and Software Design phases as part of our work to assess if choices/abstractions made during the partitioning phase were correct [118].

1.4.5 Proposition of a modified SysML-Sec Methodology

The modifications listed above result in new modeling elements and verification procedures. Integrating them into the current models involves additional methodology steps. Our additions also add new relations between diagrams that need to be captured in our methodology.

To address safety and security at the same time, it is thus necessary to develop a new methodology taking into account all the different modeling and verification steps. The new methodology better relates requirements and verifications across the different phases, and further integrates attacker models into the modeling and analysis methodology.

1.4.6 Taxonomy for Safe and Secure Autonomous Vehicle Design

As this thesis was under the direction of Institut Vedecom, we present their Level 4 autonomous car in our case study. The VEDECOM Autonomous Car, as shown in Figure 1-1, is equipped with a Velodyne Lidar,

Figure 1-1: Vedecom Autonomous Car

radars, and one front and one rear Mobileye camera.

To prepare our case study, we present a survey of the related issues, such as published safety and security issues, and the possible approaches to solve these issues. However, some of the proposed solutions come with secondary effects. For example, as discussed previously, executing security protocols increases program execution time. We discuss the conflicts and downsides of these solutions in terms of how they affect the safety, security, and performance of a system. These concerns and solutions are then summarized in a taxonomy, on which our design and verification process is based.

1.5 Organization of this Thesis

This thesis starts with presenting our motivation: Autonomous/Connected Cars, in Chapter 2. We describe the architecture of the Vedecom autonomous car, then the possible safety and security concerns, such as reported safety problems caused by software flaws, and demonstrated possible attacks on connected vehicles and their sensors, and then proposed solutions. The aspects of design are summarized in a taxonomy on autonomous vehicle safety and security. Chapter 3 presents the Related Work, regarding other methodologies and toolkits related to software design, design of embedded systems, and safety and security verification techniques.

Chapter 4 presents a proposal for an improved SysML-Sec methodology with the different phases applied to our Autonomous Vehicle case study. Chapter 5 presents the contributions to security modeling in HW/SW Partitioning, and Chapter 6 presents the security verification of mapping models with ProVerif,

and automatic generation of security mechanisms based on those results. Chapter 7 presents the contributions to measuring latencies between events, and how the requirements and latencies are related across different phases. Finally, Chapter 8 concludes this thesis and discusses potential future work.

Chapter 2

Context: Autonomous Vehicles

> "Society tolerates a significant amount of human error on our roads. We are, after all, only human. On the other hand, we expect machines to perform much better. ... Humans have shown nearly zero tolerance for injury or death caused by flaws in a machine." –Gill Pratt, Toyota Research

The introduction of self-driving cars is expected to decrease accidents, ease traffic, decrease pollution, offer transport to the disabled, elderly, and children who cannot drive, and change the very fabric of our daily commutes [101]. Unlike conventional vehicles, autonomous vehicles will rely entirely on software and sensors, instead of potentially flawed human decision-making for control. To ensure the safety of passengers and other individuals in proximity, manufacturers must ensure the safe and secure function of the vehicle software. According to Guillaume Duc, professor at Telecom Paristech and chair of the Connected cars and cybersecurity project, "Autonomous cars will not exist until we are able to guarantee that cyber-attacks will not put a smart vehicle, its passengers or its environment in danger." [156].

As with other embedded devices, vehicles have not been free from design flaws as demonstrated through the years. Even worse, while the increased connectivity of vehicles has offered new conveniences, safety measures, and comforts for drivers, they have also created avenues for attack for hackers.

2.1 Safety and Security Flaws

2.1.1 Safety Flaws in Commercial Vehicles

Since the introduction of automobiles, there have been design flaws posing safety risks, some of which resulted in injuries and deaths of the occupants [120, 199]. Many involved mechanical failures which resulted in multiple injuries and deaths.

Due to its placement, the fuel tank of the Ford Pinto was found to susceptible to fires in the event of a rear-end collision, causing hundreds of deaths in the 1970s [343]. Even worse, it was proved that Ford was aware of the flaw and chose not to correct it, resulting in multiple lawsuits. Ford ultimately recalled 1.5 million vehicles.

In 2009, it was reported that Toyota vehicles were involved in multiple accidents due to uncontrollable sudden accelerations due to a stuck gas pedal [333]. Multiple fatalities resulted from accidents when vehicles accelerated despite the driver attempting to brake. Ultimately, Toyota recalled millions of vehicles. In 2017, BMW recalled 1 million vehicles due to a fire risk [272]. There were around 40 cases of parked cars catching on fire due to valve heater or wiring problems, in some cases resulting in fire damage to the owner's garage and house as well.

Software flaws have also prompted the recalls of vehicles. For example, in 2016, Hyundai recalled SUVs since the transmission control computer had an intermittent software problem which could prevent the vehicle from accelerating [232]. After at least 1 death and multiple injuries due to a software flaw that prevented airbags from deploying during accidents, in 2016, General Motors recalled millions of vehicles to be repaired.

These design flaws have caused injuries and death, and cost manufacturers millions in fines, lawsuits, or recalls.

2.1.2 Survey of Hacks on Connected Vehicles

The authors of [159] presented privacy and security risks of a Tire Pressure Monitoring System. By monitoring RF signals from the sensors, the authors found the unique sensor ID for identification of the vehicle, and spoofed packets to send fake tire pressure warnings. This attack could force a driver to pull over thinking he had a flat tire, and also the ability to track cars by their wireless signals is a privacy concern.

Telematics units can be attached to cars, and often used for insurance purposes, have been an avenue of attacks on the operation of vehicles themselves. The authors of [188] performed an attack through the OBD-II port using the CAN-to-USB interface. The authors built a CAN packet sniffer/injector and determined how to control units, many commands discovered through fuzzing. Furthermore, the attack restarts the ECU and erases any evidence of the attack code. However, the attack required physical access to the OBD device.

Another minor attack [345] used a malicious application on a paired smartphone and a car with an OBD-II scan tool. When an OBD-II and smartphone with diagnostic application are connected through bluetooth, an attacker can send malicious CAN frames to the vehicle. While it required the user to accidentally install the malicious application and for the car to have a telematics unit installed, it demonstrates another avenue of attack.

The authors of [109] analysed the Metromile TCUs, an aftermarket device interfacing with the OBD-II port. The ssh keys were common to all TCUs and could be acquired by dumping the NAND flash. Updates to the TCU were sent through SMS messages, which the authors used to create a new console starting a remote shell to obtain access to the device through ssh. The authors note that the update did not have the vehicle verify the server's identity, which is a major vulnerability.

New phone apps allow users to control comfort settings. However, vulnerabilities in these apps, such as lack of authentication protocols, can be a problem. One such example is the WebAPI for the Nissan Leaf, which required only a VIN number to access certain climate control and status of a vehicle [150]. While they did not offer an avenue for attack on the car operation itself, except for draining the battery, researchers were able to recover driving history, which may be a privacy concern.

Most notably, Miller and Valasek's hack is completely remote and prompted a recall and change in Sprint's network [223]. The authors connected to the built-in telematics unit Uconnect's Diagnostics Bus, open to any 3G device using Sprint. They reprogrammed the unit with custom firmware to send CAN messages and control ECUs. This attack is unique as it could attack a vehicle anywhere on the Sprint network and did not require that the vehicle have more than the default setup.

In 2016, researchers from Tencent presented how to gain control of a Tesla that connected to their malicious wifi hotspot [67].They reported that a vulnerability in the Tesla's browser would allow them to run code if it visited their page. The malicious code helped them gain access to the car's head unit, which they used to overwrite the gateway to the CAN bus, and subsequently inject their own commands onto the CAN bus. Tesla has reportedly responded by requiring that firmware of components writing to the CAN bus be code signed. However, during Defcon 2017, the same group of researchers demonstrated that they could again gain remote control of a Tesla via cellular data and wifi [337].

Recently, it has been demonstrated that Denial-of-service attacks against the CAN bus could disable safety features [246]. By sending multiple erroneous frames to a CAN node, the node could be completely turned off, preventing it from sending and receiving any more messages. In this case, the authors needed physical access to the vehicle to carry out the attack, but their attack could be combined with the previous remote attacks on the CAN bus.

In all of these attacks, connected features have exposed their users to potential privacy or safety risks.

2.1.3 Safety Limitations of Autonomous Vehicles

Unlike conventional vehicles, autonomous vehicles rely on their sensors to perceive the world around them. Any failure in a sensor could lead to a grave impact on vehicle safety. However, it has been noted that these sensors are not always reliable.

The GPS is vital to determining the current vehicle location, which helps a vehicle determine not only it's trajectory, but also the nearby traffic lights, signs, and etc [66]. However, GPS signals can be easily blocked, in a tunnel or due to tree cover, for example. Its resolution is also not precise enough to navigate without other sensors.

In 2016, the Tesla Autopilot caused a fatal crash when its perception system failed to distinguish a white tractor-trailer against a bright sky [180]. In 2018, a driver again running Tesla autopilot died in a fatal crash after the vehicle drove into a concrete barrier on the freeway [300], likely due to confusing lane markings. The Tesla autonomous vehicle uses only radar and cameras, which other developers find insufficient, as radar fails to see detail and cameras can fail in problematic lighting conditions, such as the glare in this situation [295]. Lidars have been suggested to be vital, as they are more accurate than radar, and can provide a 3D image [90].

However, Lidars may fail to function in all environmental conditions as well. Snow and rain may confuse both lidar and radar systems [216, 229]. They may also obscure the lane markings, though Ford demonstrated that their autonomous car may navigate in snow with sufficient use of surrounding landmarks even if the lane markings are unavailable [229].

In 2018, a Uber self-driving vehicle killed a pedestrian in a collision [323]. While the sensors detected the pedestrian, the software ignored it as a false positive and did not brake. Distinguishing between significant

obstacles requiring emergency braking and insignificant obstacles (such as plastic bags blowing across the road) is a difficult task, as braking constantly would be annoying to passengers, and failing to brake for a pedestrian or larger obstacle leads to a collision and possibly injuries.

It has also been proposed that autonomous vehicles cannot read human cues, whether of pedestrians or human drivers [44]. The autonomous car cannot read hand signals giving right-of-way at an intersection, or a pedestrian's intention to traverse a cross-walk or to wait for the car to pass. If autonomous cars yielded to every pedestrian, as would be safest, they would unfortunately slow down traffic.

Human interaction is also involved in situations such as changing lanes. The Google autonomous car was found to be at fault when it crashed into a bus when merging into a lane, as it expected the bus to slow down or stop. Human drivers might be more adept at reading the bus driver's behavior and signals [229].

Furthermore, the road infrastructure is often imperfect, with degraded or missing lane markings, non-functional traffic lights, and etc [216]. The autonomous vehicle must navigate despite such cues. Other situations have been cited to be difficult for autonomous vehicles, such as bridges and urban driving [229]. Bridges lack environmental cues such as landmarks, which help a vehicle determine its exact locations. Navigating cities involve too many obstacles, pedestrians, and cars to process safely, and the GPS may be blocked by tall buildings.

Each sensor has conditions under which it functions unreliably, and an autonomous vehicle must take these limitations into account and interpret their data accordingly to ensure it correctly perceives the surrounding area.

2.1.4 Survey of Potential Attacks on Future Autonomous Vehicles

Even in perfect environmental and operational conditions, sensors cannot be assured to be functional or completely reliable, as they may be susceptible to malicious attackers. While autonomous vehicles remain in the development phase and there do not exist production models to attack, researchers have proposed proof-of-concept attacks on their sensors.

The authors of [255] demonstrated attacks on cameras and LIDARs. First, they showed that it was possible to blind the camera with a laser. They next demonstrated that it was possible to cause the LIDAR to detect fake objects. The authors of [294] enhanced the lidar attacks. From a greater distance, the authors could fool the sensors into detecting fake objects.

Researchers have also shown how to spoof and jam the ultrasonic parking sensors and Millimeter-wave Radars on Teslas [346]. They were able to show that by using signal generators, they could prevent detection of obstacles, or cause the sensors to detect non-existent obstacles. These spoofing and jamming attacks could force unsafe behaviors such as crashing into an obstacle or braking suddenly for no reason.

GPS spoofing was demonstrated by researchers from the University of Texas [149]. The authors demonstrated how they created false GPS signals to redirect a GPS-guided drone, and then later misdirected a yacht. Autonomous vehicles greatly rely on GPS for both their location and determination of surrounding environmental data, so this threat could greatly affect vehicle navigation.

The authors of [40] demonstrated that sound waves, such as from ultrasonic sensors, can affect hard drives, or more specifically, the shock sensors within the hard drive head unit. The attacks could prevent reads and writes on the hard drive, slow down, damage certain sectors, or become non-operational. Attackers

may potentially use the ultrasonic sensors within autonomous vehicles to either damage the hard drive of the control computer, or spoof other sensors.

Vehicular Ad hoc NETworks (VANETs) offer a collaborative exchange of data on traffic, environmental hazards, etc, which may decrease accidents and traffic jams. However, as predicted, malicious participants may track the location of a vehicle, or send fake traffic data to clear a road for themselves. In addition, V2X involves connections to numerous vehicles or roadside units. This connective interface offers opportunities for an attacker to access the interior control systems and possibly gain control of the vehicle itself [248].

Instead of hacking the autonomous vehicle directly, researchers have studied how seemingly invisible changes to street signs can fool image recognition software [122]. The policemen, government officials, and human drivers will not notice a difference, but these modified signs could confuse an autonomous vehicle's perception and cause it to ignore a street sign, and possibly provoke an accident.

These examples show the vulnerabilities of sensors, and how they send misleading information to the autonomous vehicle due to malicious individuals possibly attempting to provoke an accident or other undesired situations for personal gain.

Whether due to malicious attackers or naturally-occurring environmental conditions, connected and autonomous vehicles face a range of safety and security hazards, as summarized in Table 2.1, adapted from our ITS paper [200].

Table 2.1: Table of Hazards of Autonomous/Connected Vehicles

Hazard	Attack/Fault	Reference
Remote Control of Vehicle by Attacker	Wifi/3G network	[67, 223, 337]
Control of Vehicle by Attacker	Telematics Unit attached to OBD-II port	[109, 188, 345]
Safety feature shutdown	Denial-of-Service attack on CAN bus	[246]
Falsified Sensor readings	Camera/Lidar/Radar	[255, 294, 346]
Falsified GPS signals	GPS	[149]
Loss of Privacy	Smartphone App/Tire sensor	[150, 159]
Misinterpreted Traffic Signs	Sign Modification	[122]
Poor Lidar/Radar Data	Snow/Rain	[216, 229]
Poor Camera Data	Darkness/Sun Glare	[180, 295]
Loss of GPS Signal	Buildings/Tunnels	[66]
Poor Road Infrastructure	Damaged Lane markings/Power Outage	[216]

2.1.5 Conclusion

In this section, we presented the flaws, limitations, and vulnerabilities in vehicles and their sensors. No sensor is reliable in every single situation, and these faults and attacks demonstrate the need for multiple independent sources for information and resilient design. Furthermore, malicious entities could take advantage of the connected nature or predictable limitations of sensors to provoke accidents. The next section presents approaches to assure the security and safety of autonomous vehicles despite these vulnerabilities.

2.2 Approaches to Vehicle Safety and Security

As presented in the previous section, autonomous vehicles must overcome the limitations of their sensors and prevent their connected nature from offering access to hackers. Other works have discussed sensor design techniques to prevent jamming/spoofing [207, 254] or acoustic interference [40], but we focus on the solutions related to designing the internal control system of the vehicle and how they can interface with the sensors. The simplest solution adopted by every autonomous vehicle developer is to use multiple sensors. Other solutions, like accurately 'fusing' the data from various sensors, avoiding other safety issues, and developing a secure vehicle architecture have been the subject of various projects and papers.

2.2.1 Proposals for Safe and Secure Automotive and Embedded System Design

APSYS and Vedecom have proposed many techniques for ensuring the safe comportment of an autonomous vehicle [39]. To avoid flaws in a single sensor or algorithm, their propositions rely on redundancy and data fusion. The data from multiple sensors should be combined based on their reliability in the current environmental conditions, and the nearby obstacles are calculated with multiple perception algorithms. The coherence in detected variables across sensors should be evaluated, as one sensor giving incongruous data may be due to an error. Based on the perception data and destination, trajectories are also calculated with multiple algorithms, and the possible trajectories are evaluated again based on their level of confidence, past record, coherence to determine a final trajectory which should be safe. When significant errors, data incoherences, or other conditions preventing safe autonomous driving are detected, the system can enter a fail-safe mode, where the driver should regain control or the vehicle should safely navigate to a stop [79].

Redundant hardware has been used in other projects [83,189,320] to ensure a system is more fault tolerant. For example, the UAVCAN project connected mission critical devices to a backup bus in case the primary bus failed. However, while redundancy improves fault detection and tolerance, it increases the amount of hardware used and therefore the cost of the system.

The standard ISO26262 describes how to ensure safety in vehicles [160]. It proposes a safety lifecycle, including hazard and risk analysis, development of safety goals, integration of safety measures, and assessments such as safety validation [50]. Their suggested methodology shown in [50], is based on the V life cycle. Security goals can be analyzed in a similar way as safety goals. By systematically following these steps with safety and security validations, designers and managers should design a system minimizing flaws and vulnerabilities.

Many European projects have recognized the importance of studying security of connected vehicles, and proposed solutions for aspects of vehicle security.

Like many other works [51, 247], the PRESERVE project studied how to ensure security in V2X communications [171]. Their V2X Security Subsystem, which involves software for secure communications such as pseudonym management and cryptography, can be added to all nodes in the V2X network. For more efficient cryptography, an ASIC-based Hardware Security Modules was also developed and can be added to the system [228].

Similarly, the SEVECOM (Secure Vehicle Communications) project proposed an architecture including different modules added to a vehicle for V2X communications, including a Pseudonym Manager, Hard-

ware Security Modules, and etc [194]. They also added an 'In-Car Security Module' to interface with the vehicle systems using a firewall and intrusion detection system to prevent attacks on the internal communications from the open V2X communications.

The Open Vehicular Secure Platform (OVERSEE) project intended to develop a secure communication platform for all possible vehicles [244]. New applications, such as e-tool, V2X, and remote diagnostics, will be added to a vehicle's network, which may pose a security risk. They intended to develop of a single platform providing a secure runtime environment for applications and communicating with the vehicle. The OVERSEE platform connects to the in-Vehicle networks, at a single point of access to facilitate security, with a firewall protecting the internal network. Their platform will also allow the development of platform and vehicle-independent applications, as the OVERSEE platform and not the vehicle software will be directly executing the programs.

Plausibility/Coherence checks have been proposed for use in VANETs, to help detect maliciously-sent data [32, 241]. However, they have also been suggested for use in cyber-physical and industrial systems to help detect failing components or attacks [59, 190, 321]. Various detection schemes, such as monitoring the entropy between related clusters of sensors, help detect when abnormal data is being sent into the system.

[253] suggested the addition of firewalls between different ECU domains. Certain vehicle nodes have no reason to communicate with one another should be separated by a firewall. As there have been propositions that attacks to the CAN bus might be possible via the Infotainment center, the vehicle controls should be isolated from those components. The authors also suggest that Denial-of-Service attacks such as SYN flooding should be prevented by the firewall. It has also been suggested that firewalls should be added to IoTs in general [3]. Authentication and encryption are argued to be insufficient as weak passwords and brute force attacks may allow an attack to succeed. A firewall can enforce security policies that block most cyber-attacks, by blocking certain ports, protocols, and unauthorized IP addresses.

Various anomaly detection schemes on the CAN Bus has been proposed as a solution. Anomaly detectors have used the Hierarchical temporal memory (HTM) algorithm [336], expected sequences of message IDs 7995934, support vector machines on packet frequency [308], and other machine learning approaches. Anomaly/intrusion detection may therefore detect some attacks when a hijacked node sends out messages to carry out an attack. [309] presented a framework to simulate attacks and evaluate their anomaly detector, which resulted in a generally high success rate, though it has not been tested on actual attacks or with an attacker who might can adapt his approach to bypass the anomaly detector.

The work in this thesis follows along the recommendations from the EVITA project (E-safety vehicle intrusion protected applications), another a European collaborative project completed at the end of 2011, has defined a complete secure automotive architecture, with hardware accelerators and security protocols in particular [100]. The EVITA project assumed a 'Dolev-Yao' attacker who can listen to all traffic as well as inject data on the buses. The security recommendations included adding hardware security modules, periodically distributing keys, and isolating domains and monitoring for intrusions using firewalls [292]. An EVITA-compatible HSM can be added to ECUs. For the different cost and security requirements, three types of HSMs were defined [27, 142]. The EVITA HSM Light Version is the simplest version which performs cryptographic operations and protecting keys, and is intended for sensors and actuators. The HSM Medium Version, which additionally supports additional cryptographic operations, though only in software, and a secure boot process, is intended for domain controls. Finally, the largest HSM Full Version supports hardware acceleration for cryptographic operations such as Whirlpool and ECC, is intended for V2X applications [281].

[151] proposed similar methods for embedded systems in general. The authors proposed that security in embedded systems must apply across many levels, from hardware up to the detailed security protocols. The architecture involves both secure and insecure modules across which functions must be mapped. Security requires additional hardware, power, and computation time, so only the vital functions should be protected. Their secure modules involved use of security co-processors, with a memory resistant to attack.

Similar to Hardware Security Modules, [324] also presented a device for securing embedded systems, the Lincoln Laboratory's Lincoln Open Cryptographic Key Management Architecture (LOCKMA). LOCKMA also is implemented in a security co-processor which uses dedicated hardware for more efficient cryptography. Their device is also designed to securely boot, resist tampering, and a physical unclonable function for device identification.

To prevent attacks on the sensors of autonomous vehicles, [183] suggested a system of watermarking. A varying signal was added to actuators, and if the sensor measurements did not show the watermark, then the system would know that the sensors were tampered with. The statistics of each sensor's watermark is known to other sensors. Their experiments involved sending forged position sensor communications to the collision avoidance module. In 1 out of their 2 preliminary tests, the watermarking helped the system resist the attack and maintained the correct trajectory. Another work to resist sensors that were tampered with, [245], used mathematical models to estimate the state of the vehicle and compared expected values with measured values. Their system can better identify attacks and which sensors are compromised.

And while viruses have not yet been demonstrated to infect connected vehicles, embedded devices were demonstrated susceptible in the Mirai botnet DDos attacks [233], and fitbits were shown to be theoretically capable of being hacked and infecting any nearby computer with malware [11]. Similarly, it has been proposed that equipment used by mechanics for vehicle diagnostics could be infected with malware and spread it to other vehicles and diagnostics tools [127]. In response, anti-virus software for cars has been proposed, including by the company Argus Cyber Security [134, 348].

2.3 Taxonomy

To summarize the various issues regarding autonomous vehicle safety and security, we present a taxonomy of the risks, vulnerabilities, results, and countermeasures, adapted from the taxonomies presented in [46, 312]. This taxonomy relates the various elements relevant to considering the safety and security of autonomous vehicles, and presents an overall high-level view of the entire system and its needs and risks. This taxonomy focus on the risks posed by the autonomous system alone, and does not present the risks of a non-connected car (such as mechanical failures). It can also be adapted for analysis of other embedded systems, as the only changes required would be on the specific external factors affecting behavior, and the specific unsafe and insecure behaviors.

As shown in Figure 2-1, the autonomous vehicle system, composed of hardware and software, executes as a system behavior. The behavior of the autonomous vehicle is affected by factors provoking failures, both internal or external, including coding bugs or poor weather conditions. The failure causes may then cause the behavior of the system to violate desired safety and security properties of the system, which then in turn invoke undesired, unsafe comportments, such as ignoring an obstacle. To prevent the undesired final comportment of the autonomous vehicle, countermeasures can be added. Figure 2-2 and Figure 2-3 show the detailed complete taxonomy divided across the figures for readability. The first figure shows the factors

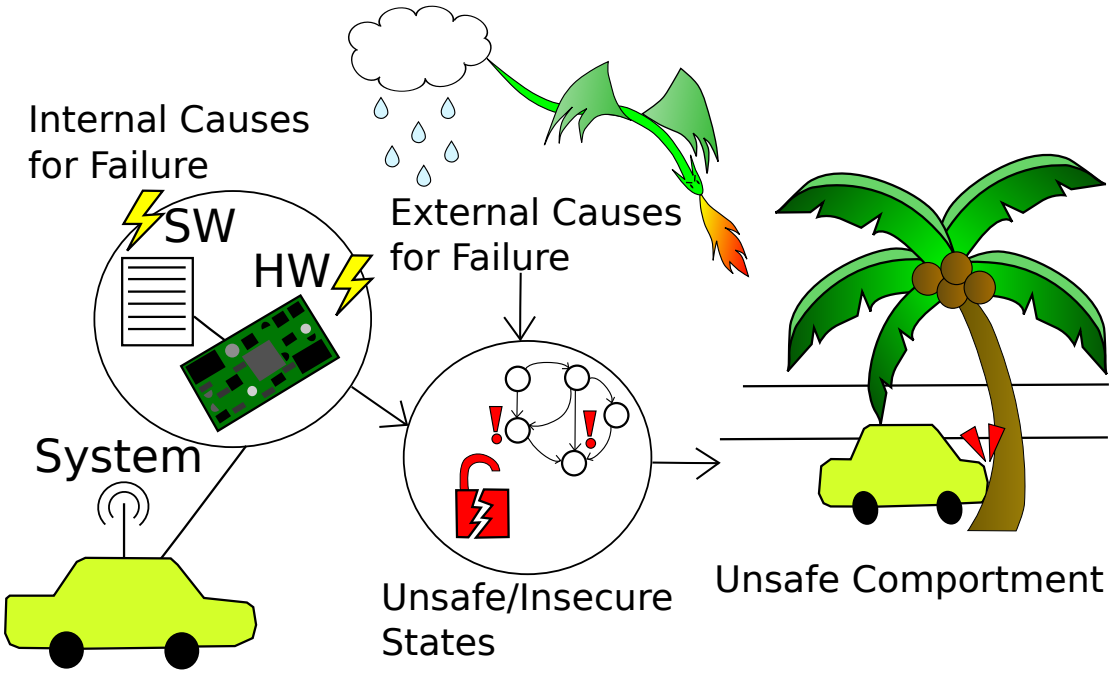


Figure 2-1: Taxonomy Overview showing how Internal and External Factors can result in Unsafe Comportment

causing or permitting failures or attacks and the countermeasures to prevent them, while the second figure elaborates on the exact unsafe and insecure properties that can lead to dangerous real-world behaviors.

2.3.1 Potential Causes of Failure

Many factors, internal and external, can permit or induce system failures.

Internal factors which may allow the system to enter an unsafe or insecure state involve system flaws, vulnerabilities, and limitations. For example, **flaws** such as software bugs due to coding mistakes might cause a system to function differently than specified, hardware flaws such a lossy channel may result in loss of important system messages, and CPUs with too many tasks mapped on them may fail to calculate vital system commands in time. **Vulnerabilities** can involve ports or communications accessible to an attacker, software flaws leaving the system vulnerable to buffer overflows, or a security protocol used that turns out to be insecure. **Limitations** involve Safety of the Intended Function (SOTIF) elements, where otherwise functioning elements are not effective in all situations, such as sensors which do not function in all environmental conditions [229] and algorithms which cannot process all driving situations, especially gestures by a policeman signaling traffic [298].

External factors which can induce unsafe or insecure behavior include an attacker, or environmental conditions. **Attackers**, as described in Section 2.1.2 and 2.1.4, can modify code, manipulate sensors, and inject messages to control the behavior of a system they should not have access to. As described in section 2.1.3, sensors can be affected by **Weather conditions**, such as darkness, rain, snow, and fog. Poor **Infrastructure conditions**, such as a non-functional traffic light, and poor **Road Conditions**, such as roads in poor condition, may also make accidents more liekly. **Traffic conditions**, such as unexpected

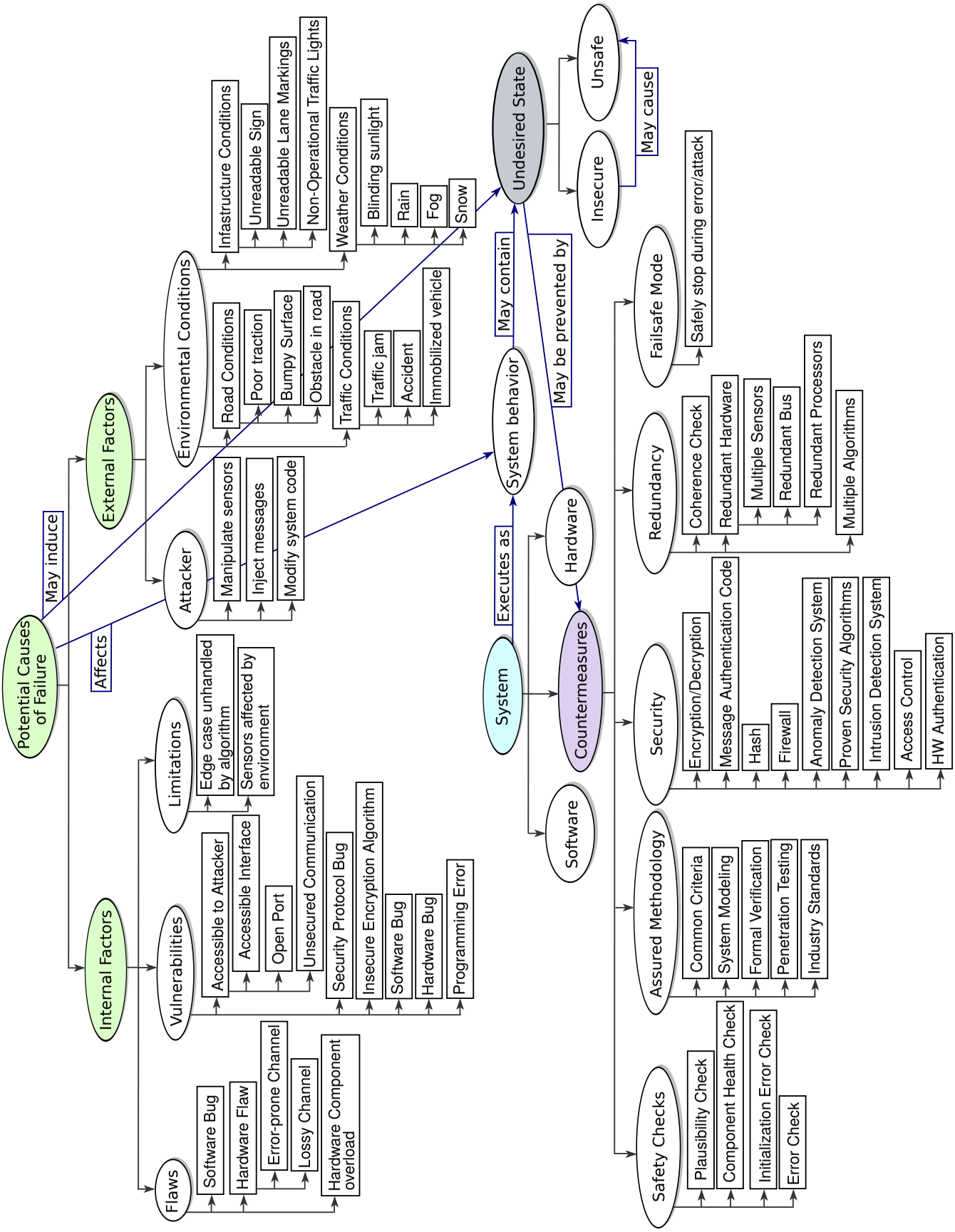


Figure 2-2: Taxonomy for Autonomous Vehicles Part 1

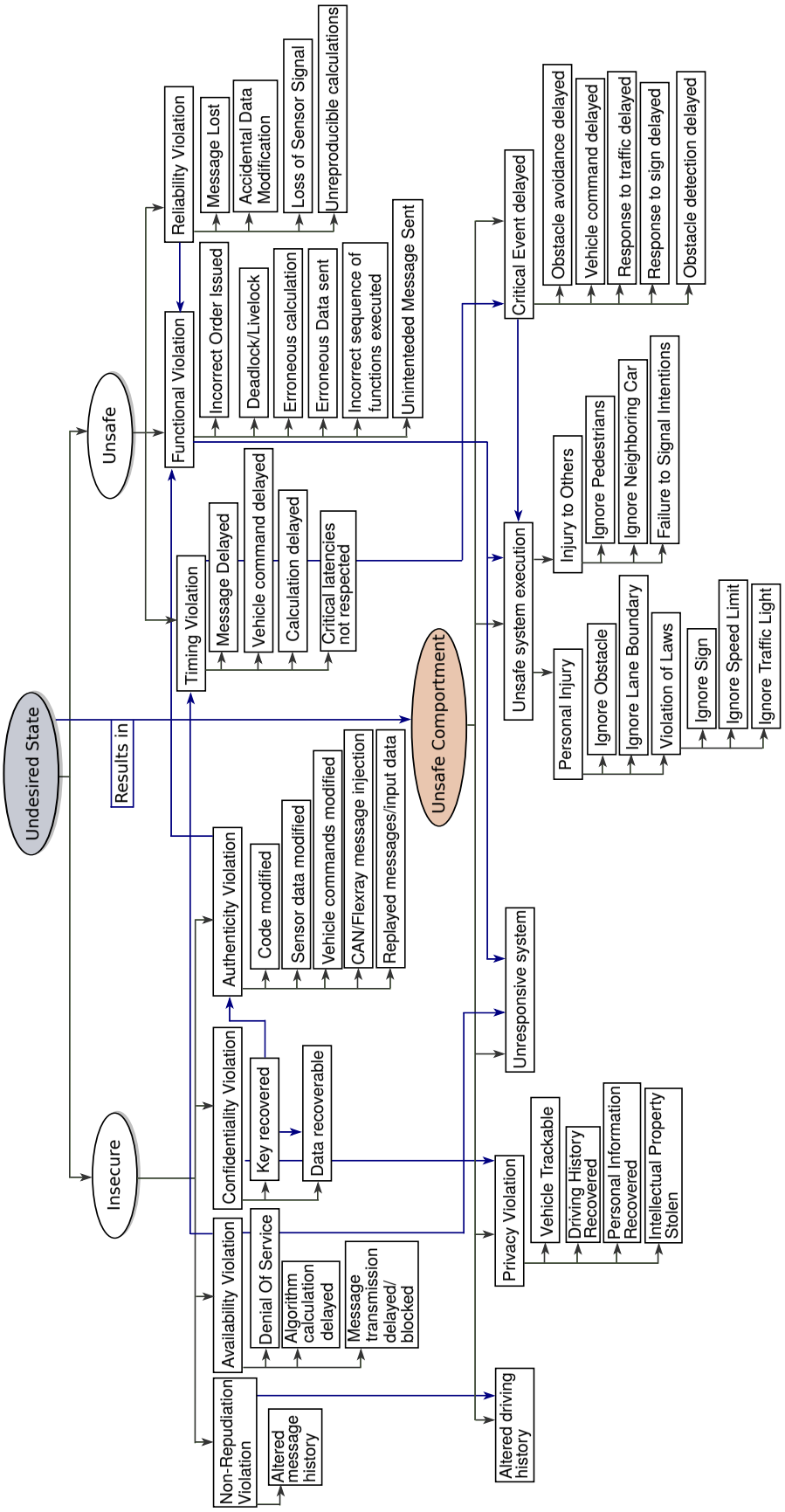


Figure 2-3: Taxonomy for Autonomous Vehicles Part 2

traffic jams on a freeway, or accidents between other vehicles, may also be difficult for an autonomous vehicle to navigate.

A combination of such factors may induce a failure, such as an attacker taking advantage of an insecure security protocol to inject messages to gain unauthorized control of a vehicle, or driving in adverse weather on a road with poor lane markings may cause the vehicle to fail to stay in lane.

2.3.2 Undesired States of System Behavior

The undesirable states that a system can enter due to the potential causes of failure can be classified as unsafe or insecure. As shown through the blue arrows which shows causation relationships, violation of security properties can lead to unsafe states.

2.3.2.1 Insecure States

Non-Repudiation violations occur when an entity sent a message or performed an action, and then he/she is capable of denying it. The history of actions taken by a vehicle may be vital to a lawsuit assigning responsibility in an accident, and may be important in V2X to determine if a vehicle has been sending false information [54]. However, we do not study this property in this thesis.

Availability violations occur when an attacker can prevent the system from providing a necessary service. The availability of a system can be compromised by denial of service attacks. A subsystem may become unavailable if the attacker can delay calculations or message transmission.

Confidentiality violations occur when an attacker is capable of recovering sensitive data. For example, if an attacker can access a cryptographic key, he/she can recover data and also use the key to forge messages, resulting in Authenticity violations.

Authenticity violations occur when an attacker can forge messages or data that are accepted by the system. For example, an attacker might modify the system code, or inject vehicle commands to provoke an accident.

2.3.2.2 Unsafe States

Timing violations occur when the system does not respond to events in time as required. In a real-time system such as an autonomous vehicle, calculations should be performed, and messages should be sent in time for the vehicle to respond to events such as the appearance of an obstacle. Availability violations may provoke timing violations when the subsystem fails to send critical commands before a set deadline.

Functional violations occur when the vehicle behaves against requirements, such as issuing driving orders that would send the vehicle off the road or into an accident, deadlocks, incorrect calculation of a trajectory or obstacle locations, etc. Some of these incorrect functionalities may be induced by an attacker who can modify system function or messages.

Reliability violations occur intermittently due to system flaws, such as the loss of a message due to a lossy channel. The unreliable behavior of a system may also cause functional violations, such as incorrect

behavior due to the lack of reception of a critical message.

2.3.3 Unsafe Comportment

If the system behavior falls into an undesired state, then its comportment, or real-world behavior, may be undesired as it risks losses due to privacy violations, legal issues, or dangerous behaviors. For example, if the system fails to ensure non-repudiation, then the **vehicle driving history might be altered**, and the driver could deny fault in an accident. **Personal Privacy** violations, such as an unauthorized entity gaining access to a vehicle's driving history, are caused by confidentiality violations on driving history or other information that can identify the vehicle.

An **unresponsive system** can occur due to availability violations, reliability violations, or deadlocks. **Unsafe system executions**, ones which risk the safety of occupants, pedestrians, or neighboring vehicles, can occur due to incorrect system function, or the **delay of a critical event**. For example, a vehicle failing to respond in time to an obstacle could lead to an accident and possible damages or injuries to the occupants or bystanders.

2.3.4 Countermeasures

To prevent the system from entering these undesired states and behaving unsafely, countermeasures should be included in the system. Countermeasures can involve using an assured design process, validation/verification, and testing, or safety and security countermeasures as described in the following section. Countermeasures can involve **Safety Checks** checking the plausibility of data, or that all components initialize and function correctly without errors, **Redundancy** of hardware or software, and checking that the redundant data received are indeed coherent, **Failsafe Mode**, which will bring the vehicle to a safe stop when errors or other conditions prevent the autonomous driving utility from functioning safely, and **Security** mechanisms, such as methods of securing data, or communication monitoring such as anomaly detection systems.

Some of the countermeasures can be directly linked to the undesired states that they prevent, such as encryption or message authentication codes preventing message injection, or redundancy preventing reliability issues such as messages lost due to a lossy channel. We will discuss some of the links but do not show them on our taxonomy as it would make the diagram too crowded and complex.

2.3.5 Conclusion

The presented taxonomy offers an overview of the factors affecting safety and security in autonomous vehicles, which summarizes published works on this topic and better relates them into a coherent framework. We can use this taxonomy to consider how we can avoid potential safety and security hazards, and the unsafe situations from which they arise, by integrating the associated countermeasures.

2.4 Countermeasures

Based on the taxonomy presented, and previous projects and publications, we select the most commonly-agreed upon solutions applicable to the design of embedded systems. This section summarizes the main countermeasures which we study in the rest of this thesis.

2.4.1 Safety Countermeasures

While countless methods for assuring the safety of a system exist, we focus on the main ones used in the Vedecom car.

Coherence checks can help ensure the correct functionality of a system despite forged or erroneous sensor data. If there is a significant discord in the sensors, then it is possible that an attacker is spoofing one of the sensors, or one of the sensors is malfunctioning, and therefore the user should be warned.

Plausibility checks take into account the possible range of values and historical data to filter input data. For example, an extreme jump in the current location from the GPS in a short interval is impossible, and is likely due to a malfunction or reception issue. The MABX box, which converts vehicle commands for the ECU, also performs filtering for safety. The vehicle commands can be accepted or rejected based on the maximum allowed acceleration, braking, turn, depending on the current speed, etc. Again, due to the lack of specific data values, we again model the plausibility check as a function with computation complexity, and then the possibility to either accept or reject the received data.

Redundancy of vital functions and sensors helps to ensure system function in critical components. Even if one component malfunctions, either the vehicle should continue to function safely, or enter the failsafe mode by warning the occupants and navigating to a safe stop. The main function of the autonomous car is the Perception unit, which takes in all the sensor data and generates the set of all obstacles in the surrounding area. No matter how rigorously it is tested, numerous combinations of sensor measurements and obstacles exist, and any single perception algorithm may still have flaws. Furthermore, if the processor or communication bus should fail, then no perception data would be sent to the supervisor, preventing the system from continuing to function.

2.4.2 Security Countermeasures

Our primary security countermeasures protect our system or data against an attacker. Certain internal communications of the system may be accessible to the attacker, so it is important to ensure that the attacker cannot recover/tamper with important data that could result in undesired situations. Encryption mechanisms prevent an attacker from being able to recover (and understand) certain data. Message Authentication codes can be added to a message so that the receiver can determine that the message has not been modified. Timestamps and nonces can also be used to prevent duplicate messages from being received and accepted in a replay attack.

Hardware Security Modules (HSMs) have been suggested in EVITA, PRESERVE, SEVECOM, and other works on embedded system security [92]. They perform security protocols and protect the cryptographic keys that they contain, conforming with the requirement proposed of secure memories for cryptography. They are assumed to be protected again software and hardware tampering. In addition, they may contain

cryptographic accelerators which perform encryption faster than regular processors. However, they are an additional hardware component added onto the system. Commercial Hardware Security Modules include ARM Trust Zone, Infineon Aurix HSM, and etc, some of which are EVITA-compatible [293].

Firewalls separate the different subsystems with different levels of security. They can isolate untrusted communications to prevent them from accessing critical internal systems. Firewalls can be either hardware or software firewalls, or both. [70] discussed how firewalls implemented in hardware can protect an embedded system. Their hardware firewalls prevent attacks from recovering sensitive information in internal system communications and external memories. [107] suggested a hybrid firewall, since hardware firewalls offer greater throughput but cannot handle more complex rules. Their approach splits the processing between hardware and software, where complex rules are implemented in software, while simple rules are managed by the FPGA-based firewall. Icon Labs has proposed various IoT security products [152], including the Floodgate Defender firewall, which can be added in between an embedded device and the Internet, and configured with security policies.

Intrusion or anomaly detection systems can detect both attacks or component malfunctions [173]. If an attack is detected, then the system may warn the user, or enter a failsafe mode. Commercial products are also available for vehicle manufacturers, such as from Symantec's Anomaly Detection software [45].

Another important security consideration is ensuring code integrity, so that the software itself cannot be modified. As some demonstrated attacks have involved modifying system code, code signing techniques and Trusted Execution Environments should prevent these attacks. The Intel Software Guard Extensions (SGX) helps protect code and data, even in an insecure environment [69]. Other software integrity approaches involve secure boot, software attestation, and etc [208], which may involve a Trusted Platform Module, a hardware component that determines that the running software has not been tampered with [318].

While there exist other more specific security mechanisms and protocols, this thesis focuses on how to abstractly model security for the selection of an architecture, so we group them into their general classification of encryption, MAC, etc.

Table 2.2: Table of Countermeasures

Countermeasure	Additional Hardware?	Addressed Vulnerability	Reference
Redundancy	✓	HW failure/Algorithm flaws	[39, 83, 189, 320]
Plausibility/Coherence Check	✓	HW failure/Attack	[59, 190, 321]
Data Security		Command injection	[100, 151]
Hardware Security Modules	✓	Command/Code injection	[27, 194, 228, 324]
Firewall/Intrusion Detection	✓	Command/Code Injection	[3, 45, 244, 253, 292]
Code Signing/Trusted Execution Environment	✓	Code injection	[69, 244]

2.4.3 Secondary Effects of Countermeasures on Safety, Security, and Performance

An autonomous car is a system in which safety is especially critical, as any malfunction could result in grave monetary damage or personal injury. Its design therefore involves careful consideration of different mechanisms to improve the security of the system. However, we note that adding these countermeasures may cause secondary effects, as repairing one flaw may cause a unfortunate cascade of further repairs.

For example, adding data encryption or authentication improves the security of a system, and should improve safety by preventing attacker-induced unsafe behavior. However, the added time to secure data degrades performance, and may delay safety-critical events.

Even firewalls, which filter communications and should prevent hacker-generated communications, will still apply a certain delay to communications. As any data protection or filtering will involve a delay, there will be an adverse affect on performance, and an ultimately unknown effect on safety.

For our safety countermeasures, a coherence check may prevent the impact of an attacker if it detects an incoherence between the injected and correct data, but only if the attacker cannot access both buses easily. It may be therefore helpful to secure the data with different encryption algorithms and keys to prevent an attacker from easily accessing both sets of data. Furthermore, the delay due to the coherence check may affect performance.

Failsafe modes can engage when the system detects a safety problem, such as hardware failure, or a security issue, such as an attack. While they are intended to improve the safety of the system, their effect also depends on their implementation, as the degraded mode might involve removal of certain security protocols, making the system ultimately less secure.

Many added features for connected vehicles should improve safety, such as automatic braking, V2X systems that can signal if a car in front is braking, etc. However, this added connectivity has adversely affected the security of these systems, by adding new avenues for attack. No hacks could be carried out on a completely isolated system.

By taking into account the limitations of sensors (especially due to environmental conditions), we realize that an autonomous car cannot depend on solely one sensor to perceive the world around it. Providing additional sensors improves our perception algorithm, but receiving and processing all the data, and then calculating coherences, occupies additional execution time. In the same manner, other safety checks such as monitoring or watchdog timers, also require additional hardware or software, and may impact performance [280].

Table 2.3 summarizes the impacts that the countermeasures have on the system properties of safety, security, and performance, whether positive, negative, or unknown. We note that any negative impact on security or performance may then lead to a negative impact on safety.

Thus, the overall effect on the system of adding a single countermeasure is often unclear. This interplay between safety and security demonstrates the complications in adding mechanisms to improve safety and security. The consequence that adding one countermeasure to ensure one requirement may have lead to the system violating another requirement supports the need for a methodology involving iterations of modeling and verification steps until the system is verified and all requirements are met.

Table 2.3: Impact on Safety, Security, and Performance of Countermeasures

Countermeasure	Safety	Security	Performance
Redundancy	+	?	-
Data Security	+/-/?	+	-
Failsafe Mode	+	-/?	?
System Monitoring/Watchdog	+	?	-/?
Automated Security (braking, V2X)	+	-	-

2.4.4 Conclusion

To ensure the safe and secure comportment of embedded systems, especially autonomous vehicles, various safety and security countermeasures will be modeled and studied in the rest of this thesis. Some solutions involve architectural additions, such as Hardware Security Modules and Firewalls, which can be purchased and integrated into a system. Others, like security protocols, plausibility checks and coherence checks, are implemented in software. As adding these countermeasures involves an increase in the monetary cost of the system, computation time, and board area, they should be carefully evaluated and added only where necessary.

2.5 Design Process Requirements

This chapter described many of the safety and security issues that autonomous vehicles face, and possible methods to counter them. Using a Model-Driven Methodology, we should ideally be able to verify that our system will not exhibit the undesired insecure or unsafe behaviors, and also check the effect of the countermeasures added. This process should thus result in the design of a safe and secure system, minimizing the flaws on the final system, which may involve costly modifications or patches (if they can be fixed at all).

The ideal methodology would be able to model all of these attacks and countermeasures presented in this chapter, and check how a system resisted attacks and faults by safety and security verification. As discussed by [276], there exist instances where safety and security conflict, and therefore they recommended a single framework to analyze both at the same time.

2.5.1 Methodology Capabilities

The full design of a system occurs in multiple phases at multiple levels of abstraction [262], as real-world embedded systems can be too complex to design all at once [278, 283]. Before a system can be designed, it is necessary to resolve the requirements of the system, to consider which of the aforementioned attacks and environmental factors it should be able to resist. During the design of embedded systems, both the hardware and software need to be designed, especially as we need to evaluate the placement and impact of the countermeasures which can be hardware or software-based [143, 313].

2.5.2 Properties to Verify

To check that a system will not enter any of the undesired states, safety and security verification should be performed. The undesired insecure and unsafe states from our taxonomy are summarized as the formal properties that can be checked. Safe system function can be more precisely expressed as the verifiable safety properties of deadlock, reachability, liveness, and other safety formula [28]. These properties can be expressed formally and checked with a model-checker such as UPPAAL or with reachability graphs [15].

Deadlocks occur when no further actions can be taken. For example, they can occur if all functions are blocked waiting for a communication or event from another function. A deadlock situation is unsafe as the system can no longer process input, and will fail to prevent the system from entering unsafe situations.

Reachability of a state or condition determines if that state or condition is present in at least one execution path of the system. Ensuring the main system functions are reachable ensures that the model is correctly designed.

Liveness of a condition checks if the condition will eventually hold in all execution paths, or if after one event occurs, then another event will always occur. For example, if a critical system error is detected, then the system should warn the user and enter failsafe mode. This property can also be expressed as 'Leads to' in UPPAAL.

System function can also be checked formally with **custom safety formula** expressing that a property to be always true, true for certain states, or for a property to be true for all states in an execution path.

To ensure that a system performs consistently and avoids the timing violations that delay response to critical events, **performance** should be verified. The usage (or load) on hardware components should be checked to determine if one component is not executing too many tasks. Timing properties should be checked to determine if events will occur within certain deadlines.

2.5.3 Security Properties

As previously described, the insecure system states fall into the categories *Confidentiality*, *Authenticity/Integrity*, *Availability*, and *Non-Repudiation*. We study the 3 violations which can prevent safe system function: Confidentiality, Authenticity, Availability, and Access Control, and define the properties to verify, as based on the definitions in the EVITA project [274].

Confidentiality is a property regarding whether certain data within the system can be recovered by an attacker, or unauthorized individual. Data is proved confidential if the attacker can recover the encrypted form but cannot decrypt it, for example.

Authenticity can be divided into two properties. **Integrity**, also called *weak authenticity* is a property regarding whether certain data within a system can be modified by an attacker or unauthorized individual. A communication fulfills the property of integrity if it has not been changed during distribution without the receiver detecting the modification.

Like integrity, **strong authenticity** is a property related to communications. Where weak authenticity only determines if a message has been modified by an attacker, strong authenticity ensures that messages being received in a certain communication exchange must have been sent in that exchange. For example, if an

attacker recovers and replays a message, then that communication satisfies the property of Integrity but not Strong Authenticity.

Availability refers to the ability of the system to offer services when requested by authorized users [339], and possibly within a set time frame.

Access Control refers to the ability of only authorized entities to access data or perform certain actions. It can be related to both Confidentiality and Authenticity, as attacks should not be able to access confidential data, and should not be able to perform If an attacker can modify the code of a system and modify behavior of certain components, then access control should prevent these hacked internal components from performing malicious actions.

These security properties may be evaluated formally with various tools or with simulations and analysis methods, as described in the next section.

2.5.4 Conclusion

In summary, to ensure the safe function of an autonomous vehicle, and ensure it is able to resist both attacks and adverse environmental conditions, we require a design methodology capable of modeling the system at multiple stages (preliminary analysis, architecture/mapping, and detailed software design), and evaluating the safety and security of the system.

In the next section, we present design methodologies and tools for modeling, analysis, and/or formal verification in our efforts to find one which fulfills all of these requirements, or to find how to best adapt an existing methodology to better address all of these needs.

Chapter 3

Related Work

"For me context is the key - from that comes the understanding of everything." – Kenneth Noland

---

In the previous chapter, we described the potential causes for an autonomous vehicle to enter unsafe situations risking personal injury, property damage, or disclosure of personal information, and summarized the proposed countermeasures. To evaluate that our system will not enter the unsafe or insecure states, we propose that a systematic design methodology could be used to model and then evaluate our system. This design methodology for safety and security-critical embedded systems should support a range of analysis, design, and verification capabilities. The capabilities can be summarized as follows:

- Modeling Phases
  - Analysis (Requirements, Failures, Attacks)
  - Architecture/Mapping
  - Software Design
- Modeling of Countermeasures
  - Hardware-based Countermeasures (HSM, Firewall)
  - Software-based Countermeasures (Plausibility check, Coherence Check, Security Protocols)
- Verification
  - Safety (Deadlocks, Reachability, Liveness, Performance)
  - Security (Confidentiality, Authenticity, Availability)

One methodology which supports most of the capabilities listed is the SysML-Sec Methodology. SysML-Sec was developed in my lab, Lab System-On-Chip of Telecom ParisTech, for the safe and secure design of embedded systems [13]. SysML-Sec is an extension of the UML Profile SysML, with some adaptations or additions to the supported diagrams. These changes are described in greater detail in Chapter 4.

UML is widely accepted as the modeling standard for software design, and easily understandable or known for most developers [106, 132]. UML diagrams span a range of usages, and can be customized by profiles for more precise needs [112]. For example, SysML uses a subset of UML and then adds extensions, so that it is more adapted for Model-Based Systems Engineering [80, 242]. SysML, for example, adds the concepts of Requirements and Parametric Diagrams which are further customized for SysML-Sec.

The entire SysML-Sec methodology, both modeling and verification, is implemented in the toolkit TTool [15]. Certain models can be verified informally with simulation, or formally with the safety verifier UPPAAL [29], and with the security verifier ProVerif [35] [252]. Models are automatically translated into their equivalence in the formal verification languages by the toolkit. UPPAAL, created for verification of real-time systems, uses timed automata to check properties such as presence of deadlocks, reachability of states/conditions, liveness of states/conditions, and conditions holding for all execution paths or all states within an execution path. ProVerif allows for the verification of the security properties of Confidentiality and Strong/Weak Authenticity of data, and also the reachability of all states within a security protocol to determine if it can correctly execute.

Before the start of my thesis, SysML-Sec involved 3 main phases: Analysis, HW/SW Partitioning, and Software Design. In the Analysis Phase, Attack Trees and Requirements diagrams are generated together as the designer considers the needs of the system and attacks it may face, with simulation and formal analysis to analyze if attacks are possible to carry out. Next, in the HW/SW Partitioning phase, the architecture and high-level function are modeled to determine a mapping and partitioning, supported by simulation of the mapping and UPPAAL verification of the functional modeling. Finally, the detailed functions are implemented in the Software Design phase, which can be verified with simulation, and formally in terms of security with ProVerif, and safety with UPPAAL. However, it does not support all of the previously-described needs, especially security modeling and verification during selection of a mapping/architecture, attacker modeling within system models, and latency measurements.

In this section, we review the different approaches to designing systems. We start by examining the main high-level approaches, the Agile vs Waterfall methods. Next, we examine the various tools and methodologies addressing each state of the design process, and the different verification and validation tools it supports. Examining the related works helps determine if another toolkit is more suitable for the design of safe and secure embedded systems, or it can instead provide insights for how the SysML-Sec Methodology can be adapted.

3.1 Software Development approaches

Software development methodologies can be generally classified into the Agile method or Waterfall method, with other methodologies variations and enhancements. Agile methods involve designing a small part of the system, testing, and then slowly adding to the system until it is complete, while the Waterfall-type methods involve systematic steps of analysis, design, and then testing. We describe these two approaches to discuss which software development tools and approaches can be applied to embedded system, and also to describe other characteristics of embedded systems design methods. This section describes the advantages and disadvantages of both the Agile and Waterfall methodology, and which is more suited for the design of embedded systems.

3.1.1 Agile

Agile Software Development relies on flexibility, where software is developed incrementally with design, coding, and testing cycles [158]. It has been commonly adopted by software companies as it allows for delivery of products more rapidly and easy evolution of requirements during the design process.

Scrum and Extreme Programming (XP) are adaptations, where Scrum relies on 'sprints', or phases where a certain amount of tasks are divided up, and XP relies on Test-Driven Development and Pair programming, where unit tests are first developed, after which the minimal amount of code to pass the current unit tests, and the process is continually repeated until the full program is developed. These methods involve constant meetings and discussions between developers and managers.

However, while some works have suggested that Agile methods could be applied for embedded system design [2, 129], hardware cannot be easily or inexpensively changed like software, and individual developers may only be experienced in certain domains [269]. To change the hardware components of a large system on each iteration would be unrealistic and expensive.

3.1.2 Waterfall/V Life Cycle

The Waterfall method involves step-by-step design, starting with [273]. The V Life Cycle was later proposed as an adaptation of the Waterfall model to better reflect the connections between the design and verification steps [270].

In efforts to make the Waterfall method more flexible and more rapid, the Incremental Model involved repeated iterations of the design, testing, and implementation phases, with client feedback after each iteration [84]. The Spiral Life Cycle [38], using ideas the waterfall, evolutionary prototyping and incremental methods, involves iteraions of 4 phases, determining requirements, risk analysis, development and testing, and then planning. The system is then constantly improved over each cycle.

3.2 Model Driven Methodologies and Toolkits

However, despite the popularity of Agile for software development [81], for embedded system design, we use a Waterfall/V Life Cycle-based methodology with Model-Driven Engineering as a logical progression is more suitable for our purposes. Therefore, the methodologies we examine are generally based on a step-by-step methodology or cover a single step or design phase.

There exist many model-driven methodologies for systems engineering. Each of them focuses on different phases, intended applications, and verification properties. This section describes these methods and their supporting tools, their capabilities relevant to the design, and the similarities and differences with our approach.

3.2.1 Frameworks for Analysis

Before a system can be designed, the Analysis phase determines the requirements of the system, and possible attacks and failures must be decided. It is important to determine exactly what a system must do,

and what risks should be prevented. Various methods and diagrams have been developed for this purpose.

3.2.1.1 Combined Safety and Security Analysis

Many works have discussed methods for safety and security-critical systems. However, they tend to focus on determining the risks a system faces, rather than how to design a suitable architecture and system. To consider safety and security at the same time, many methodologies adapted safety analysis models to analyze security at the same time.

SAHARA [210] describes how to perform STRIDE security analysis [221] with Hazard Analysis and Risk Assessment [160] separately, and identifies hazards caused by safety and security threats. The analysis involves completion of Safety Hazard and Security Risks spreadsheets, which details the attacks/hazards, the risk level, and the corresponding safety goals. Similarly, STPA-Sec [347] also adapted a safety methodology Systems Theoretic Process Analysis (STPA) for joint safety-security analysis. The method determines the unacceptable losses that a system may face, and then the possible vulnerabilities or faults that caused that hazard. With the analysis in mind, the requirements and countermeasures can be decided. The methodology does not offer a supporting toolkit, but can instead to applied to text-based and graphical models for analysis. These works perform analysis as a guideline for design, and do not use any formal verification to analyze the probabilities or possibilities of the occurrence of hazards.

Failure Mode, Vulnerabilities and Effect Analysis (FMVEA) [285] examines both failures modes and threat modes of each component along with their effects, and determines the ultimate severity and probability of a problem occurring. It follows a set of steps, where each component is analyzed to determine its failure modes and threat modes. Each failure or threat is then analyzed to determine its effect, severity, causes, and probabilities. It involves filling out a spreadsheet for each device and threat/attack. This technique is also subjective, and the probability values assigned are based on the user's experience and estimations.

[264, 265] demonstrate how their method CHASSIS considers Safety and Security together using Use Case, Misuse Case, and Failure Sequence diagrams in a common model. These diagrams can be graphical or textual. Safety and security hazards in the form of misuse cases are developed, and then trade-off analysis unifies all requirements and identifies when safety and security conflict. After drawing the misuse case diagrams, mitigations to prevent these issues are listed, and the list of mitigations can be analyzed to determine if any conflict, and generate a complete set addressing all safety and security issues. Finally, CHASSIS concludes by generating hazard analysis tables. While these techniques targeting the requirements and analysis phase offer a detailed approach to considering threats against safety and security, they are not yet automated.

Other methods are based on graphical models, which help support quantitative analysis. [256] used Boolean Logic Driven Markov Processes to model safety and security. BDMPs are similar to fault and attack trees, as they refine a root undesired event into more detailed causes, and they also allow for the modeling of sequences and reactions. The combined safety and security BDMP includes both safety nodes such as random or activated failures, and security nodes such as attacker actions, and timed and instantaneous security events. Random failures can lead to different attacker actions, and attacker actions can provoke failures. Their toolkit accordingly provides quantitative calculations such as probability, average time to attack, etc.

[299] added possible attacks to Component Fault Trees (CFT), and calculated the combined probability of

failure or difficulty of attack. Component Fault trees are similar to Fault trees, but they model components explicitly with all possible faults within each component. They build a CFT, and then considers where security problems could arise due to an attacker. Safety faults which can be caused by an attacker are extended with the details of the attack. Like with fault trees, their method also allows the calculation of minimum cut sets (MCS) (minimal set of events which can provoke a failure), and each MCS's probability of occurrence, required expertise of attacker etc.

3.2.1.2 Safety Analysis

HAZard and OPerability (HAZOP) is a method of determining possible system components which could pose hazards [178], originating from the chemical industry in the 1970s, but is now used for computer systems as well. It suggests that a team of employees consider each system element, its operation conditions, the possible conditions which could be result in injuries or damages, and thereby add safeguards. As this technique involves filling out worksheets, it is completely text-based, and does not include quantitative analysis.

[86] presented a framework to analyze the safety risks faced by autonomous and connected cars specifically. Their technique involves generating a table of attacks along estimated numerical values of time of attack, level of attacker expertise, and attacker motivation, and the ultimate attack impact, motivation, and ease of attack are calculated based on a mathematical formula. The resulting attacks are displayed on a threat matrix, showing ease of attack versus attack impact. Their work helps determine which attacks to prioritize preventing, and is partially based on quantitative measurements. However, it does not involve true formal verification or calculation of probabilities of attack.

Fault trees [330] analyze how a single undesired state can occur due to logical combinations of system faults or events. They use graphical models to describe the causes of a top-level failure, which is represented by the root node of the fault tree. The occurrence of the root failure is described as logical statements regarding intermediate failures, gradually refined into detailed causes for the fault. Fault trees support the use of analyzers to determine logically if the root fault is logically possible, and the probability of occurrence of the root fault if each node is marked with its probability. They can be used to determine how reduce the possibility of occurrence of the root fault by determining which initial faults can be prevented.

3.2.1.3 Security Analysis

Many approaches help determine the security requirements in the first analysis phase.

Attack Defense Trees [187], extended from Attack Trees [286], analyze the possible attacks against a system, in conjunction with the defenses that the system may implement. Similar to fault trees, they instead break down a top-level attack into detailed attack steps. The supporting toolkit ADTool analyzes attack scenarios to determine the cost, probability, time, etc, required for a successful attack.

The Knowledge Acquisition in Automated Specifications approach's Security Extension aims to identify security requirements for software systems [325]. The methodology uses a goal-oriented framework and builds a model of the system, and then an anti-model which describes possible attacks on the system. Both models are incrementally developed: threat trees are derived from the anti-model and the system model adds security countermeasures to protect against the attacks described in the anti-model. Countermeasures and security risks are expressed logically, based on a list of threat and countermeasure patterns. It offers

the advantage of modeling requirements and attacks together, but does not calculate probability or other measures of attack success.

SecuriCAD by foreseeti supports threat modeling and risk management, and operates at a higher level of abstraction. It models the entire infrastructure as different components including clients, networks, datastore, etc [93]. It also models specific attacks (memory corruption, SQL injection, code injection) and defenses (firewalls, intrusion protection, encryption). The probabilistic simulations generate attack graphs, and estimate parameters such as time to attack, etc. While their system is especially suited to modeling enterprise systems, their tool may be used for preliminary analysis of connected devices as well. However, it is not apparent how SecuriCAD could be used to design embedded systems at a much lower level of abstraction.

Similarly, the Predictive, Probabilistic Cyber Security Modeling Language (CySeMoL) is an attack graph tool that can analyze the security of entreprise architectures [147]. The models include assets, or components that can be attacked, and defenses that can reduce the probability of a successful attack. Attackers take different actions to compromise different components. The probabilities of discovery of vulnerabilities, successful intrusion detection, and other attack and defense steps were based on consulting experts and real-world data on system vulnerabilities. From the attack graphs, their analyzer determines the probability of a successful attack depending on the amount of time available to the attacker. Their tool however focuses more on modeling a network and components such as datastore, operating systems, and etc.

3.2.1.4 Conclusion

SysML-Sec currently uses Fault Trees, Attack Trees, and Requirements Diagrams in the preliminary analysis phase. All of these presented analysis methods are important for considering needs before the start of a system, but do not support the more concrete design of the system itself. An advantage of our toolkit supporting both analysis and detailed system design is to link our analysis diagrams to system design diagrams, to be able to better track if the design models address the requirements/risks conceived in the analysis phase, or determine than a requirement/risk is not relevant and should be removed. In practice, requirements and system design often evolve together over iterations [237, 338].

While textual models and graphical models each have their own advantages and disadvantages [130], we chose to use graphical models for modeling as they are easier to understand and to learn to use [82]. Hierarchical tree structures provide a full overview of the system, which can express the underlying reasons for the attack more clearly [155].

As this thesis delves more in detail into the relationship between safety and security, we will discuss in future work if additional modeling diagrams combining fault and attack trees are necessary, as proposed by the works on BDMPs and Component Fault Trees. Requirements diagrams may already include description of safety and security needs together, but some of these suggestions on combining modeling safety and security risks may eventually be adopted for TTool.

3.2.2 Frameworks for the Design of Embedded Systems

Many works have been proposed for designing embedded systems to fulfill industrial standards.

The MontiSim framework provides a tool for the modeling of requirements and systems, supporting various simulation tools for different domains, including autonomous vehicles [126, 213]. However, they use Component and Connector models, and performs simulations based on a fixed hardware, focusing on the detailed software implementation and behavior, and they lack high-level design and formal verification capabilities.

Other toolkits are specialized for automotive systems, such as Medini, which supports safety analysis and design based on ISO26262. It supports the entire methodology, from analysis phase activities including hazard and risk analysis, HAZOP checklists, safety level determination, requirements diagrams, to architectural and system modeling in SysML. It also allows import and conversion to Rhapsody, Enterprise Architect, and Matlab/Simulink models. It supports simulation and probabilistic analysis of faults, but not security analysis [10].

EAST-ADL is an architecture description language specialized for automotive systems also using the V-Model methodology [37]. It supports the methodology proposed by ISO26262, modeling safety constraints, fault/failure modeling, and timing and safety analysis [58, 75]. Models can be transformed into Simulink or UPPAAL models for analysis [215]. Simulink models can then be simulated with a plug-in FMUSim that preserves EAST-ADL properties, allowing the measurement of latencies and various system values.

Mbeddr is a development environment for embedded software, with extensions so it can be customized for different domains [334] . It focuses on modeling the software instead of hardware. It supports modeling requirements and systems, and offers model-checking including simulation and formal verification capabilities. The C-code can be verified with CMBC and Spin-based model checking. However, mbeddr does not model or verify security properties.

Many other design methodologies handle the complete design flow of embedded systems, from analysis to prototype code generation. Metropolis supports formal verification, simulation, and synthesis [24]. It follows the Y-chart method describing the functional model, architectural model, and then mapping functions onto an architecture. The models describe communicating processes, which can be analyzed with formal verification, simulation, and synthesis.

The authors of [279] introduce an abstract design space exploration (DSE) framework, and its integration into design space exploration solvers. Their tool Generic Design Space Exploration, is intended to support DSE for any domain, and allows the use of different solvers for DSE. They allow the user to specify different metrics and constraints to find an optimal solution.

MAESTRO [271] models embedded firmware, with support for automatic design space exploration and code generation. It also supports evaluation of power consumption, timing, temperature, etc. The Koski design flow models multiprocessor system-on-chips in a UML profile with automated design space exploration [170]. The entire process includes requirement description, application and architectural modeling, architecture exploration, verification by simulation, and code generation.

Capella [259] relies on Arcadia, a comprehensive model-based engineering method. It is intended to check the feasibility of customer requirements, called *needs*, for very large systems. Capella provides architecture diagrams allocating functions to components, and advanced mechanisms to model bit-precise data structures. Capella is however more business focused, and lacks formal verification capabilities.

Sesame [95] proposes modeling and simulation features at several abstraction levels for Multiprocessor System-on-Chip architectures. Pre-existing virtual components are combined to form a complex hardware

architecture. Models' semantics vary according to the levels of abstraction, ranging from Kahn process networks (KPN [169]) to data flow for model refinement, and to discrete events for simulation. Currently, Sesame is limited to the allocation of processing resources to application processes. It models neither memory mapping nor the choice of the communication architecture.

The ARTEMIS [258] project originates from heterogeneous platforms in the context of research on multimedia applications in particular. It is strongly based on the Y-chart approach [177]. Application and architecture are clearly separated: the application produces an event trace at simulation time, which is then read in by the architecture model. However, behavior depending on timers and interrupts cannot be taken into account.

MARTE [331] models communications, applications, and architecture. However, it intrinsically lacks a separation between control and message exchange. However, even if the UML profile for MARTE adds capabilities to model Real Time and Embedded Systems, it does not specifically support architectural exploration. Other works based on UML/MARTE, such as Gaspard2 [115], are dedicated to both hardware and software synthesis, relying on a refinement process based on user interaction to progressively lower the level of abstraction of input models. However, such a refinement does not completely separate the application (software synthesis) or architecture (hardware synthesis) models from communication.

Design Patterns are solutions to common design problems that can be applied to all instances of similar problems [26]. While design patterns were first proposed for software design, [87] proposed how to use design patterns in embedded systems. Design patterns can also be used with UML. [310] uses ACCORD/UML with Component Based System Engineering for prototyping embedded systems, and allows for modeling of the application and environment, including simulation. As it was intended for developers who are not software experts, their toolkit offers pre-generated design patterns with automatic code generation. Their verification consists of prototyping and testing without formal verification.

Similiarly, Problem Frames [164] describe common software development problems, and can be used to generate requirements. Problem frames model the context of the problem, and then the requirements that should be generated to solve the problem. Standard solutions to each problem are described as 'Solution Structures', which can then by applied for all situations matching the problem context. Problem frames, first used for functional problems, have been extended for security and performance-related problems to generate corresponding requirements [4, 138].

Synopses System On Chip Verification offers various tools, like simulation, verification, virtual prototyping, emulation, and debugging [303]. For example they allow for performance verification to check latencies and system bottlenecks, and simulation using a variety of tools including SystemC, Matlab, etc. Formal verification can check many implementation details, such as of registers, bus checks, etc. Many of their tools, however, operate at a very detailed level, such as simulations at the transistor level, and security verification of locations of data storage.

3.2.2.1 Security in HW/SW Partitioning

Of the works which investigate security during the architecture and mapping phase, [104] relies on Architecture Analysis and Design Language (AADL) models to consider architectural mapping during security verification. The authors note that a system must be secure on multiple levels: software applications must exchange data in a secure manner, and also execute on a secure memory space and communicate over a

secure channel. Our work, however, investigates protections against an external attacker instead of access control.

Another approach performs Design Space Exploration using Integer Linear Programming on a vehicular network protecting against replay and masquerade attacks, to map nodes to an architecture [205]. The authors use MACs, a counter, and key distribution to ensure the strong authenticity of CAN, TDMA, and V2V communications. They examine the needed MAC size for low probability of the attacker guessing the message, and ensure the timing constraints for message arrivals are met. However, their work targets automotive systems and network communications, instead of general embedded systems, though the authors have developed other Design Space Exploration tools for general System-On-Chip based on timing, power, and area constraints, but not security [239, 351].

[136] enhances Design Space Exploration with the ability to map security tasks in a real time multicore system with the algorithm HYDRA. Their work assumes an attacker who can intercept communications, forge messages, and prevent the availability of services. To impede the attack, security tasks must be performed periodically. Security tasks are abstracted to consider only that they must execute within a set deadline to maintain the security of the system, and not the exact mechanisms for security. Furthermore, their work does not consider hardware-based security countermeasures.

[166] also considered how to secure communications in embedded systems, with encryption performed in software or on FPGA. They considered how to ensure only the confidentiality of their internal messages, with a single encryption algorithm AES. They consider all possible mappings with static and reconfigurable FPGA, and determine if the system meets timing constraints. Their work is focused on scheduling and constraint satisfaction, and not on modeling of architectures or other encryption algorithms.

Likewise, the Simple Modeling Language for Embedded Systems (SMOLES) [307] was enhanced with a Security Analysis Language [89]. SMOLES models systems as a set of components with input and output ports, and tasks are mapped onto the hardware platform. Models can be verified for schedulability, timing, including latencies, and safety properties with UPPAAL. The addition of security algorithms can secure communications across partitions, and also models the attacker capabilities in terms of the size of keys that can be cracked. Their analysis tool can then analyze the fulfillment of integrity and secrecy requirements. However, their work does not model architectural countermeasures, or security protocols beyond a set of encryption algorithms.

3.2.2.2 Conclusion

The HW/SW Partitioning tools help the designer decide on a mapping and architecture with analysis and simulation engines determining execution time, architectural load, functionality, security, and safety of the system. Some also support automatically choose the best mapping with Design Space Exploration. While there exist a range of tools which can check the performance and functionality of a mapping, the few which handle security do not model attacker behavior or support evaluation of the hardware countermeasures proposed in the previous section. This review of the available toolkits supporting the HW/SW partitioning phase demonstrates that no existing tool will support the security needs proposed in Section 2.5: ability to verify security properties of the system (mainly Confidentiality, Integrity, Availability), and ability to evaluate if countermeasures can help the system resist attacks. The lack of a toolkit supporting our needs for secure design of embedded systems drives the contributions described in this thesis.

3.2.3 Frameworks for Software Design

3.2.3.1 Safety and Security Modeling

Similar to SysML-Sec, [48, 49] consider safety and security issues, and translate their model of blocks with communications into a specification for formal verification to analyze if safety or security failures can occur. Their work uses the formal analyzer Alloy, and considers the status of components. The status involve safety failures such as missing data, erroneous data, or ill-timed data, or security failures due to an attack such as missing data, injected data, or erroneous data. The verifier can then formally check if the system will correctly function despite the failures or attacks on some of the components. Their work, however, considers the high-level view of how failures/attacks affect system function, but does not determine if the failures/attacks were possible based on the more detailed behavior of the system. Their approach also does not verify the safety and security properties listed in Section 2.5.3.

3.2.3.2 Security Modeling and Verification

[329] proposed modeling security in embedded systems with attack graphs to determine the probability that data assets could be compromised. Behaviors are modeled as time-dependant stochastic processes. Their toolkit checks for confidentiality and integrity of data across different designs, but does not model encryption mechanisms. While their approach is also UML-based, they focus on estimating probabilities of success for attacks instead of verifying security properties.

Another probabilistic approach uses state transition models to calculate mean time to failure of the system due to an attack, and is concerned with the security properties of Confidentiality, Integrity, and Availability [211]. The system is specified with parameters based on the attacker's capabilities and system's capability to resist attacks, such as the probability a system can detect an attack, probability that the, time to handle and attack, time an attacker can remain undetected, and etc. Their system, however, is represented with a state machine diagram and is primarily concerned with whether the system is in a hacked or unhacked state. Their work offers an approach to quantitative analysis of the ability of the system to resist intrusions, but does not explicitly model the system or countermeasures.

SecureUML enables the design and analysis of secure systems by adding mechanisms to model role-based access control [206]. Roles can be assigned to different users, and assets are given security rules. Authorization constraints are expressed in Object Constraint Language (OCL). SecureUML however does not model detailed system behavior or security protocols.

UMLSec [168] is a UML profile for expressing security concepts, such as encryption mechanisms and attack scenarios. It provides a modeling framework to define security properties of software components and of their composition within a UML framework. UMLSec considers the security properties of confidentiality, integrity, strong and weak authenticity, and access control concepts. It also features a rather complete framework addressing various stages of model-driven secure software engineering from the specification of security requirements to tests, including logic-based formal verification regarding the composition of software components. However, UMLSec does not take into account the HW/SW Partitioning phase necessary for the design of e.g. IoTs, nor safety verification.

SysML has also been extended to better model security. For example [240] proposed adding threat agents, vulnerabilities, and encryption to the model. The tool provides a list of vulnerable items, such as connec-

tions and ports, which the designer should then attempt to mitigate. However, their work does not discuss how to verify the security of the system. [195] similarly added security notations to SysML, including vulnerabilities and security properties, such as integrity. Based on the different components selected, the toolkit uses the Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) vulnerability database to determine which vulnerabilities exist in that model. Their inductive definition programming framework then analyzes the provided model, and check if the system respected the security properties. They can model the existence of an authentication method abstractly, but not track encrypted communications or keys, for example.

Secure MDD uses UML to model security critical applications, and performs automatic formal verification using the Karlsruhe Interactive Verifier (KIV). Security requirements are formalized with OCL Constraints, which can verify properties like if the attacker can recover certain data. Their tool also generates the Java code for the application [225].

The authors of [78] leveraged Attack Trees to analyze network security. Both system vulnerabilities and attacker capabilities are modeled and analyzed to determine the possible attacks a system may face. [148] also used a hierarchical attack representation model (HARM) to assess different moving target defenses, modeling both system vulnerabilities and attack graphs and trees. [117] described a similar framework modeling attacks and defenses in IoTs. These works described the system vulnerabilities along with the attacker actions, but they do not model security countermeasures or use formal verification.

The Software Architecture Modeling (SAM) framework [5] aims to bridge the gap between informal security requirements and their formal representation and verification. SAM uses formal and informal security techniques to accomplish defined goals and mitigate flaws. Their approach covers both the analysis and design phases. SAM relies on a well established toolkit - SMV - and considers a threat model.

ASE is a security modeling methodology following the steps of analysis, design, security modeling, and security verification [322]. They consider the security attributes confidentiality, integrity, availability, and accountability. Requirements are generated based on identified threats, and then countermeasures are generated based on the requirements. Security verification consists in ensuring that all countermeasures are added by checking links, but does not use formal verification to check the detailed security properties.

[42] demonstrated some of the advantages of Functional Block Design using Simulink vs UML. Functional Block Design offers more detailed control modeling, and supports more modes of computation. However, UML supports better requirements elucidation and verification capabilities. While the detailed implementation of certain algorithms in our system are also modeled in Simulink, starting with the high-level design and architectural modeling are more suited to UML.

[102] recognized that Matlab/Simulink is more suited for detailed dataflow modeling, while modeling languages such as UML and AADL are more suited for architecture modeling. Their tool, Fokus Embedded Systems Architect, generates code from both the UML and Matlab models. Classes are marked to indicate if their behavior is specified by previously-generated C code, UML code, or Simulink code, and certain functions can refer to Simulink functions. Their code generator then generates C code for each UML class or function accordingly. At this point, however, their work focuses only on code generation and not verification.

Other works suggest more high-level methodologies, some with a suggested set of countermeasures. A well-known methodology, Common Criteria suggests the full process for ensuring security, involving identifying threats, assessing risks, implementing countermeasures, and assurring the effectiveness of counter-

measures [1]. [267] discussed how Common Criteria can be used for embedded system design. They suggested countermeasures such as code signing, writing efficient code, minimizing ports, and intrusion detection systems that can wipe memory if an attack is detected.

[33] described how to use security design patterns, which can either protect authenticity and confidentiality of the system or data, or the availability of the system. After resources and attackers are identified, the relevant design pattern can be applied. Detailed policies relating to the pattern are then configured and validated.

As adding safety features to embedded systems affects non-functional properties, such as performance, reliability, and etc, [18] proposed that design patterns should also be marked with their side effect, or implications on these non-functional properties. [135] proposed a methodology and toolkit supporting adding Security and Dependency Design Patterns. By providing a repository of design patterns, they allow for reuse of their developed security solutions. These documented and proven Design Patterns can be useful in software design, and contains many of the countermeasures previously listed such as *Secure Communication* and *Error Detection/Correction*, but we start from a far more abstract level of design and verify the system at the end.

[305] offers various tools for supporting a Secure Software Development Lifecycle (SSDLC), such as threat modeling [302] and inspecting software [304]. Their tools, however, provide modeling only at the analysis phase, and then code analysis tools for inspecting the software. Similarly, Microsoft Security Development Lifecycle [222] is a software development process considering security at each design phase, similar to a Waterfall lifecycle. It involves analysis of security and privacy requirements, system design, implementation, code verification using dynamic and static analysis and fuzz testing, and finally release. It recommends a set of security practices, such as using approved tools and conducting testing and security reviews. It is also supported by a threat modeling and code analyzer tools. Their verification operates on the code itself, and not on models with mathematical formal verification tools.

3.2.3.3 Safety Modeling and Verification

Some works have used mathematical formula to determine the safety of the system in an environment. To evaluate the safety of navigation algorithms of an autonomous vehicle in different traffic [7, 8] used Markov Chains. Vehicle behavior is specified as an equation describing its position over time, and the set of reachable locations is calculated. Their model however, is entirely mathematical and calculates the probability of a crash at each time step. While these works evaluate the safety of autonomous driving in greater detail, they assess only the navigation within traffic and not other system properties.

To verify the safety of hybrid systems, [261] proved that the system would never enter unsafe regions with barrier certificates. Their work, however, is mathematical instead of graphical, and focuses on analyzing trajectories. They expressed the entire relevant system behavior as a system of mathematical expressions, which they then verified would never violate certain constraints. While these works precisely analyze system behavior, they depend on detailed mathematical models, and only analyze their external behavior.

Other works allow for a higher-level verification of the system, with analysis of properties like reachability of states and other verifiable expressions. [133] uses the language S#, based on C#, for simulation and formal verification of safety-critical systems. It allows the modeling of component faults and the environment of the system. Their safety analysis can generate minimum critical fault sets for hazards, check for

reachability and deadlocks, and can check if formula or invariants are satisfied. These works, however, use text-based instead of graphical models like SysML-Sec.

[167] presented how to check safety requirements within a system modeled in Simulink, and determines if the system can fulfill that requirement despite multiple faults. After the system is modeled in Simulink, including explicitly modeling the possibility of faults of components, the model is translated into SCADE. Safety properties are expressed in Lustre, and the design verifier can determine if the property holds or not.

Other works have used formal verifiers like UPPAAL to analyze their models. [124] created a tool ECPS verifier to translate AADL models into timed automata that could be verified with UPPAAL. Possible faults are modeled with Fault trees, then integrated into an AADL model. Their work is similar to our approach to safety verification, as they also use an automatic model translator and verify properties of liveness, reachability, lack of deadlocks, but they use UPPAAL instead of simulations to evaluate latencies.

3.2.3.4 Conclusion

Software Design modeling tools, whether in UML, AADL, Matlab, or textual specification languages, support a variety of different verification mechanisms. Many offer functional verification or property checks for safety verification, and others offer formal analysis of protocols for security verification. Others check the code itself for safety and security flaws. Some of these works offer capabilities are not present in SysML-Sec/TTool, such as automatically generating vulnerabilities from available databases based on the components selected, or detailed mathematical functional modeling. However, they lack support for connecting the architecture/mapping models and the detailed software design models, which is important in determining that abstract HW/SW Partitioning models performed the correct abstractions, and also minimizes building an extra set of models manually after the selection of an architecture/maping.

3.2.4 Conclusion

Table 3.1 shows a summary of all related methodologies and toolkits, with their modeling and verification capabilities to summarize the differences with our work.

All of these other design methodologies and toolkits offer modeling and verification for various purposes with a range of approaches, but none of them support the full design process needed for the design of safe and secure embedded systems, such as Vedecom's Autonomous Vehicle, as previously described in Section 2.5. To re-iterate, we stated that the methodology should support modeling for preliminary analysis, architecture, and more detailed software design, and allow for the modeling of the relevant countermeasures studied. Furthermore, the toolkit should provide verification tools to analyze the safety and security requirements.

While the SysML-Sec Methodology lacks the ability to address some of the needs, such as better timing analysis and modeling in security, it addressed enough that we could adapt and enhance it to take all of our modeling and verification needs into account. The rest of the thesis discusses how we added the main missing capabilities. Some of the other interesting capabilities that TTool/SysML-Sec does not support, including probabilistic analysis of attacks, offer potential directions of future work described in greater detail in Section 8.2.

Table 3.1: Comparison of Related Works

Toolkit/Method	Phase			Supported Verification				
	Analysis	Part.	Soft.	Func.	Safety	Security	Perf.	Formal?
Analysis-Only Tools								
CHASSIS [264]	✓				✓	✓		
BLDMP [256]	✓				✓	✓		✓
CFT [299]	✓				✓	✓		✓
Attack Defense Trees [187]	✓					✓		✓
Design Tools								
TTool/SysML-Sec [13, 15]	✓	✓	✓	✓	✓	✓	✓	✓
UMLSec [168]			✓	✓	✓			✓
SecureUML [206]		✓		✓	✓			
Lin2015 [205]		✓		✓		✓		
AADL [104]		✓		✓		✓		
Metropolis [24]		✓	✓	✓	✓		✓	✓
MAESTRO [271]		✓	✓	✓	✓		✓	
Koski [170]	✓	✓	✓	✓	✓		✓	
Brunel2015 [48]	✓		✓	✓	✓	✓		
MontiSim [126]	✓		✓	✓			✓	
Medini [10]	✓		✓	✓	✓			
Matlab/Simulink [102]			✓	✓				
Arcadia [259]	✓		✓	✓	✓			
Sesame [95]	✓		✓	✓	✓			
EAST-ADL [58]	✓	✓	✓	✓	✓		✓	✓
Artemis [258]	✓		✓	✓	✓			
Design Patterns [87, 135]	✓		✓	✓	✓	✓		
Problem Frames [4, 138, 164]	✓	✓	✓	✓	✓	✓	✓	
SAM [5]	✓					✓		
mbeddr [334]	✓		✓	✓				✓
SysML-IDP [195]	✓		✓		✓			✓
MARTE [331]		✓	✓	✓			✓	
SMOLES-SAL [89]		✓		✓	✓	✓		✓
Attack Probabilities [211, 329]			✓			✓		

Chapter 4

Modeling Methodology

> "The good thing about bubbles and arrows, as opposed to programs, is that they never crash."
> –Bertrand Meyer

4.1 Introduction

With the growing complexity of embedded systems, designers often rely on system design methodologies. Instead of starting directly with designing the system itself, many methodologies start with an analysis phase, to better conceive all of the needs of the system [185]. The most important and hardest task of designing software systems is sometimes considered to be determining the requirements of the system, as even clients may not be able to concretely express their needs in sufficient detail to design the system [47]. Ensuring that requirements are as correctly defined as possible in the beginning prevents re-work due to modifying the system to address added or changed requirements midway through the design process. In the USS Vincennes disaster, an engineer working on the plane tracking system had proposed adding displays that would better help users monitor plane trajectory (which could have prevented the disaster), but his supervisors denied his request since the Navy hadn't explicitly asked for such a display [301].

Based on the preliminary analysis, a system can be designed in iterations, starting at a high level of abstraction [113]. By starting at a high-level of abstraction and then gradually moving to lower levels of abstraction, the designer is not required to consider the entire complex system at once, and instead can start with a more manageable high-level model. Over iterations, details are added to the system, until either the system is ready to build and program by hand, or code can be automatically generated. Automatic generation of program code and VHDL synthesis from a specification is preferred, as design time is decreased, code can be optimized, and human errors are avoided [161, 227].

In certain real-time safety and security critical embedded systems, such as autonomous vehicles, various formal verifications and simulations can help ensure that the design meets the requirements stated in the analysis phase. Safety and security considerations must also be integrated into the design methodology, with additional modeling, through added diagrams or modeling elements, and formal verification techniques. Since no other toolkit and methodology is adapted for the design of safe and secure embedded

systems, as described in Chapter 3, we propose modifications to the SysML-Sec Methodology. The high-level overview of our methodology is shown in Figure 4-1. In this section, we describe the modeling steps in greater detail, with the associated models, and briefly describe the modifications to this methodology presented in the rest of this thesis. All of the modeling and automatic verification steps are supported by our toolkit, which then clearly displays the properties satisfied and not satisfied by the current model [235].

4.2 Overview

As with many other methodologies, our methodology recognizes that we need to determine what to design before we can start designing it. We also need to balance all of the different safety, security, functional, and timing needs, some of which can conflict as described in section 2.4.3. Our methodology starts with the Analysis phase, where we consider the needs of the system and possible issues it may face. We model the requirements of our system, regarding the functionality, safety, and security. To better determine requirements, we also model the potential attacks and faults that the system should protect again. Attack Trees describe the possible attacks the system may face, and Fault Trees describe how failures in the system may lead to possible losses. The detailed attack steps of an attack tree, for example, The requirements, faults, and attacks should be modeled together, as possible attacks and failures may lead us to develop new requirements. Countermeasures to attacks and faults, especially, can be directly translated into requirements.

After we have finished our analysis, we accordingly model the system on various levels of detail. The two design phases, Hardware/Software Partitioning and Software Design, take place in the two different design environments of TTool: DIPLODOCUS and AVATAR. Each design environment contains different types of models and supports different types of verifications. Throughout this thesis, HW/SW Partitioning models within TTool are referred to also as Mapping models or DIPLODOCUS models, and Software Design models are also referred to as AVATAR models. AVATAR models, for example, indicate the entire ensemble of block diagrams, state machine diagrams, etc for a single system.

Hardware/Software Partitioning models describe the architecture and high-level functional behavior, and they show which architectural component performs each function, also called a “mapping”. First models may be abstract and ignore details, and instead only describe the high-level properties. After the system is sufficiently modeled, simulation and formal verification determine if mapping-relevant requirements (in terms of safety, security, and performance) are satisfied. Based on the results of the verification, the partitioning models may be modified, and relevant safety and security countermeasures may be added. The countermeasures at this stage include redundant hardware, firewalls, and abstract data security operators. Countermeasures operating at a lower level of abstraction are added in the following phase, such as ones operating on exact values of data communications. Design elements and verification results can be linked back to the analysis phase diagrams, such as timing results linked to a requirement on timing, or confirmation that certain data is secure can be used to mark an attack step impossible. The security verification results also determine how attacker scenarios can be added explicitly to the diagrams.

Once the architecture and mapping have been determined, we model the detailed behavior of the system during the Software Design phase. Preliminary Software Design models can also be generated automatically from the HW/SW Partitioning Models. The algorithms and behavior of the system is modeled in greater detail. Formal verification then checks the system again to ensure it meets refined requirements. In Software Design, Verification Queries are written in pragma, which are notes on the modeling diagrams [16, 252]. Safety verification is performed with UPPAAL or the TTool Model Checker, and

Security Verification by ProVerif [13, 15, 16]. After verification, each pragma automatically marked verified, not verified, or cannot be analyzed [209, 252]. Based on the verification results, the system models can be refined, and additional countermeasures, such as exact security protocols, plausibility check, etc. Once the software is sufficiently designed and verified, our toolkit can automatically generate C code for prototyping, and test sequences.

In comparison with the last former version of the SysML-Sec Methodology, presented in [12], the new methodology shows the different modeling and verification steps in greater detail, and adds safety and security concepts. In the Analysis phase, Fault trees were added to the possible preliminary analysis diagrams. In the Design phases, the safety and security countermeasures (Redundancy, Firewall, Failsafe Mode, Security Protocols, etc) which can be added to reconsider a model after verification are explicitly shown. In the HW/SW Partitioning phase, security modeling and verification have been added, as well as the automatic model transformation to Software Design models.

The system is continually reconsidered or refined until all requirements are met and the system is modeled with all relevant details included. Throughout the process, modeling and verification elements are annotated with their related requirements to track the fulfilment of requirements, as we describe in the following sections.

A metamodel helps define a modeling language or UML profile, and the concepts and relationships between the structures within it [21, 341]. Their importance lies in their ability to help designers understand the language and detect inconsistencies within a language [231].

Figure 4-2 shows the metamodel of the different modeling diagrams. The metamodel is divided into the 3 phases, Analysis, HW/SW Partitioning, and Software Design. For example, as described, Analysis models consist of Requirements Diagrams, Attack Tree Diagrams, and Fault Tree Diagrams. Each Attack Tree Diagram then consists of the attacks, logical operators linking attacks, and countermeasures.

The links between modeling phases and diagrams are shown, such as the linking of requirements to Verification Queries and Software pragma, which express how requirements help a designer determine which properties should be verified. Countermeasures can be related to requirements in the analysis phase, and these countermeasures may then be present in the HW/SW Partitioning and Software Design Diagrams. More detailed metamodels of the diagrams will be presented in the rest of this thesis.

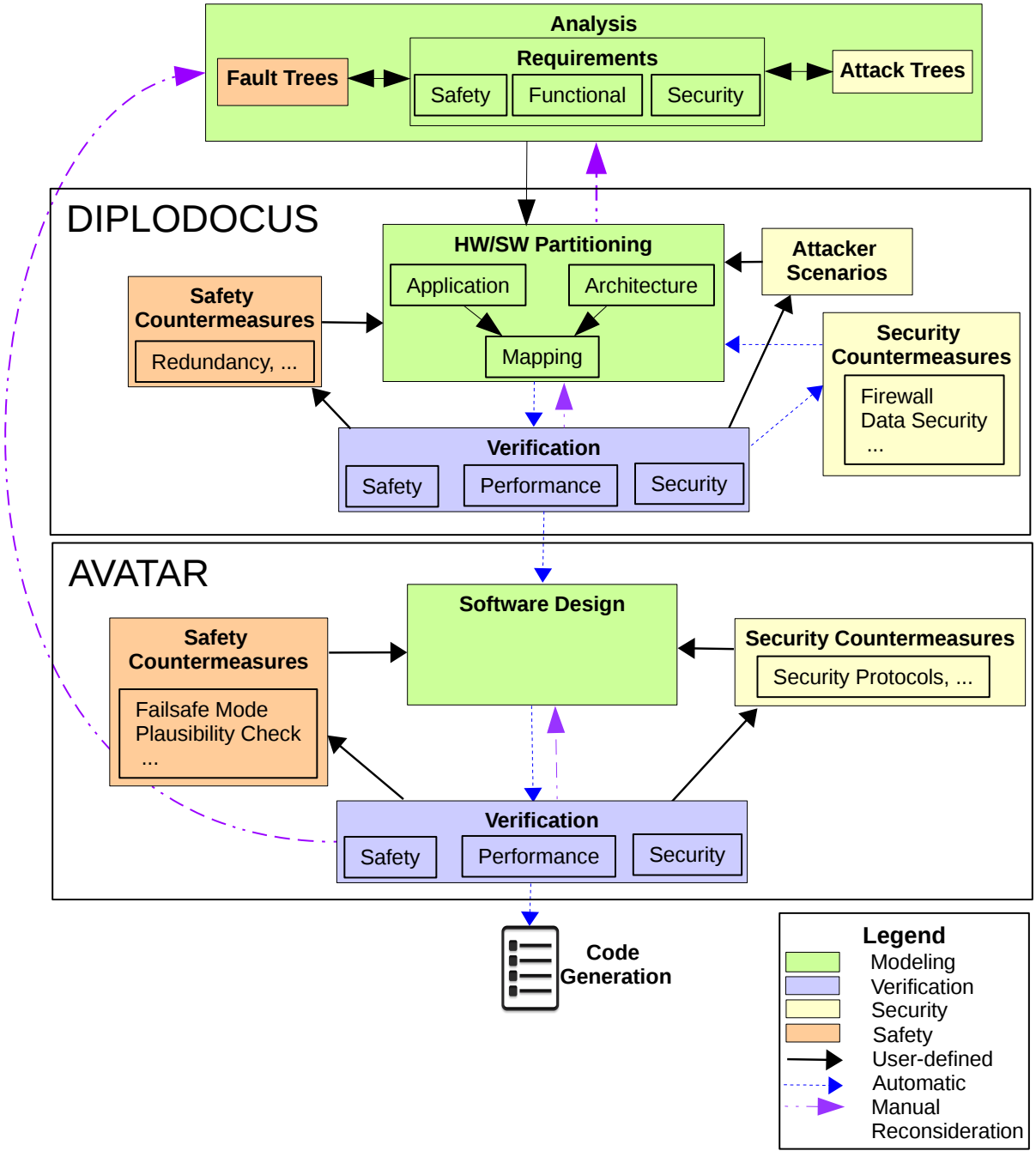


Figure 4-1: Overview of SysML-Sec Methodology for the Design of Safe and Secure Embedded Systems

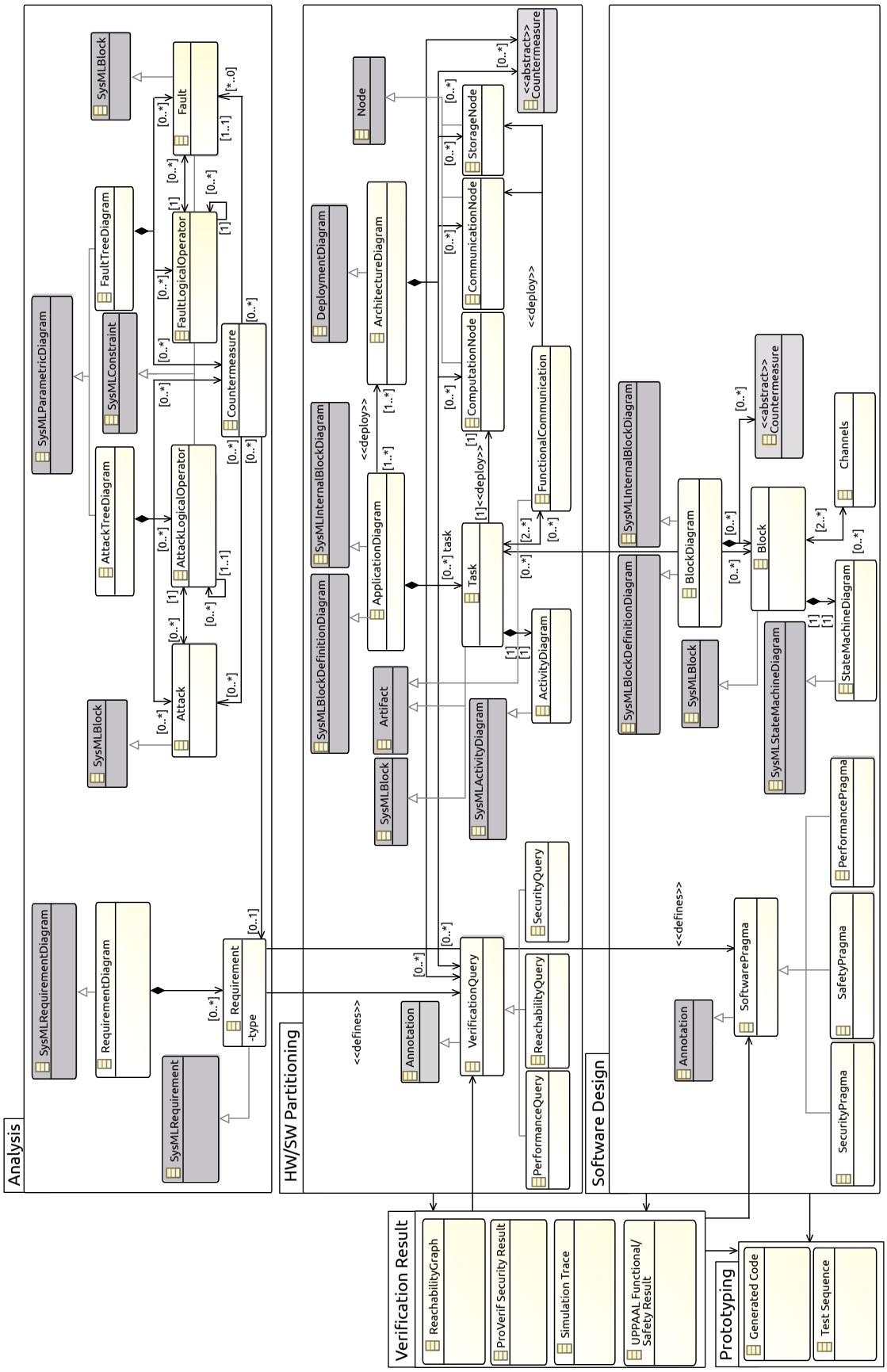


Figure 4-2: Metamodel of Diagrams for SysML-Sec Methodology

4.3 Analysis

The Analysis phase, involving specification of requirements and possible risks/attacks helps us abstractly consider our system, form a coherent list of and provides a framework for design. Models showing how faults and attacks can occur, which can be named anti-models or misuse cases, can help specification of requirements [296, 326].

4.3.1 Requirements

Safety Requirements describe properties to ensure the safe function of the system, such as the inability for the system to reach deadlocks/livelocks, error states, and delayed reactions to safety-critical events. Security Requirements describe which communications must not be recoverable or modifiable by the attacker. These requirements should address all of the issues previously listed in our taxonomy.

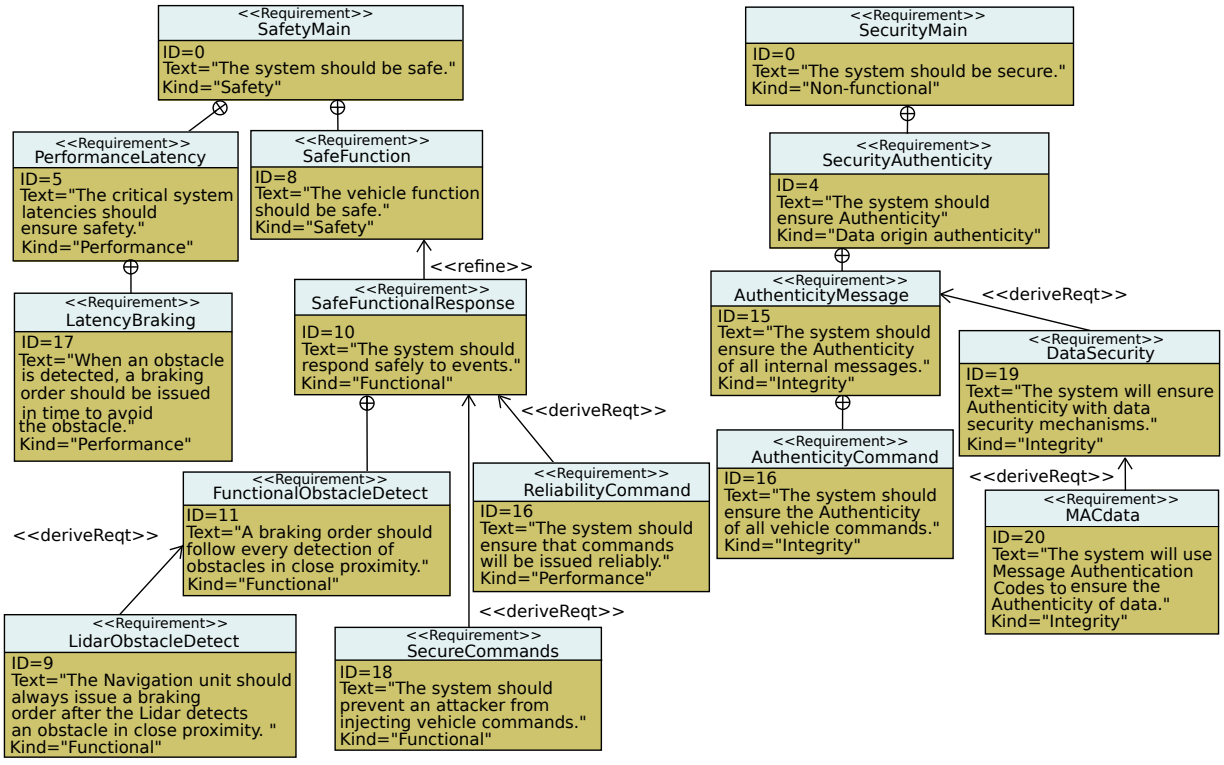


Figure 4-3: Refinement of Requirements for Vehicle Safety and Security

High-level requirements are then refined until they are implementable in a model, or a query to be verified. Figure 4-3 shows how the safety and security requirements are refined. For safety, the high-level requirement '*The system should be safe*' is refined to a safety requirement to brake when detecting an obstacle and a performance requirement on the timing of such braking orders. For security, the high-level requirement '*The system should be secure*' is eventually refined into a requirement for authenticity of vehicle commands and implementation detail to use Message Authentication codes to ensure authenticity. Safety and Security Requirements can be related, such as the 'SafeFunctionalResponse' requirement being dependent on the 'SecureCommands' requirement: more precisely, the ability of the system to respond to

events safety relies on an attacker not being able to inject vehicle commands.

In the Requirement Diagrams, the arrow marked '*derive Requirement*' links a requirement that contains implementation details. For example, to ensure Authenticity of communications, the derived requirement demands that the system implement some countermeasure. One way to ensure authenticity in communications is to use Message Authentication Codes which can indicate if the message as been altered, as we will describe in greater detail in section 5.3.4.

Taxonomy elements can be related to certain requirements to determine exactly which type of verification should be used to check for their satisfaction. For example, The requirement 'LatencyBraking' should be checked by a timing verification. 'LidarObstacleDetect' should be checked by functional verification, or more precisely a liveness property to check that the braking order always follows the detection of an obstacle too close to navigate around. These requirements then guide the specification of safety and security verification queries during the proceeding system design phases, to ensure that these requirements have been met [202], as discussed in the next chapters.

4.3.2 Attack Trees

Attack Trees are related to the Security Requirements, but detail the specific steps the attacker would take to realize an attack on the system [17]. Figure 4-4 shows one such attack to prevent braking in the case of obstacles. We consider that there are two main methods to prevent braking: either by preventing the system from detecting the obstacle, or by preventing transmission of the braking command to the car. Each individual attack step can be determined to be possible or impossible as the system is designed. To realize a root attack, a combination of individual steps must be possible so that the root attack is ultimately logically possible.

For example, one possible combination of attack steps would be to manipulate both the camera and lidar to accept false data, which would then prevent obstacle detection. If however it was impossible to manipulate the camera, then that attack path would be impossible due to the fact that the attack step '*manipulate sensors*' requires both manipulating sensors and the camera, as the system uses a combination of all sensors for obstacle detection. In noting that possible attacks may include jamming communications, a requirement '*The system must ensure communications are always available*' could be added to address this attack. Thus, the attack trees and requirements can evolve together. Countermeasures can also be added to disable an attack step, such as using an authentication method to prevent the forging of ECU commands (as stated in the requirements).

Based on countermeasures and estimations of attacker capabilities, attack steps can be marked possible or impossible, and a formal verifier then determines logically if the root attack is possible.

4.3.3 Fault Trees

Similar to Attack Trees, Fault trees represent the causes for a top level 'failure mode' event, which occur due to a logical combination of 'basic events', which are specific events that are not needed to be further decomposed [179]. These events are caused by system malfunctions or limitations, however, and not an attacker. Each basic failure event can be counter-acted with countermeasures, which as with attack trees, will disable the failure step. Failure events can also be marked with probabilities, so that the Fault Tree

formal analyzer can determine if the top level failure is possible, and if so, the probability of its occurrence. Based on the possible environmental factors and component failures, we also propose the combinations of faults that could cause the failure to detect or avoid an obstacle, as shown in Figure 4-5.

Countermeasures can also be added to prevent faults. For example, redundant hardware (buses and processors) can prevent the inability to receive data or calculate trajectories/obstacles.

4.3.4 Relationship between Analysis Phase Diagrams

Potential faults and attacks which cause system losses may motivate the designer to add requirements counteracting them, or the countermeasures listed in Attack and Fault Trees can be rephrased into requirements. Figure 4-6 shows how the braking failure Attack Trees and Fault Trees are linked to the requirements diagram. For example, the requirement to issue a braking order after the lidar detects an obstacle in close proximity can be violated due to a fuzzy or lossy communication channel, algorithm flaws, or an attacker who has jammed all inter-system communications. These failures/attacks can be in turn prevented with redundant communication buses, using multiple algorithms (as all algorithms demonstrating the same flaw is less likely), and anomaly detection schemes or firewalls.

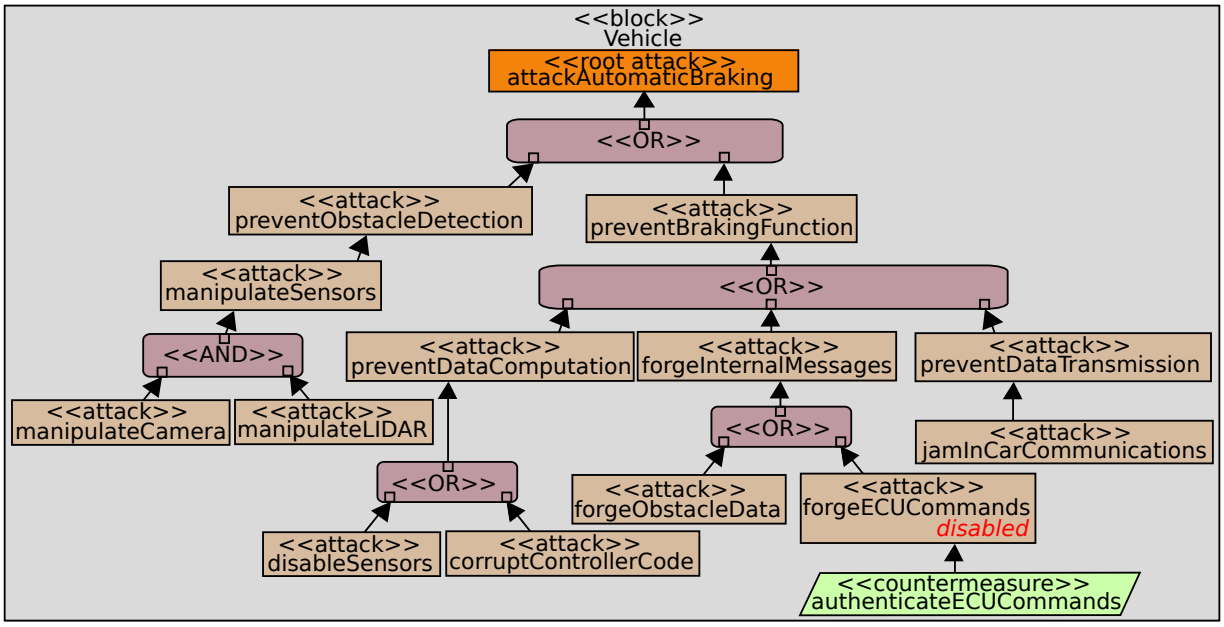


Figure 4-4: Attack Tree for Obstacle Detection Failure

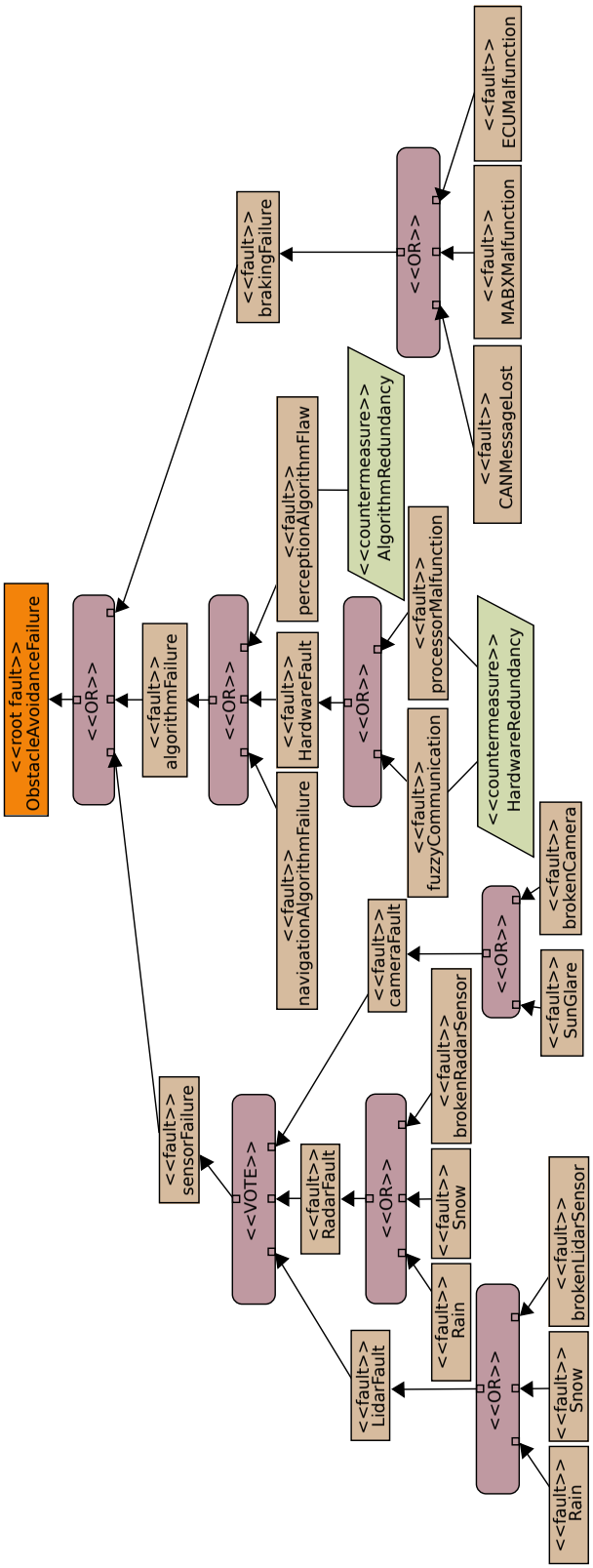


Figure 4-5: Fault Tree for Obstacle Detection Failure

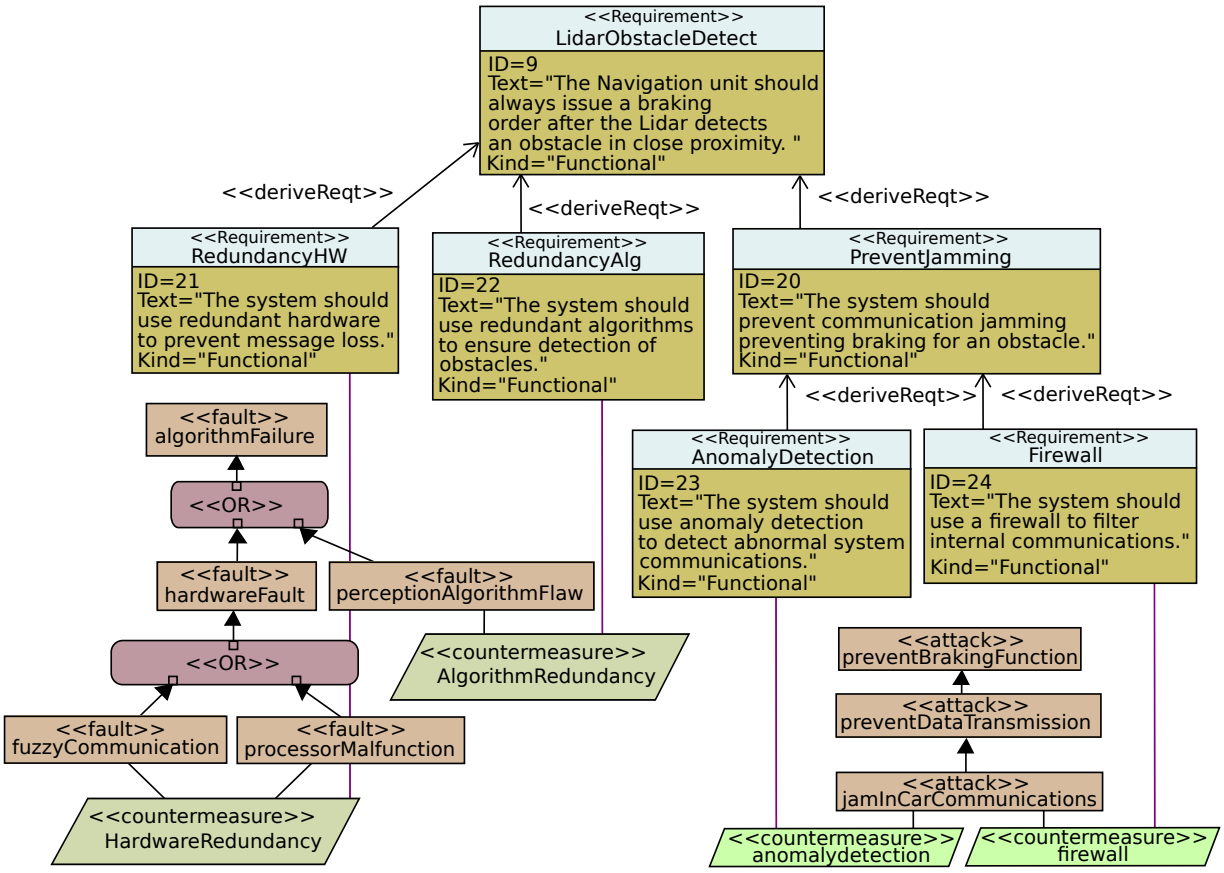


Figure 4-6: Linking Attack and Fault Trees into Requirement Diagram

4.4 Design Phases

The design of our system take part at two phases and levels: HW/SW Partitioning and Software Design. Starting design of a system at a higher level of abstraction can improve the efficiency of the design process [162, 175].

Abstractions can help in analyzing a large, complex system, by separating out the relevant properties for a certain concern from irrelevant details [162]. It helps by allowing a designer to take into account only the relevant parts of the system. For example, writing a program to sort numbers in java does not require any knowledge of the number of transistors in the computer, or any of the computer architecture details.

In this manner, we design our system at multiple levels of abstraction: first at a highly abstract level to determine the mapping, or HW/SW Partitioning, and then provide the details of the abstract software functions in the following Software Development phase. In the next sections, we describe the abstractions taken at each level.

4.5 HW/SW Partitioning

The HW/SW Partitioning Phase allows us to determine the high-level functionality and architecture of a system. The modeling process follows the Y-Chart approach [177], where application (high-level function) and architecture models are designed independently of each other, before application components are mapped to architecture. The HW/Partitioning Phase takes place in the DIPLODOCUS environment.

At this phase, we concern ourselves only with minimal detail needed to select an architecture. Therefore, we limit our modeling to the list of functions, their control operators, the complexity of their computations and their communication activities. For example, algorithms are abstracted into a computation time function, ignoring the exact calculations within, so that their execution time can be considered in the load on processors. Similarly, data communications are abstracted into the size of the data sent, and not the actual names or values of the data, so that the transit time of the communication can be taken into account in buses, bridges, and the sending and receiving processors.

4.5.1 Application/Functional Modeling

The Application/Functional Models consist of SysML Block (Definition and Internal) Diagrams and Activity Diagrams. The Block Diagram describes the system as a series of connected communicating Tasks, and the description of the communications. Communications can be ‘Channels’, ‘Events’, or ‘Requests’. Channel-type communications send only abstract data i.e. a quantity of data. Therefore, partitioning decisions are taken according to the amount of data the architecture can handle rather than on the specific values the architecture can support. Event and Request-type communications are used for coordination between tasks, including sending execution parameters to each other. More specifically, events are used for synchronization between two running tasks, while requests can be used to specifically start a single run of a different task.

Communications can be blocking or non-blocking for the sending or receiving task in certain combinations, and the maximum number of samples in the communications buffer that can be specified as finite

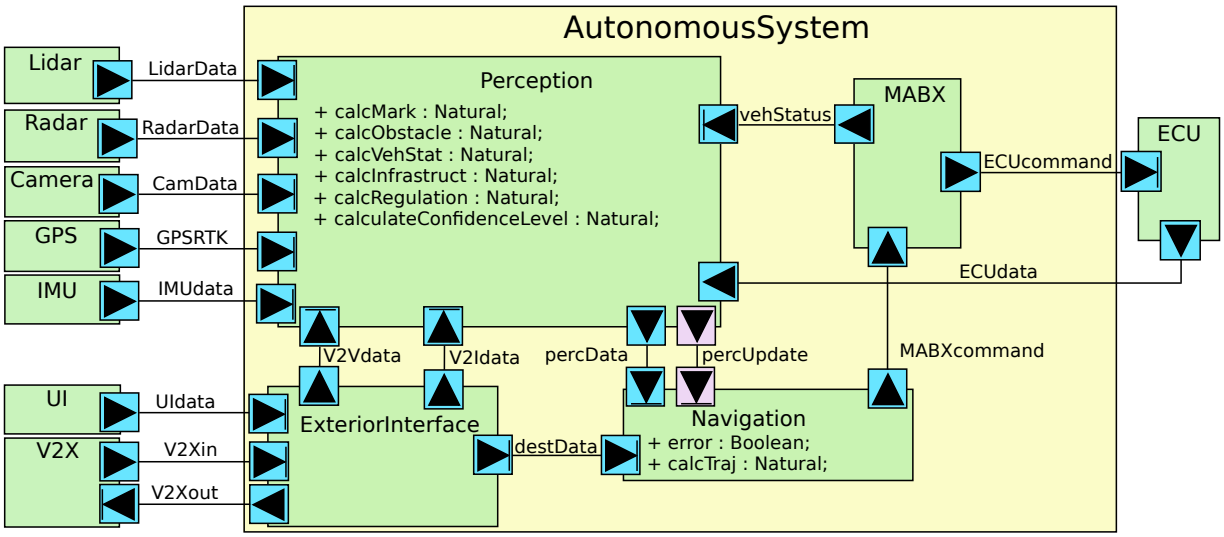


Figure 4-7: Application Model for Autonomous Vehicle

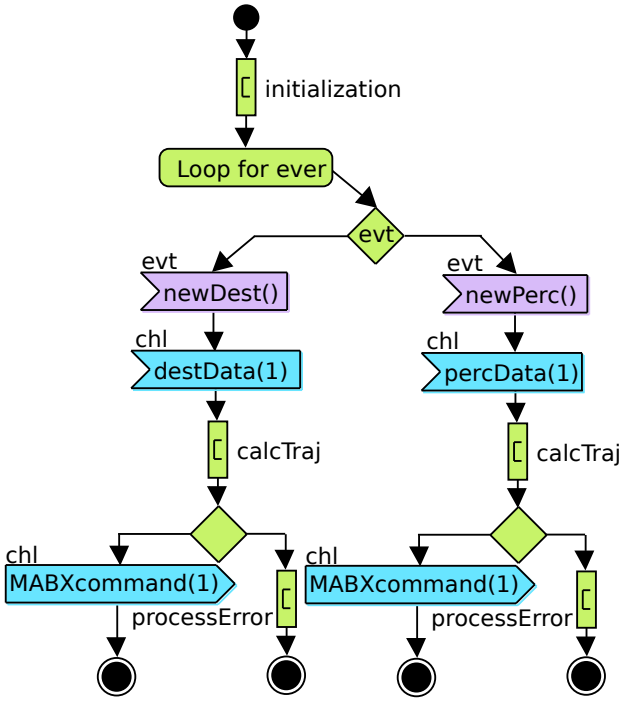


Figure 4-8: Activity Diagram for Navigation in Autonomous Vehicle

or infinite. For example, in a 'Non-Blocking Read - Non-Blocking Write' channel, the sending task can write infinite times, and the receiving task can read data even if the buffer is empty. In a 'Blocking Read - Blocking Write Channel', however, the sending task can only write until the buffer is full, and the receiving task can only read data if the buffer is not empty [94].

Figure 4-7 shows the Application Model for our Autonomous Car. The Exterior Interface manages the communications with the User Interface, used by the passenger to determine a destination, and V2X sys-

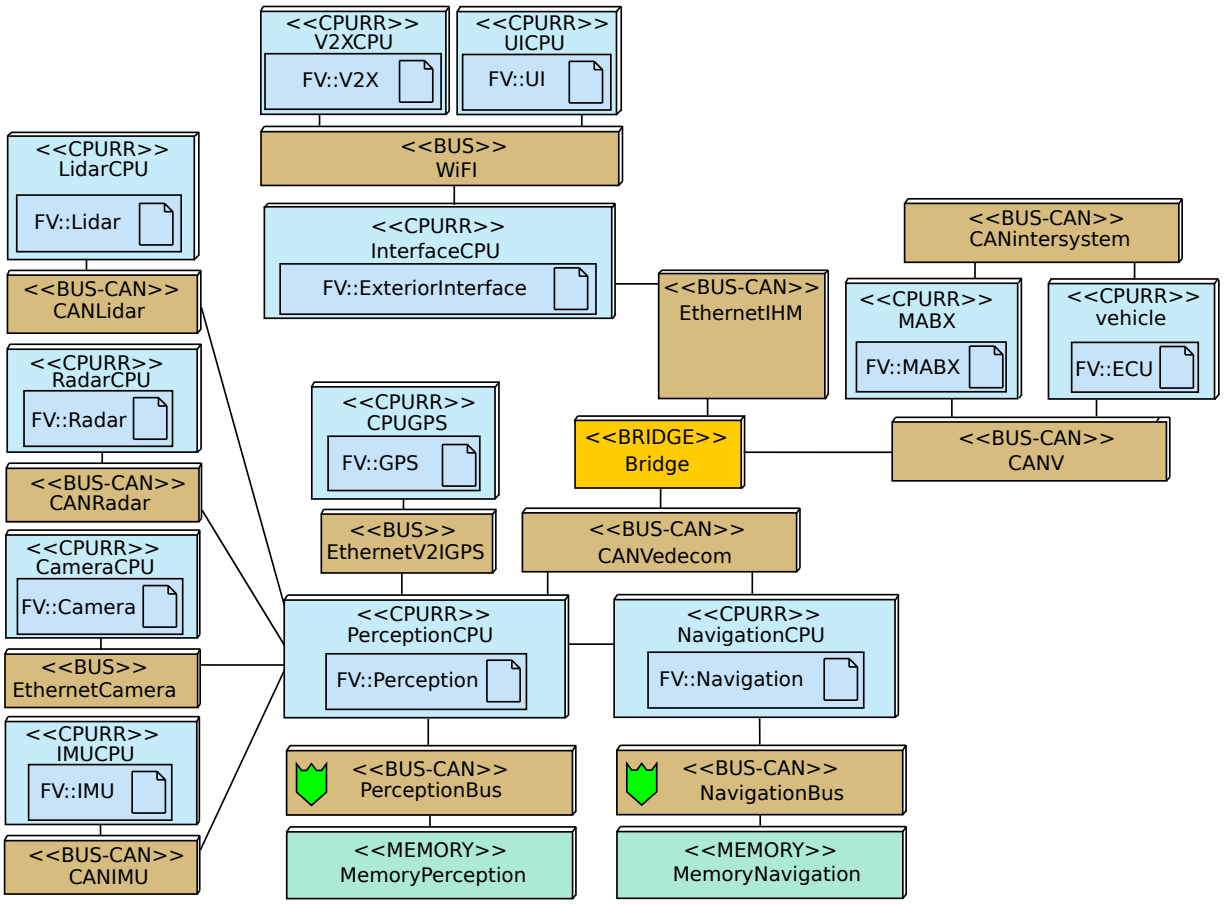


Figure 4-9: Architecture/Mapping Model for Autonomous Car

tem, which gathers data from the infrastructure (traffic lights, road information, etc) and nearby vehicles (traffic information, etc). The Perception unit communicates with the different sensors (Camera, LIDAR, Radar, Inertial Measurement Unit, GPS), and then accordingly calculates the obstacles present, and the confidence level of the current measured data. That data is then sent to the Navigation unit, which accordingly calculates a trajectory, and the vehicle commands that the vehicle should perform. The vehicle command is sent to the MABX, which filters the vehicle commands, and converts accepted commands into commands for the ECU.

Each individual task describes its abstract functional behavior using an activity diagram. Activity diagram elements include communication operators, computation elements, and control elements. Communication operators send Event or Channel signals to another task, displayed with the name of the event or data channel and the number of samples. Computation elements indicate the task performs some calculation or data processing, where complexity is expressed in terms of elementary operations (int, float, custom operations) as either a constant or interval. These complexity are transformed into cycle time when the task is mapped to an execution node: this time depends on the execution capabilities of the execution node. Control elements include *For Loops*, *Choice*, and *Select Event*, *Random Sequence* elements, which control the execution flow. Non-deterministic behavior can be modeled, such as with *Random Sequence* operators which execute a serious of operators in random order. *Select Event* operators allow execution of any possible received event next, and *Choice* operators allow for an execution flow to continue randomly

on any of the possible branches.

An extract of the activity diagram of the Navigation Task is shown in Figure 4-8. The navigation system starts with an initialization sequence modeled only as a complexity operator. After the initialization, the navigation task loops to continually wait for the reception of a new destination or new perception data (in the form of current proximate obstacles), indicated by the *Select Event* operator. For example, if it receives the "new perception" event from the Perception Task, it acquires the new perception status with the *Read Channel operator*. Next, it calculates the route based on the new data. If route calculation fails, it processes the error, and if it succeeds, it forges a new MABX command and sends it to the MABX box by writing data to the MABXcommand channel.

4.5.1.1 Architecture Modeling

The Architecture Model is a set of hardware components, with processing components of Hardware Accelerators and CPUs — a processor also contains an Operating System —, communication components of Buses, Bridges, and Firewalls, and finally storage nodes of Memories. Nodes have parameters to customize their behavior: data-path, performance capabilities, scheduling policy (processor, bus), cache-miss (processor), etc.

Processing nodes are components which can perform the functions mapped to them, and their performance parameters specify their behavior in terms of operating frequency, scheduling policy (round-robin, priority-based, etc). The communication nodes allow for messages to pass across them, and are specified by scheduling policy (to determine which communication transits first), and bus width (size of data that transits at a time). These performance parameters will be used to determine if they can support all of the functions or communications which are mapped to it.

Each Bridge or Bus can either be "Public", and therefore insecure and accessible to an attacker, or "Private", and therefore secure and inaccessible to attackers. These properties of communication architectural components affect the security verification described in the following chapter. At this point, reliability-related flaws in hardware, such as a lossy bus, are not modeled, but some can be modeled in the Software Design phase.

Figure 4-9 shows the Architecture for the Autonomous Car. Each node shows its specific stereotype. Moreover, processors are displayed in blue, Buses are displayed in tan, Bridges are displayed in dark brown, and Memories are displayed in green. Buses and CPUs are connected if there is a link between them. Additional memory should be provided to store data for channel communications, but they are not shown in the interest of space.

4.5.1.2 Mapping Modeling

With the application and architecture modeled, the application is then mapped to determine the location of functions and communications. Each task of the Application Model must be mapped to either a CPU or Hardware Accelerator. Tasks to be implemented in software are mapped to processor, while tasks to be implemented in hardware are mapped to Hardware Accelerators. Communication channels between tasks may be mapped to show their exact path along memories, buses, bridges and firewalls.

When the application is mapped to architecture, we can obtain performance metrics during simulation to evaluate the suitability of this mapping [163], such as worst case execution time, load of architectural components, etc. For example, when tasks are mapped to a processor, the load is calculated as the number of active cycles (performing calculations, sending data, etc) of the processor divided by the total number of cycles of the simulation. This metric can help us determine if too many tasks are mapped to a single processor, or if one task involves too many computation-intensive operations.

When samples of channel data across a bus, at each time, the number of bytes sent is equal to the width of the bus, meaning that the bus may break up a single sample of larger channel communications across multiple instances. This process repeats until all of the data are sent. The load of buses, bridges, and memories can then be calculated to determine if certain communication buses are supporting too many communications.

As shown in Figure 4-9, all of the sensors (Camera, LIDAR, etc) are mapped to the processors on the left, and the control system tasks, Perception and Navigation, are each mapped to a CPU in the center. While we do not need to design the ECUs, we include it in our model in order to model all the system communications. The prefix 'FV' stands for Functional View, which is the name of the Application/Functional Modeling. Tasks are distinguished by their originating functional model as a single mapping may contain tasks from multiple functional models.

4.6 Software Design

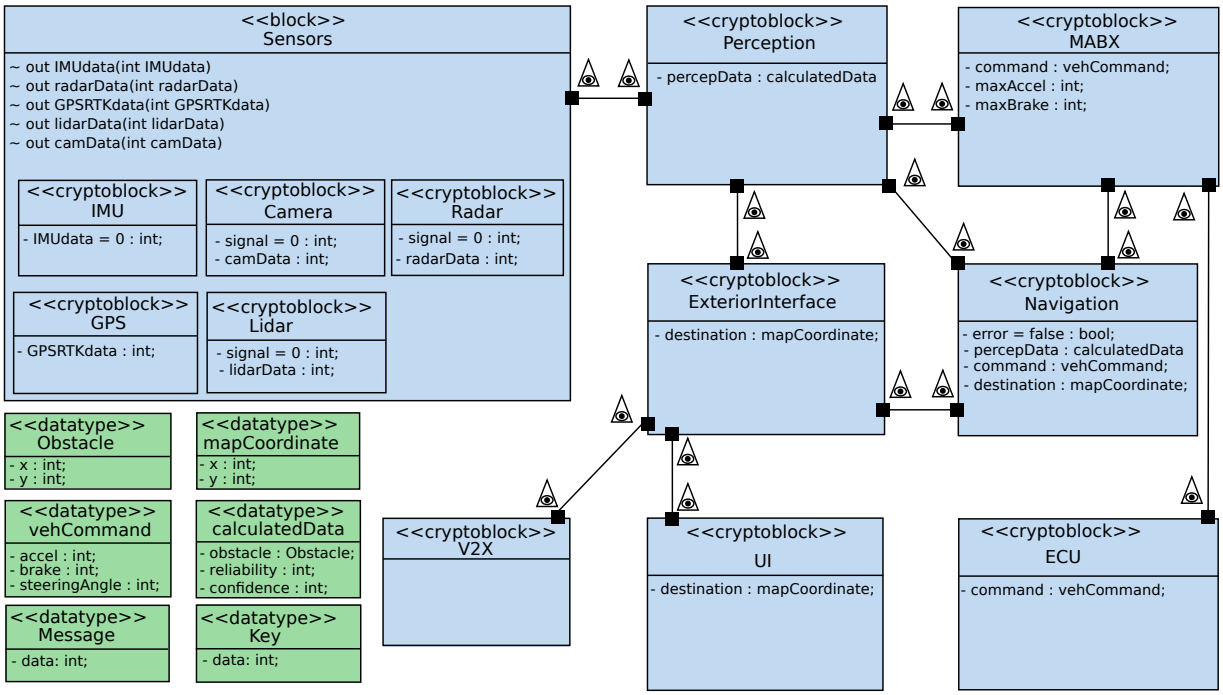


Figure 4-10: Software Design Model for Autonomous Vehicle

The Software Design phase enables the design of the detailed implementation of the system. The detailed functional behavior of the tasks implemented in software should be defined, so that the code can either be

automatically generated or manually developed from the Software Design models. The Software Design models take place in the AVATAR environment of TTool. AVATAR is capable of modeling and simulating system function, generating code, and taking into account safety and security issues within the same model. It supports safety formal verification using UPPAAL or the custom TTool Model Checker, and security formal verification using ProVerif [15, 252].

Figure 4-10 shows the Block Diagram, containing all of the tasks and showing the location of communications. Each block may correspond to one or more tasks from HW/SW Partitioning, and are refined to contain more attributes. In addition, user-defined data types beyond boolean and integer can be used. In this case, as the system performs operations on map coordinates, vehicle commands, etc which are composed of a set of integers, it can be more convenient to explicitly define them as a data type.

Communications are also refined to include more details, such as the exact values being transmitted. The communications between two blocks can be classified as *synchronous*, where the block sending a message must wait until the receiver is finished accepting the message before performing other operations, or *asynchronous*, where the block sending a message can send the message and proceed with other operations [110]. Communication links can also reflect the safety and security properties of the system. Links can be designated 'lossy', in which they drop packets with a certain probability. The ability of the attacker to access a communication link is modeled by classifying a channel as 'private', or inaccessible to the attacker, or 'public', and accessible to the attacker, and therefore marked with an eye symbol. In our example, all of the communications are public, since all of the communications are mapped to pass through public channels in the mapping. This definition will be described in detail in Chapter 6.

The behavior of each block is defined in a State Machine Diagram. These diagrams reflect the detailed behavior, refined from the activity diagrams in the HW/SW Partitioning phase. Security protocols can be modeled in Software Design, as a block can be defined as a 'crypto-block', and then the actions within the state machine diagram can include the cryptographic primitives such as symmetric encryption, verify certificate, verify MAC, etc. For example, a key distribution protocol was modeled in our paper in [209].

Figure 4-11 shows the first refinements of transforming the activity for the Navigation task for the Software Design phase.

The communications are presented differently in the HW/SW Partitioning vs Software Design phases. Communications in HW/SW Partitioning can send quantities of data (channel communications), or synchronize with events (events and requests). Both types of communications are classified as 'signal' communications in AVATAR.

Also, as we we will discuss in Section 6.3, we no longer use events for synchronization, so the reception of destination data or perception data is modeled with a single receive signal. Using the 'Select Event' operator in HW/SW Partitioning indicates that the first received event should be executed, which in Software Design is the same function performed by the choice of two receive signals. In addition, the exact values being sent in the communication are modeled. The reception of one sample of perception data is refined to be reception of the set of data including the coordinates of an obstacle and the confidence the algorithms have about the existence of this obstacle.

The functions performed by the task are refined as well. For example, the details of the algorithm for calculating trajectories would be added to state machine diagrams. With exact data modeled, we can determine if the new perception data and destination data require the generation of a new vehicle command, or if it can continue along the current route (such as continuing to drive along a straight empty stretch of

freeway at the same speed). Also, with the actual algorithms for detecting errors would be included, we split the one error state into a variety, and model how the system responds to errors at different phases in the calculation process.

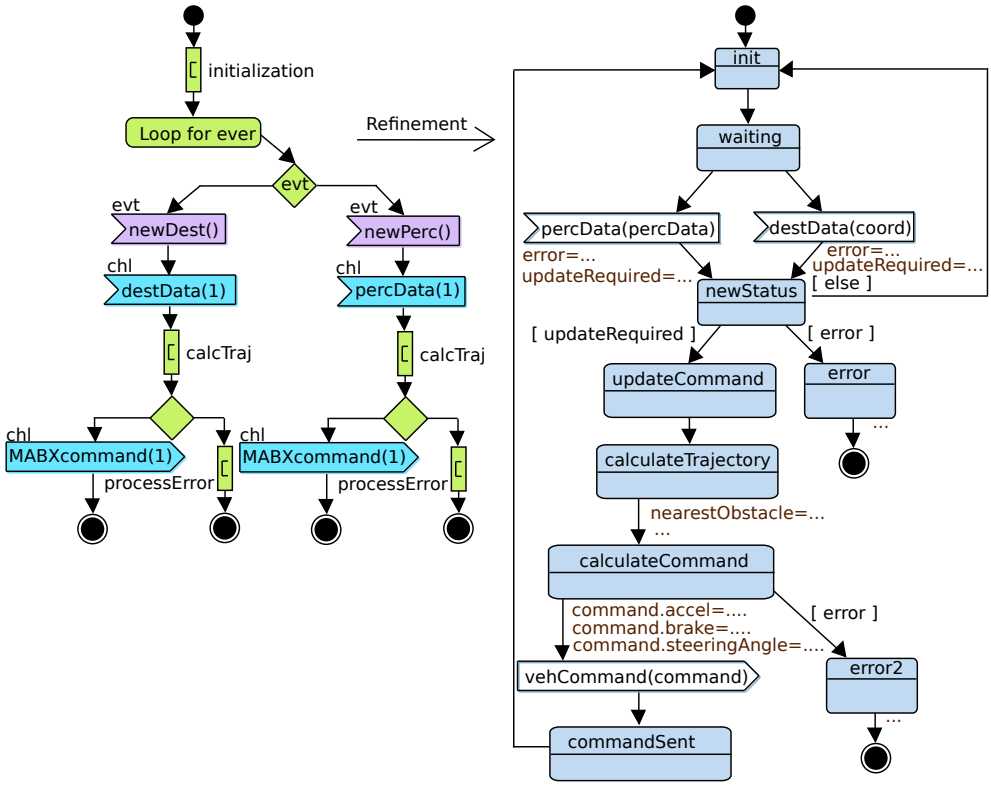


Figure 4-11: State Machine Diagram refined from HW/SW Partitioning Activity Diagram

Chapter 5

Security-Aware HW/SW Partitioning

> "Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves." – Alan Kay

5.1 Motivation

With the rise of attacks on connected embedded systems, ensuring their security is becoming an important research question. As systems may be too large and complex to accurately consider and analyze in one's mind, modeling helps to describe a system and balance all of the different requirements, including security.

As described in Section 3, most security modeling frameworks operate at a low level of abstraction. They tend to model the exact security mechanisms and all the data involved. We however, propose that there is value in starting security modeling at a higher level of abstraction.

Hardware / software partitioning, an important early phase in development of embedded systems, models the abstract, high-level functionality of a system and distributes functions of the system among candidate hardware architectures. This phase decides on the "best" architecture according to several criteria, including hardware cost and performance. Specifying the execution location of each function in architecture generates a model referred to as a 'mapping'.

Figure 5-1 shows the entire process of fixing a system after flaws are detected in an embedded system. If security flaws are detected in a production system (Step 1), then ideally they should be able to be fixed with a patch adding security mechanisms (Step 2), and before the flaw has been used maliciously resulting in monetary losses and/or personal injuries. While some security vulnerabilities were reported and demonstrated only by researchers, allowing the companies to repair the flaws before widespread damage [88], other vulnerabilities resulted in actual damages, such as from the Stuxnet, Mirai, and other botnet attacks [335]. Besides the monetary losses due to recalls or repairs, companies often also suffered reputation damages after security breaches [76, 172, 260].

However, patching a system by adding security mechanisms may causes the system to violate other requirements, such as timing (Step 3 in Figure 5-1). When modeling security mechanisms after selecting a mapping, designers may determine that the performance impact due to added encryption/decryption may render the selected mapping non-optimal, forcing them to redo their model and select a different mapping. The partitioning of hardware/software may also be changed if security protocols are chosen to be moved to be executed in hardware. Some of the security countermeasures that could be added, such as Hardware Security Modules and Firewalls, described in section 2.4 are also hardware-based. It may thus be necessary to re-consider the architecture (Step 4). In an embedded system, changing the hardware at this phase could prove to be costly.

Therefore, at a minimum, for an accurate selection of a mapping, security algorithms should be modeled in terms of their execution time. However, even if we add execution time operators to our models as with previous works [14, 288], it is not easy to determine where these protocols should be added in larger systems. Before they can be added, we must first determine the vital security properties of the system, such as which data need to be protected, which in turn depends on which data are accessible to an attacker. To determine what data is accessible to an attacker, it is necessary to determine which locations of the architecture are accessible to the attacker and which data is accessible from those architectural locations. As shown in Figure 5-2, modeling security without dedicated operators is not very detailed.

As systems change, the minimal required security algorithms will vary across architectures and attacker models. It may be difficult to keep track when security algorithms should be added. Furthermore, it would be more certain to use formal verification to check the security of models. However, without an ability to relate security algorithms to the data that they protect, it is not possible to check if data accessible to an attacker has been secured. Therefore, manually modeling security only in terms of overhead or calculation complexity, but without tracking data security, is insufficient due to the lack of support for security assessment.

Also, certain attacks, such as Denial-of-Service attacks, operate by adversely affecting the performance of a system [72, 266]. Performance of processors is only studied in the HW/SW Partitioning phase, so it may be helpful to model and simulate these attacks on the architecture, to better examine their impact.

To address the above-mentioned issues, in this chapter, we investigate how to express security properties in an architecture and mapping model despite its high level of abstraction. Modeling security protocols at this level of abstraction is not quite straightforward. As previously noted, data is modeled very abstractly. Data communications do not model the actual values, and algorithms are modeled only as an execution time and do not include the actual formulas, or attributes on which it operates. Security protocols cannot be directly added into these models, as the data being secured is not yet modeled at this stage. While we could model the exact data being secured, it is too detailed and unnecessary. The exact names of data being secured is not relevant to the factors considered in the selection of a mapping (cost, execution time, chip area).

We discuss how to model both the ability of the attacker to target the architecture, and efforts to adapt security mechanisms to the high level of abstraction. To support our security modeling, we distinguish between secure and insecure architectural locations. Communication buses can be modeled ‘internal’ (based on assumed attacker capabilities), and secure against an external attacker. On the contrary, external buses can be spied on. Functions mapped to the same processor can consider their message exchange secure. The chosen architecture thus affects the need for encryption algorithms and cryptographic material storage. Security mechanisms are abstracted to an execution operator, but also include a tag to help track the security operations applied to a certain data.

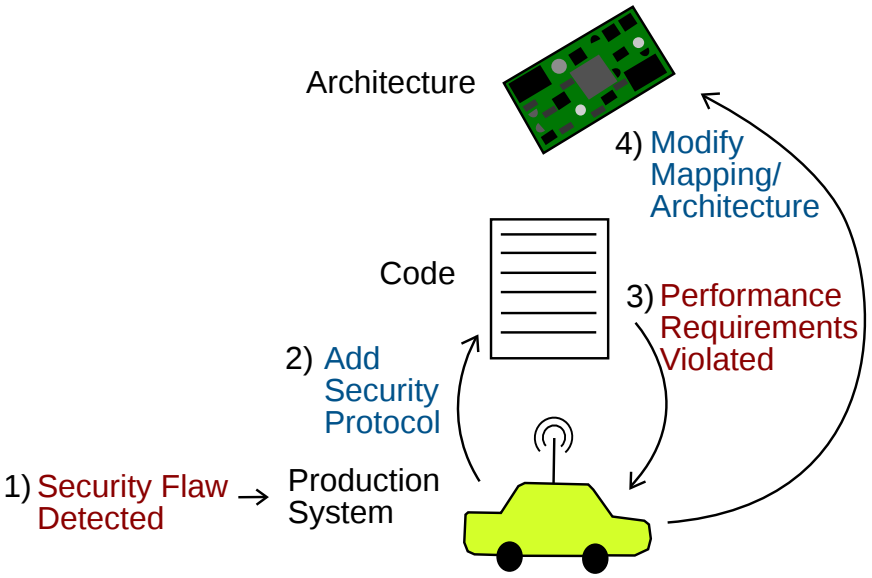


Figure 5-1: Fixing Security Flaws across levels of abstraction

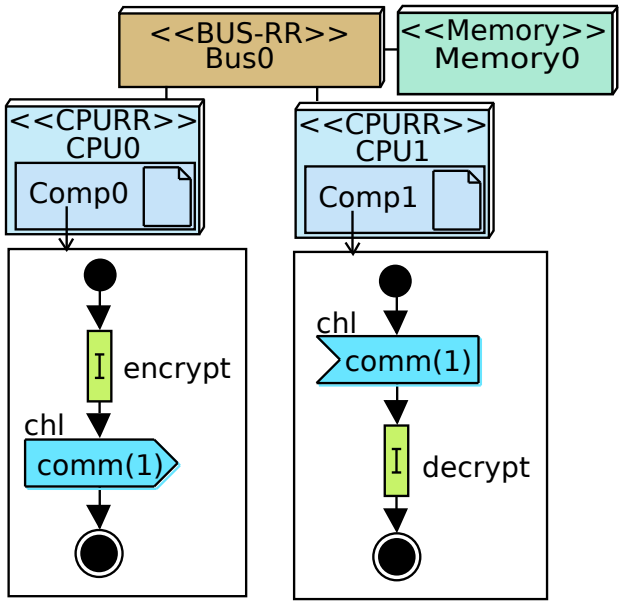


Figure 5-2: Modeling data security without dedicated operators

5.2 Attacker Model

We assume the same Dolev-Yao Attacker model as used for security analysis in Software Design models [209]. The Dolev-Yao Attacker is an idealized attacker who has full control of the network, as he can read, remove and send all messages on a public channel, and perform cryptographic operations (encryption, decryption, hash, and etc) and other operations on messages (splitting/joining) [55, 65, 85]. We assume the attacker has no physical access to the system, and therefore do not consider side-channel attacks [349]. This model assumes perfect cryptography, or that messages cannot be decrypted by brute force, but only

if the key is known. The Dolev-Yao model is said to use 'black-box' cryptography as it does not allow guessing of keys, and probabilistic models have been developed to address this limitation [60, 317, 350].

However, in our case, the Dolev-Yao model is sufficient for describing security during HW/SW Partitioning as we are interested only in abstract representations of security, and we only need to classify communications as either accessible or not accessible to an attacker, matching with the Dolev-Yao models of public vs private channels. Also, by using the previously-adopted verifier, ProVerif, and we can re-use much of the basic block to Proverif translation to minimize the amount of translations built in TTool.

5.3 Security Modeling

HW/SW Partitioning diagrams were enhanced with new elements to describe both attacker behavior and security mechanisms. Figure 5-3 shows a detailed metamodel of the security modeling elements.

5.3.1 Architecture Vulnerabilities

Assumptions regarding attacker capabilities are reflected on the architecture. On the architectural modeling, buses can be specified as public or private, corresponding with the Dolev-Yao model of whether they are accessible or not to an attacker. For example, devices communicating on a WiFi network would be modeled as exchanging over a public bus, while the internal bus would be modeled as private. Private buses are marked *secure* with a green shield, as shown on the architecture diagram of Figure 5-4. On the other hand, the public bus 'PublicBus' does not contain a green shield. The distinctions between bus types also model assumed attacker capabilities: if we assume that an attacker has no physical access to the system, then we can describe internal buses as private, but if an attacker could physically probe the bus, then it must be indicated as public.

5.3.2 Attacker Scenarios

Attacker scenarios explicitly model the attacker interactions with a system. They are based on the Attack Trees, but contain more implementation details and can be simulated in conjunction with the system model. By modeling attacker actions directly, we can better examine his effect on the system and evaluate possible countermeasures.

An attacker scenario consists of one or many attacker tasks, representing separate functions working together to carry out an attack on a system. An attacker task's behavior includes all possible actions of a normal application task (read/write data on a channel, control operations, execution of calculations, etc). In addition, where regular tasks can only read and write on channels directly associated with themselves, the attacker tasks may read and write on any public channel. A public channel is defined to be any channel mapped to a path including at least one bus accessible to the attacker. Furthermore, attacker tasks possess an additional capability: 'Code Injection', which replaces an application task's behavior with an attacker-determined one, modeling an attacker's capability to change a task's execution flow which may include code modification.

Attacker behavior during simulation depends on the architecture. Read/Write Public Channel commands can only be successfully executed if the communications are in fact mapped to at least one public bus. We

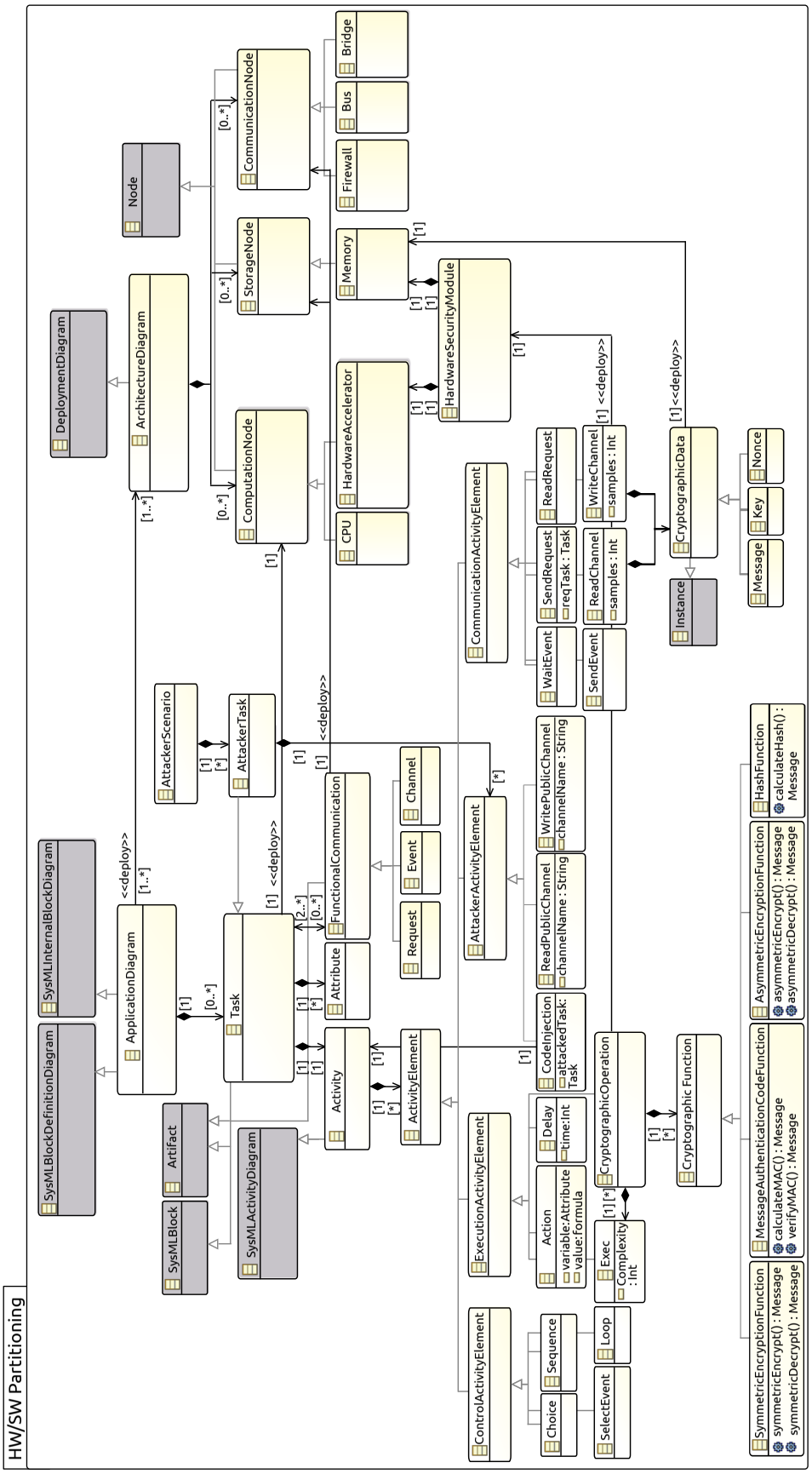


Figure 5-3: Security Modeling Metamodel

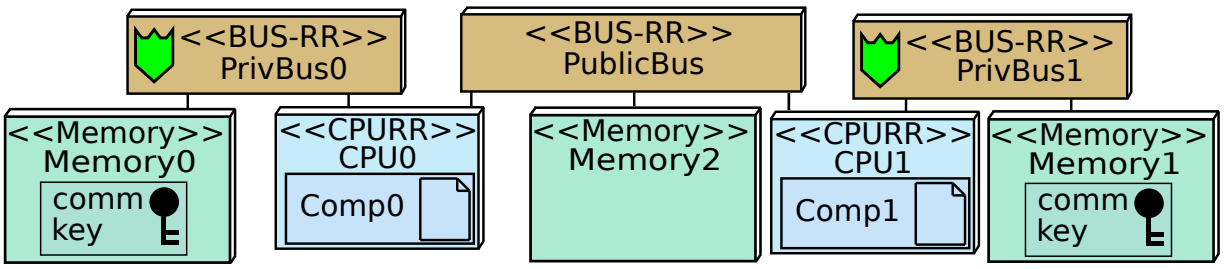


Figure 5-4: Sample Architecture with Insecure Bus

considered classifying certain storage nodes/CPUs as tamper-proof, but while there have been many works on preventing code injection with stack canaries or ensuring memory is not both writable and executable [71, 251] and tamper-proof program execution/memory such as Execute-Only Memory [19, 22, 57, 116, 204], attacks have managed to bypass some of these countermeasures [53, 68, 97, 125, 165]. Therefore, we could not guarantee the existence of completely tamper-proof CPUs/memories.

At this point, we base the assumption of the ability of the attacker to modify code based on previous attacks such as Miller and Valasek's hack through the vulnerability in the cellular data connection, and Tencent's hack on Tesla through a browser [223, 234]. The Vedecom prototype is not yet fully equipped with all of its connections, so while we cannot test if they contain vulnerabilities, we can assume possible sites for attacks based on previous works. The V2X gateway is open to the outside world, and is often proposed as a likely source for attack [194, 248]. While methods of protecting the V2X Gateway have been proposed, we cannot be assured that an attacker will never bypass them, and thus we investigate the damages that an attacker could cause by modifying the code of the V2X Gateway, and then how to prevent those losses.

We demonstrate this idea by generating one attacker scenario based on the attack tree from Figure 5-5, more specifically the branch on preventing the braking function by jamming in car communications. In this attack example, the attacker is not attempting to change the behavior of the system via forged messages (as protection mechanisms may prevent forged messages from being accepted), but instead trying to delay valid commands from the system and possibly provoke an accident due to late braking. However, when we start to generate the step "Modify V2X Code" in order to effectuate the "inject ECU commands" action, we realize that on the current architecture, the V2X Gateway has no access to inject communications into the ECU Gateway. The only components that the V2X Gateway communicates with are the Exterior Interface and the User Interface. Since the Exterior Interface is connected to the ECU Gateway, if it could be hacked, then fake ECU commands could be sent as quickly as possible. Figure 5-6 shows how the attack scenario executes on the architecture. The attacker compromises the V2X Gateway, which then compromises the Exterior Interface. Then, the Exterior Interface mass injects ECU commands to the ECU Gateway.

The activity diagrams of the components relevant to this attacker scenario are shown in Figure 5-7. They are modeled to have a 'hacked' (when the attacker is active) vs 'normal' behavior (without the attacker). By comparing the behavior of the system with and without the attacker, we can better understand the attacker's effect. As we can see, the attacker can signal a code injection to the V2X Gateway. When the V2X Gateway receives the signal, it changes to the hacked behavior, which involves signaling a code injection to the Exterior Interface. The hacked Exterior Interface then begins sending ECU commands as quickly as possible in an attempt to occupy all the processing time of the ECU Gateway and prevent processing of the legitimate commands sent by the MABX.

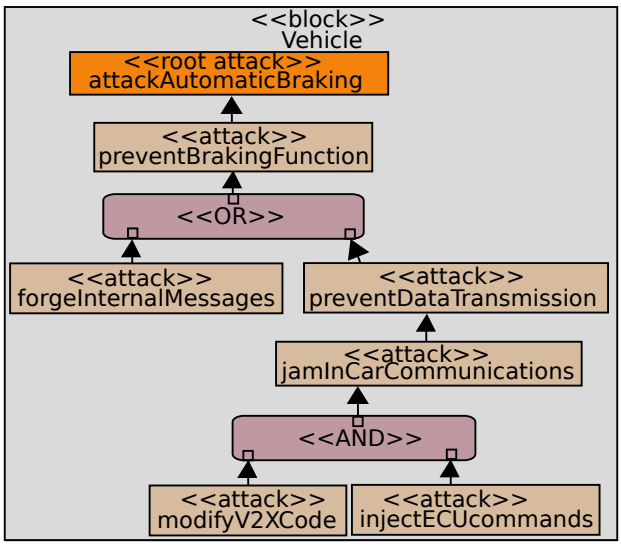


Figure 5-5: Extract of Attack Tree for Attacker Scenario Model

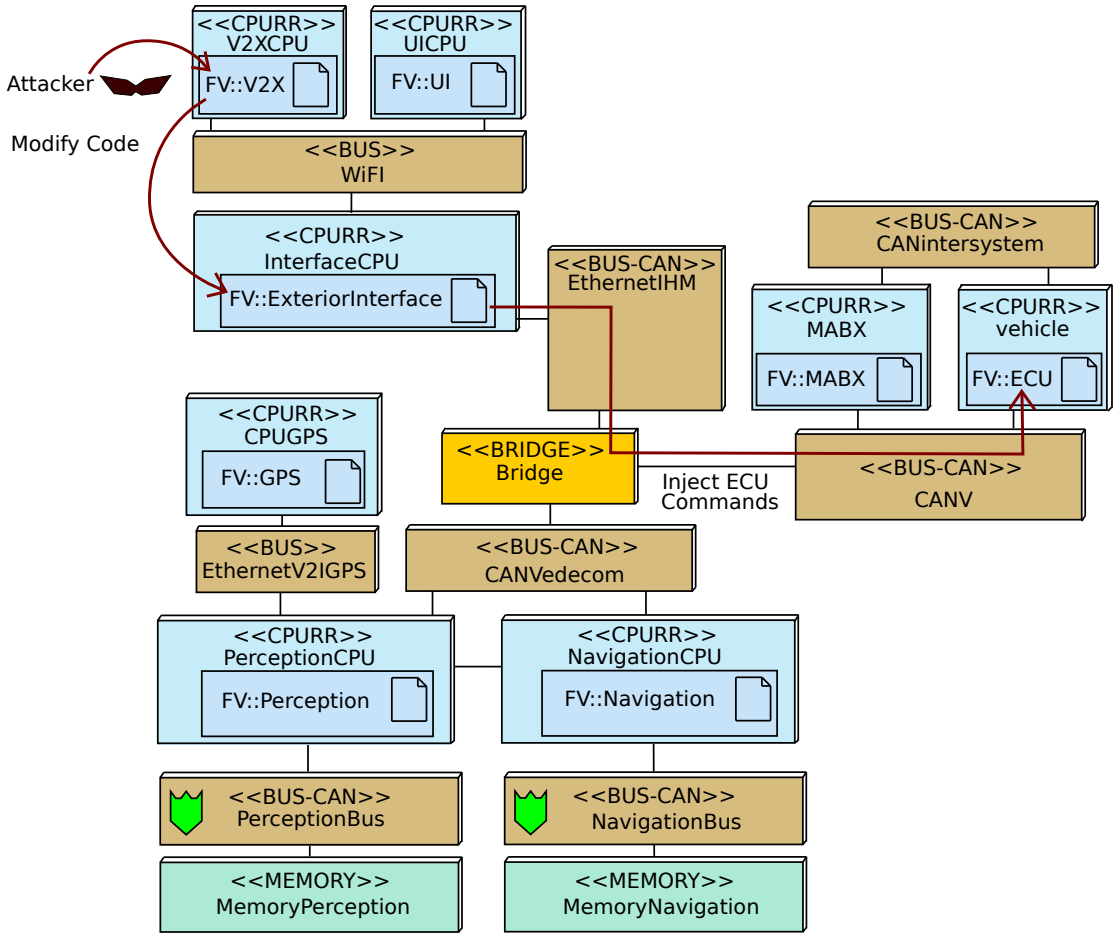


Figure 5-6: Attacker Scenario execution on hardware

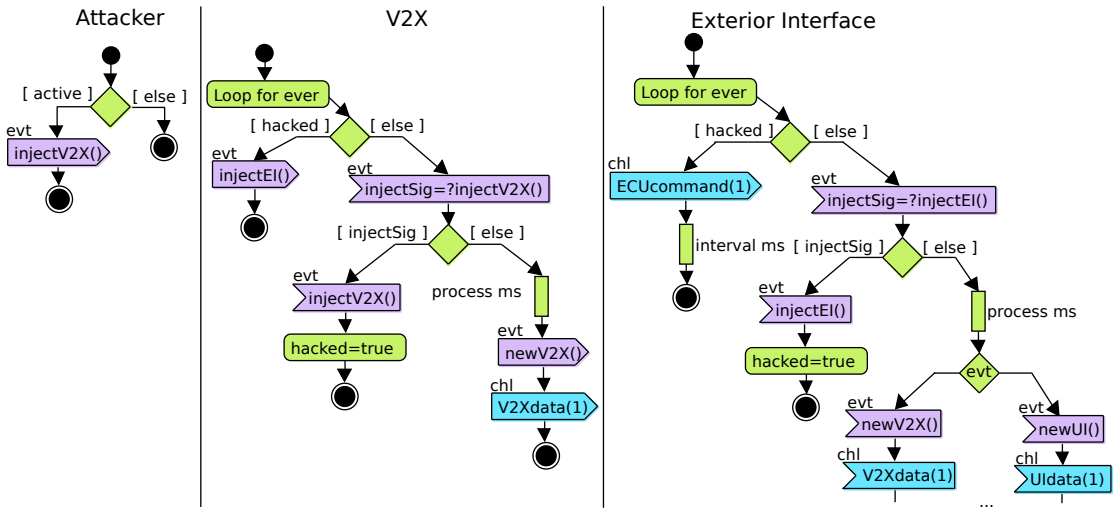


Figure 5-7: Activity Diagrams of components in Attacker Scenario

5.3.3 Attacker Scenario Analysis

Attacker Scenarios can be simulated to analyze their effect on the system. One of the new security properties that can thus be analyzed is 'Availability'. During simulation with and without the attacker, we can check the load across processors and the number of forged commands sent compared to the number of legitimate commands sent, to see if the attacker can effectuate a denial of service.

Figure 5-8 shows the performance impact due to the addition of the attacker. The MABX box experiences a significant decrease in active time as it is often waiting for its command to be accepted by the ECU Gateway, which now has to process both the ECU commands from the hacked Exterior Interface as well.

As this system is vulnerable to denial of service attacks and does not preserve the property of Availability, in the rest of the chapter, we describe countermeasures which can better secure our system.

5.3.4 Security Countermeasures

5.3.4.1 Data Security Mechanisms

When an attacker can access a public bus, and therefore all of the communications which traverse across it, the attacker can tamper with the communications in a method that could adversely affect system behavior. As described previously, an attacker should not be able to send commands and gain control of a neighboring car, or modify perception data to indicate that there are no obstacles. In other fields, the insecure communication protocols to communicate wirelessly with implantable medical devices such as pacemakers has allowed researchers to recover personal and treatment information from the patient, disable the device, or send a dangerous shock to the patient [214].

Securing the communications can involve adding a proven security protocol. While exact security protocols are modeled in the later development phases [209], in this phase, we still need to take into account the execution time to execute security protocols for an accurate HW/SW Partitioning, and find a method

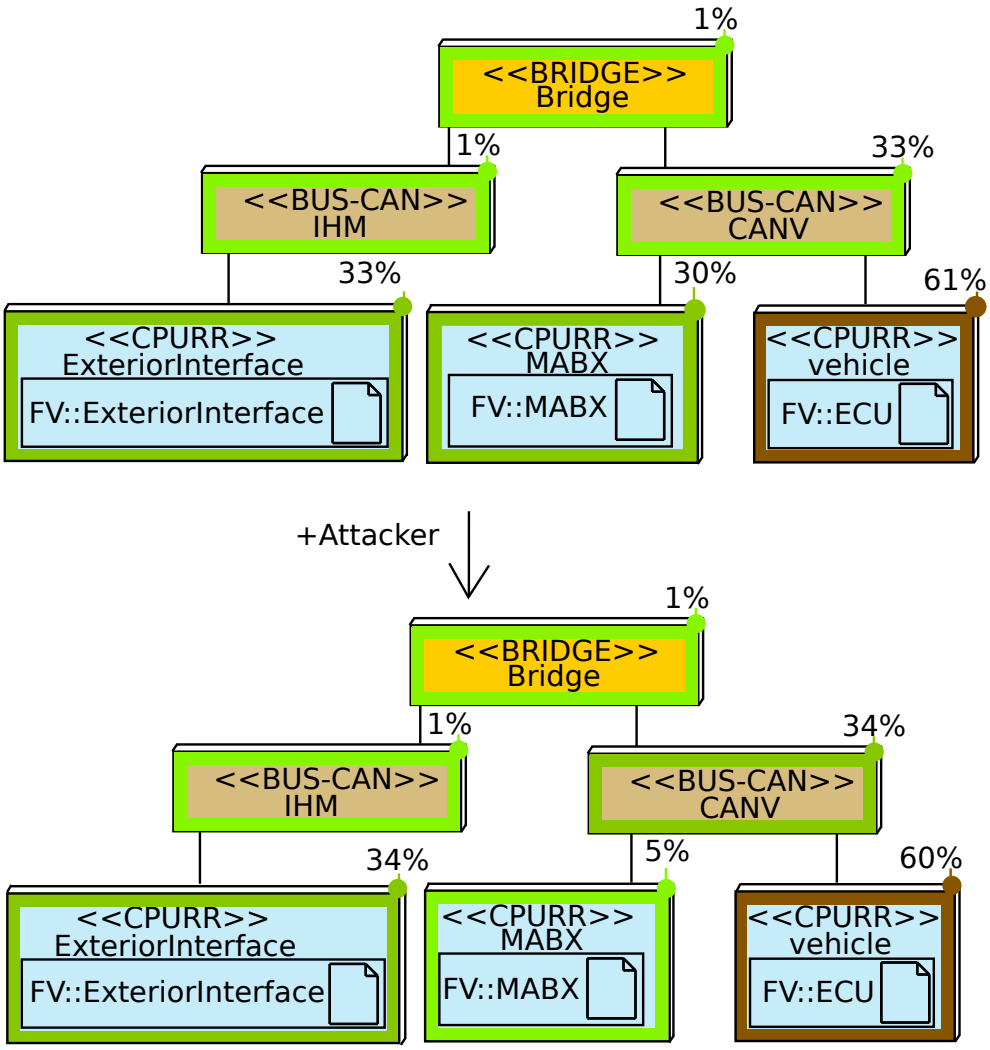


Figure 5-8: Performance impact due to Addition of Attacker

of formally verifying that data is indeed secure. At this level of abstraction, we are not interested during partitioning in the implementation of encryption algorithms; we only need to consider parameters that will affect the partitioning choice (satisfaction of security properties, execution time, data size).

To abstractly model security protocols and take into account only these relevant properties, we introduce *Cryptographic Configurations*, a tag to be added to communications to indicate the presence of security and the relevant performance overhead. When a channel is tagged with a Cryptographic Configuration, the user can also select if any cryptographic operations have been performed on the data.

Cryptographic Configurations are graphical artifacts that allow the security verifier to track data encryption elements. Within activity diagrams, they appear as an upside-down pentagon marked with their type, as shown in the left side of Figure 5-9, where 'AE' represents Asymmetric Encryption and 'D' represents Decryption. Cryptographic Configurations can be typed as follows: *Symmetric Encryption* and *Asymmetric Encryption* patterns encrypt data along with a key/keys specific to the pattern. A *MAC* can be added to messages to authenticate it and determine if it has been modified. *Hash* calculates a hash of the data. *Nonces* can be concatenated to messages before encryption to verify authenticity. *Advanced* allows the

user to indicate their own encryption scheme, such as combinations of Cryptographic operations.

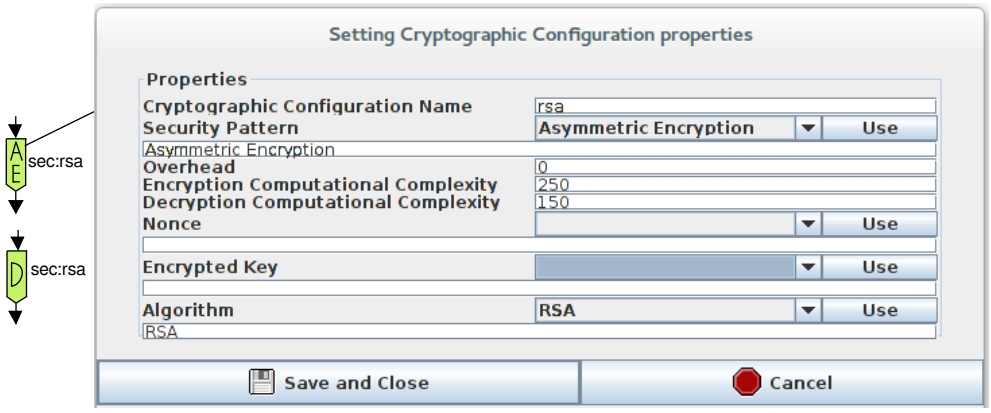


Figure 5-9: Specification of Cryptographic Configuration for Asymmetric Encryption and Decryption

Figure 5-9 (right) shows the specification of a Cryptographic Configuration. The designer can choose the type and then the corresponding performance properties, or select a pre-built algorithm which will automatically estimate the performance parameters based on the experimental results in [73]. These estimated parameters should be modified to better reflect the actual CPU used, but they can still be used to compare performance of different algorithms.

Encryption operations may be characterized by an encryption and decryption computational complexity (a measure of the relative execution cycles depending on the processor), and overhead (additional bits added to the message due to encryption). The computational complexities for operations (encryption, decryption, forge key, calculate hash, etc) apply for the block size (or block size for default key size if key size can vary), such as 128 bits for AES, 512 bits for SHA-256, and 214 bytes for 2048-bit key RSA [74, 131, 250]. If the data length exceeds the block size, then the data is encrypted over multiple runs of the algorithm, with total computation complexity = message size / block size * 1 block computational complexity. Within the encryption configurations, a specified nonce may also be concatenated onto messages to prevent replay attacks. Cryptographic material such as nonces are characterized by a size in number of bits. These parameters allow us to model the impact of security mechanisms on performance when evaluating candidate mapping. In addition, Cryptographic Configurations can apply to keys, where keys are encrypted and then distributed.

By default, the data is sent with all cryptographic operations applied, but in some cases, such as with the use of Hardware Security Modules as we describe next, it is necessary to send the data in the Cryptographic Configuration in its unsecured form and without a MAC. When a channel sends such data, the name of the Cryptographic Configuration is written in red instead of black. Distinguishing between the to possible forms of the data in a Cryptographic Configuration is important for when we wish to track the data in a Cryptographic Configuration across multiple tasks. There exist more complex situations than one task encrypting data and then sending it to another task to be decrypted. For example, let us imagine a case where Task1 sends unsecured data to Task2, who then encrypts it, and then sends the encrypted data to Task3, who then decrypts it, and sends the decrypted data to Task4. If we wish to check the Authenticity of the data, to ensure that the same data sent by Task1 is that received by Task4, then we need a method to tag the data across this path to link them.

Each key must then be mapped to an accessible memory, and is considered inaccessible to the attacker

only if the path to memory is secure. Otherwise, any data encrypted with the key is recoverable by the attacker. Mapping keys helps determine if the architecture allows sufficient secure paths to memory for storage of keys.

In our case study, to prevent an attacker from forging and injecting perception data, we wish to ensure the authenticity of this data. One method to ensure this security property would be to exchange a nonce between the Navigation and Perception tasks, calculate a message authentication code (MAC) that is concatenated onto the message, and have the Navigation task verify that the nonce and MAC both match the expected values. We will describe how these security operations ensure authenticity in greater detail in the following chapter in Section 6.8.

5.3.4.2 Hardware Security Modules

Security protocols can add undesired overhead. As previously mentioned in the list of possible countermeasures, Hardware Security Modules (HSM) are a specially designed hardware element for performing cryptographic operations more efficiently than regular processors. We model them as hardware accelerator that performs encryption and decryption in fewer cycles.

For example, we can add a HSM to the Perception task to perform the security operations previously described on Perception data. Figure 5-10 shows a HSM added onto the architecture diagram, involving a Hardware Accelerator and secure memory.

Figure 5-11 shows the Activity Diagrams for the Perception task and HSM task. The Navigation and Perception tasks previously exchanged a nonce. When the Perception task needs to send Perception data, it sends a request to start to the HSM, and then sends the nonce, and then the data. The HSM then receives the data, concatenates the nonce onto it, calculates the MAC, concatenates it onto the original Perception data, and then sends the combined message back to the Perception task. The Perception task then sends the Perception data + MAC to the Navigation task. As previously described, the Cryptographic Configuration name is written in red on the write channel operator sending the unsecured data to the HSM.

It is necessary to manually mark if cryptographic operations have been performed on the data or not, for instances where we need to track data which has been sent between tasks before the execution of encryption, hash, etc for purposes of verification. In this case, we need to know that the data hsmSec_perceptionData sent by the Perception task was concatenated with a MAC by the HSM, before being sent back to the Perception task and then sent to the Navigation task. We describe the importance of distinguishing between unsecured vs secured data for security proofs in the following chapter.

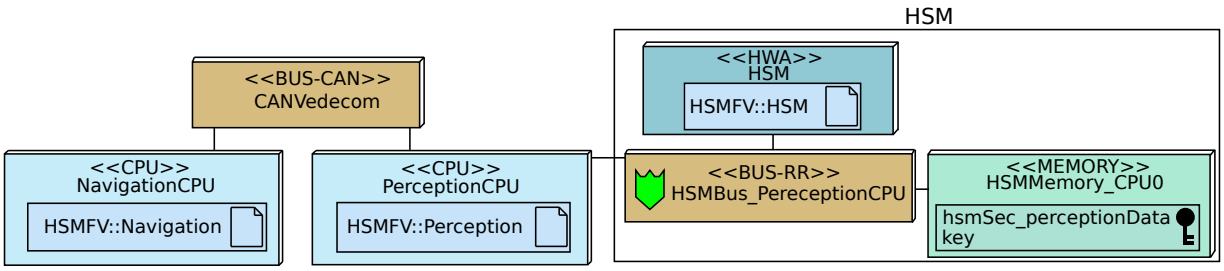


Figure 5-10: HSM in Architecture Diagram

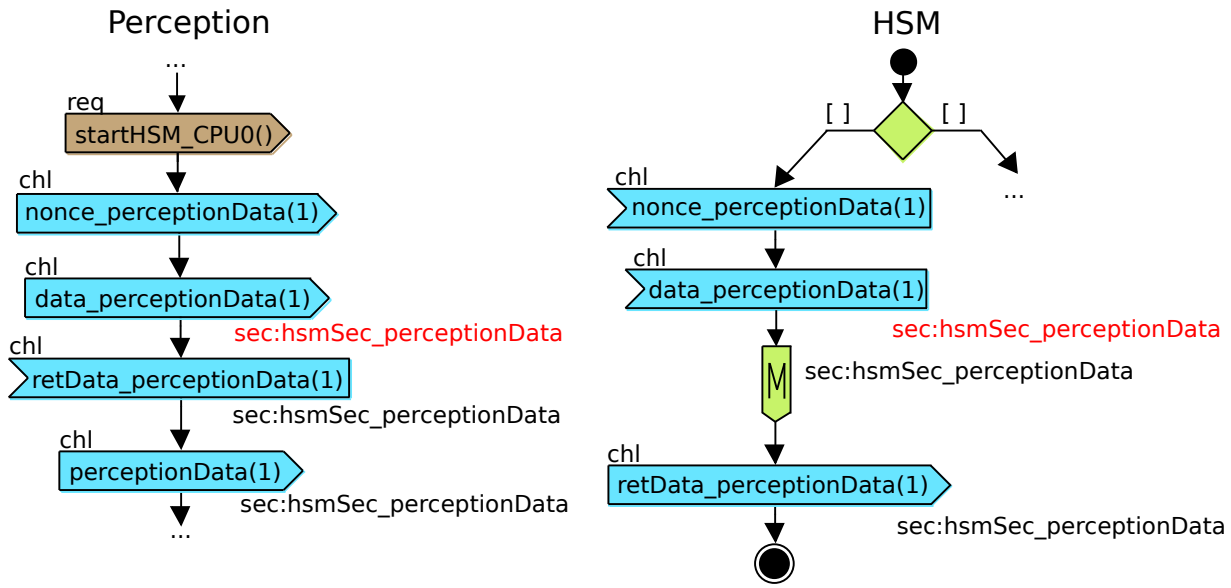


Figure 5-11: Perception and HSM Activity Diagram

5.3.4.3 Firewall

In our attacker scenario, securing communications alone is insufficient to prevent the denial of service attack blocking the reception of a legitimate vital message. A system should be able to detect and block abnormal communications, and possibly detect that a component has been compromised and controlled by an attacker.

Firewalls can be used to filter communications. They can be modeled as a task which either forwards or drops communications based on the current firewall rules. Rules can be dynamically modified via signals, to represent how the Firewall may need to block a communication which have been detected as abnormal. If the Firewall receives a communication that should be blocked, it does not forward the communication to the destined receiver.

For example, a Firewall can be added to monitor communications between the Exterior Interface, Perception, Navigation, and MABX tasks as shown in Figure 5-12, as these are the most critical communications. Furthermore, while the Exterior Interface offers some separation from the V2X Gateway open to the world, additional protections may be desired.

When the firewall is added, all the communications which pass through the firewall are re-routed in the Component Diagram as shown in Figure 5-13. Communications are split into a “firewallIn_channel” and “firewallOut_channel” for each channel, where the original sending task sends data along the “firewallIn_channel” to the firewall, and the firewall forwards communications along the “firewallOut_channel” to the original receiver. For example, perception data is usually sent between the Perception and Navigation tasks. When the firewall is added so that the communication traverses it, the perception data channel is replaced by the “firewallIn_percData” and “firewallOut_percData” channels.

The Firewall activity diagram is shown in Figure 5-14. Firewall rules can be dynamically modified by any task through events, though simplified in this case to only the Navigation task. The *updateRule* event indicates which channel is to be modified, and if it now allowed or not.

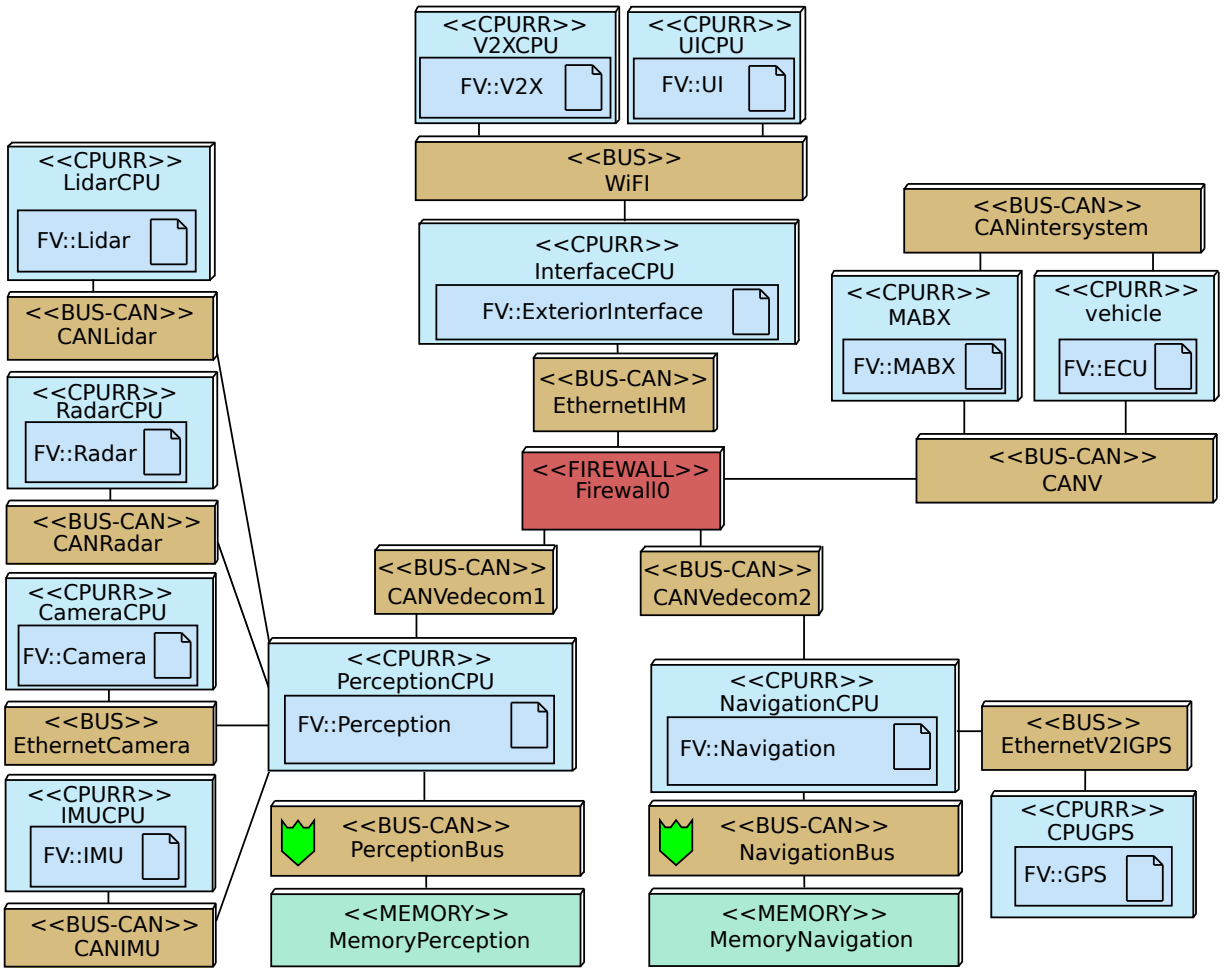


Figure 5-12: Firewall added to Architecture Diagram

When communications traverse the firewall, the firewall receives the “firewallIn” channel, and checks if it is currently allowed or not. If the channel is allowed, the firewall sends the corresponding “firewallOut” channel, and if the channel is to be blocked, it does nothing. A delay is added to model the latency of the firewall to process the data and apply rules.

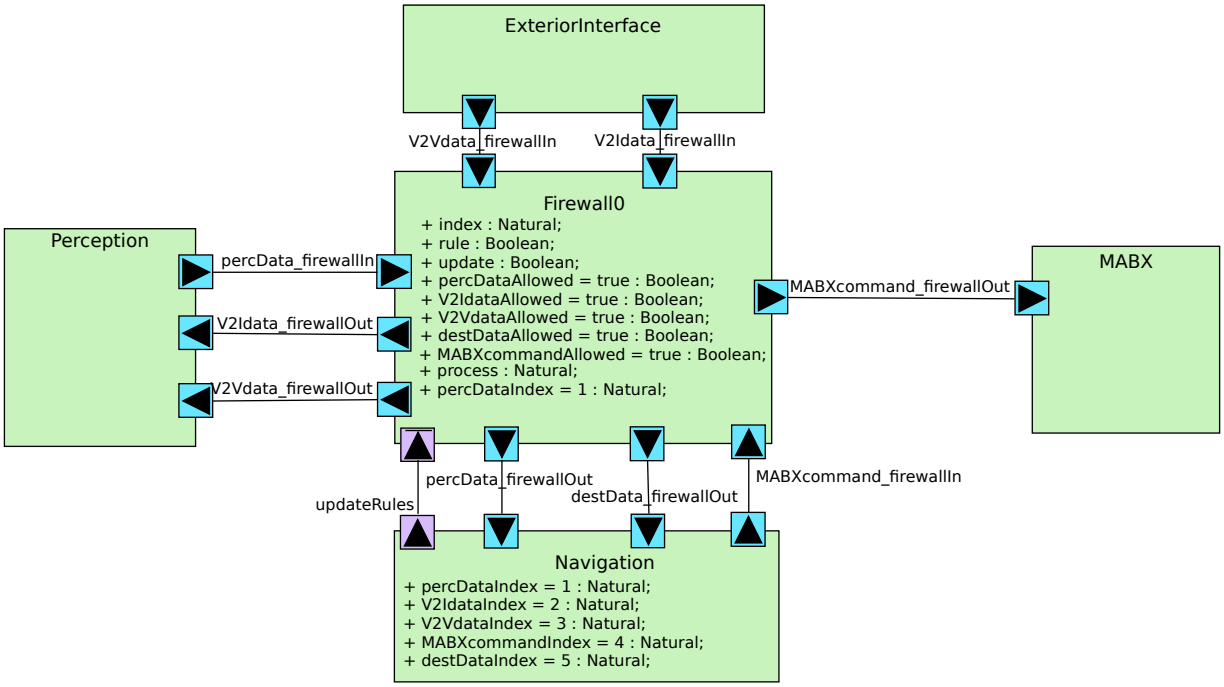


Figure 5-13: Component Diagram with Firewall

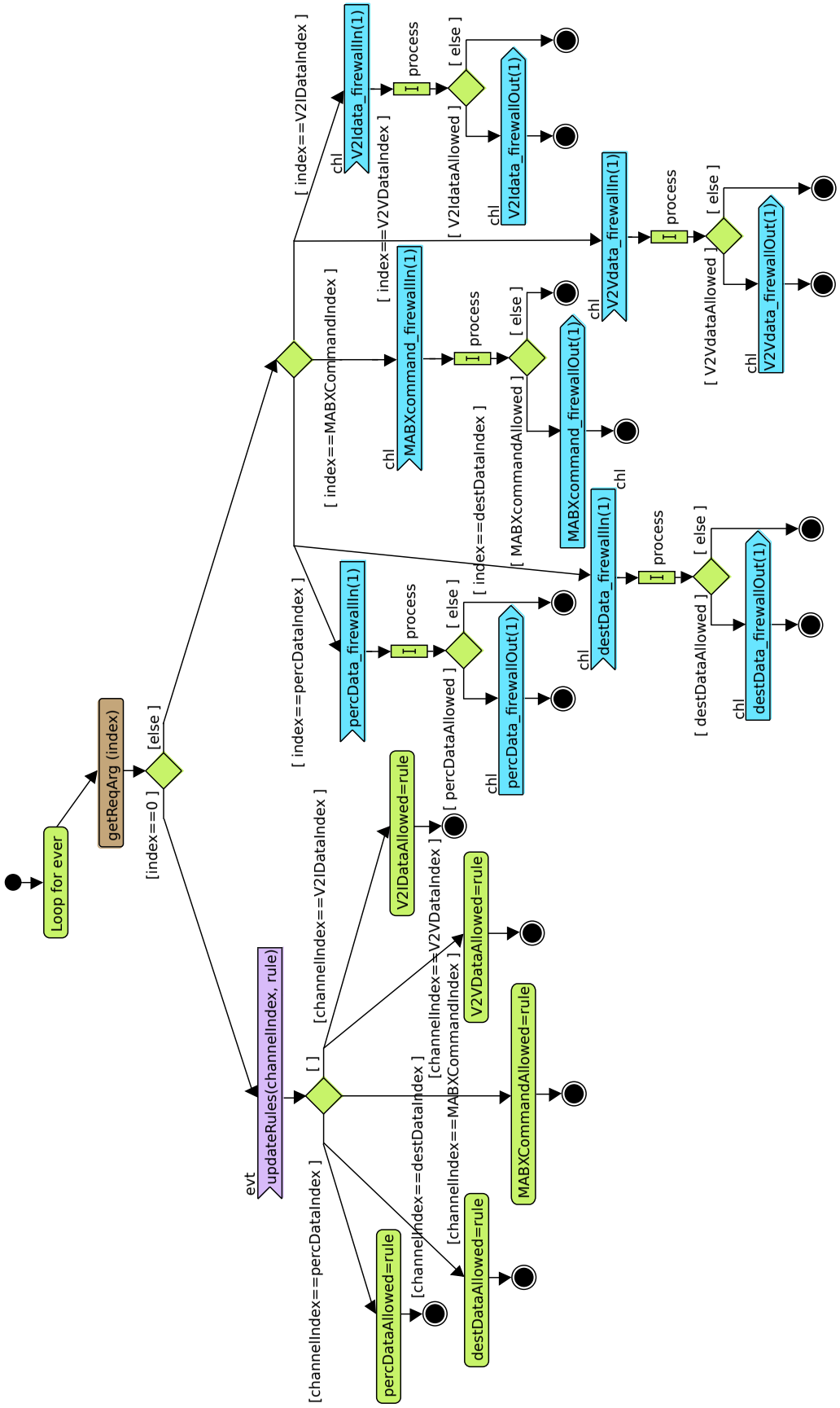


Figure 5-14: Firewall Activity Diagram

5.4 Conclusion

In conclusion, issues related to security are present in the HW/SW Partitioning Phase. Architectural components can be vulnerable to an attacker, therefore allowing their communications to be tampered with or spied on, and these vulnerabilities should be reflected in the architecture models. Also, the security countermeasures to be added affect the architecture and mapping. The execution time of security protocols should be taken into account during mapping, and the cryptographic keys used in the protocols should be provided with a secure storage location. Other security countermeasures such as Firewalls and Hardware Security Modules should be added directly to an architecture.

In a methodology without security modeling capabilities in the HW/SW Partitioning phase, all security considerations must be left to the final Software Design phase.

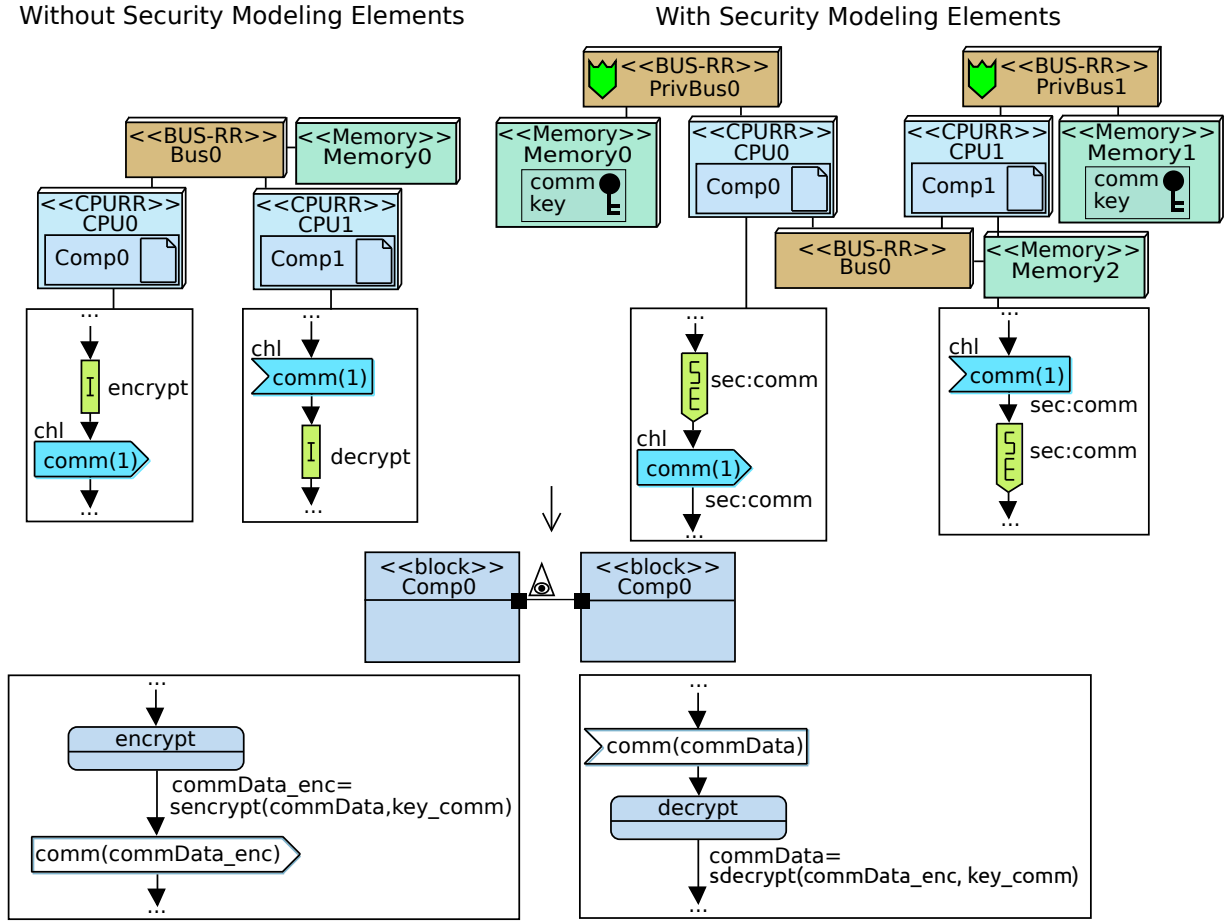


Figure 5-15: Mapping Model with and without dedicated Security Operators

Figure 5-15 shows an extract of an architecture without security modeling mechanisms, and the resulting software design model. Without security modeling during HW/SW Partitioning (diagram on left), all the security decisions are first modeled in the Software Design Phase. These considerations include deciding if each channel between tasks is accessible or not to an attacker, security protocols, etc. Instead, modeling the security of hardware communications and security protocols (diagram on right) results in a more gradual addition of security at each level of abstraction. Furthermore, we can determine if a system contains

sufficient secure memories by considering the secure storage of keys.

Furthermore, the security modeling in this phase helps us enhance our attack trees, to better reflect which attack steps are possible on the more detailed system. Figure 5-16 shows the modified attack tree based on developing the attacker scenario and possible countermeasures. Firewalls may prevent the attacker from sending messages, and security protocols may prevent the attacker from tampering with communications.

In the next chapter, we describe how we can formally verify the security properties (Confidentiality and Authenticity) of different communications. The formal verification process first requires a model transformation from mapping models into an applied pi-calculus specification.

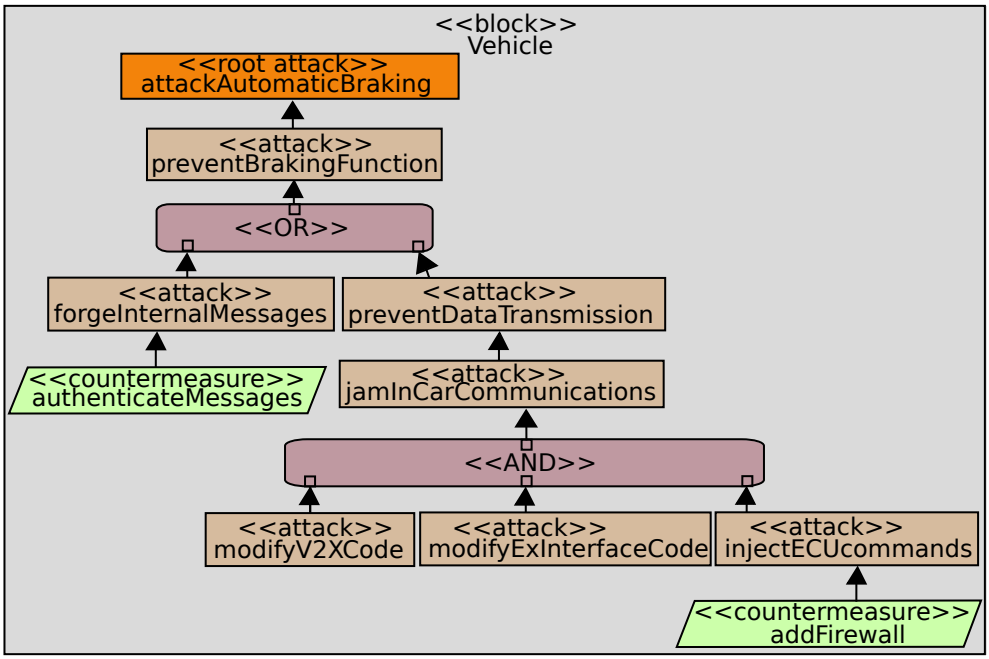


Figure 5-16: Modified Attack Tree with Countermeasures Added

Chapter 6

Security Verification

"The man of science has learned to believe in justification, not by faith, but by verification." –Thomas Huxley

6.1 Introduction

The ultimate security properties of a system should be determined on the final system, as many coding mistakes occur at a level of detail not present in modeling [332]. For example, input validation errors such as buffer overflows, programming errors such as off-by-one array iterations and format string vulnerabilities [140, 319] are implementation errors too detailed to be present in our models. Other errors, such as multi-threading errors and critical section errors, are not due to design errors, and can be difficult to detect even with testing and formal analysis tools. For example, a NASA autonomous spacecraft controller was tested and checked before operation, but one of the programs became deadlocked during the actual operation due to a missing critical section causing a race condition [139].

If code is developed without the use of models, designing models to represent the code to be used with verification may be time-consuming and inaccurate [230]. The abstract model may fail to reflect all of the details within the code, or model the system incorrectly, which may result in misleading verification results. Therefore, certain works formally check source code directly, though they are applied to software-only systems. For example, [282] works on checking the functionality of C code for embedded systems specifically, [249] developed Java Path Finder for checking the functionality and finding errors in java code, and [315] verified that code satisfied CERT secure coding practices. Checking source code however, cannot take into account security details related to architecture, such as Hardware Security Modules.

On the other hand, early detection of design flaws helps to prevent costly rework [327]. As described in the previous chapter, if flaws cannot be fixed by a software patch alone, then hardware components may need to be replaced in the system, which requires the purchase of additional components. If the product has already been sold commercially, then a mass recall may be necessary. These are costly situations which should be ideally avoided. In embedded systems, formal verification of code alone may not be sufficient as it fails to take into account the hardware components [91, 193]. Furthermore, models can be more adapted

to formal methods, as they are less complex, and thus their formal specifications are sufficiently small enough to avoid state explosion problems [332].

In the previous chapter, we described how to abstractly model the mapping of a system to express how it can be targeted by an attacker and the countermeasures that it may deploy to resist those attacks. Next, a security verification process should be used to confirm that our added countermeasures are sufficient. The security properties that we need to verify in embedded systems, as described in Section 2.5, check if important data will be kept secret from an attacker (Confidentiality), and if important data cannot be forged by an attacker (Authenticity). Furthermore, to verify that the protocols are able to execute correctly and complete (decrypt the received message), we should check the Reachability of states.

Using automated formal verification tools instead of checking a design by hand should be more reliable, as computers do not make human errors, and also more convenient as the results are returned to the user faster than if the proof was performed by hand. Formal verification tools, however, tend to be written in mathematical languages, and not in a modeling language. One important aspect of model checking is therefore to ensure that the model transformation into a formal specification is indeed correct, for verifying a formal specification that does not match the modeled system does not help us check the correctness of the model, and may actually cause us to miss actual flaws.

Formal verification methods each have their own advantages and disadvantages. Some are semi-automated, which use manual interaction to help, while others are entirely automated. One of them, ProVerif, is a automatic prover operating on a specification in applied pi-calculus and Horn Clauses, based on the Dolev-Yao attacker model [34], which, as described in the previous chapter, is the attacker model we use for our models. ProVerif is able to verify the properties important to us: Confidentiality, Authenticity, and Reachability. Its advantages are that it is suited for embedded systems as ProVerif assumes that the attacker spies on communications sent along buses, instead of within components, which is the attacker model we currently use [208]. In addition, ProVerif allows the definition of custom elements such as clauses and functions, requires no manual work by the user in terms of writing proofs or calculating properties, and does not limit the number of states explored. It also has a proven record, as it has been used successfully for many other security-critical projects [23, 103, 224]. For all of these reasons, the developers of TTool selected ProVerif as the security verification tool.

With the verification results returned by ProVerif, the designer can then add the security mechanisms described in the previous chapter. In larger systems, however, it may also be tedious to manually add all of these operators. It would be therefore helpful for the toolkit to automatically generate a new model with all the security flaws fixed, especially if the designer is not a security expert.

This chapter describes multiple aspects related to security verification. First, we describe the formal verification language we use, ProVerif. Next, we formalize the model and automatic model transformation process. We then use a proof by induction to verify that our model transformation process preserves the security property of Confidentiality.

The next part of the chapter then discusses the ProVerif verification results. We describe how the results from the prover are interpreted automatically, and backtraced to the model for user convenience. Next, we describe how the verification results can be used to automatically generate a secured modeling fulfilling all of the security properties required.

Manually adding all of the security operators can be tedious for a designer. In the interest of convenience, it is possible to automatically modify a HW/SW Partitioning model so that it fulfills all of the required

security properties. At a high level, we know that if a confidential message is exchanged, it must have been encrypted before being sent, and then decrypted after reception in order to understand the contents of the message. Adding these abstract notions of security protocols automatically thus generates a model of the system secure against the assumed attacks. This auto-generated secure model is used as a base model that can then be further refined into a final HW/SW Partitioning mapping model describing both the high-level functional behavior and architecture, including both hardware and software-based security countermeasures.

6.2 ProVerif

ProVerif assumes a Dolev-Yao attacker, which is a threat model in which the attacker can read or write on any public channel, create new messages or apply known cryptographic primitives. In other words, once the attacker has intercepted a message, he behaves as an adversary with a knowledge of basic cryptography, who can perform calculations and message injections towards deciphering the message. For example, the attacker can recover the decrypted message if he/she can also intercept a key. The attacker can also send messages along public channels within the system, such as to impersonate a legitimate actor to start a key exchange in his/her efforts to recover the key.

The ProVerif prover operates on a ProVerif text specification. A ProVerif specification consists of the description of a set of processes communicating on public channels as a list of actions of the different processes, queries indicating the list of properties to verify, and declarations defining functions, channels, constants, etc. Given a specification, the prover performs calculations to examine all possible execution paths and possible attacker attempts to access sensitive data, inject messages, etc, and then returns the results indicating if security properties queried (Confidentiality, Authenticity, Reachability) are satisfied or violated.

The definitions in this section are used to help explain how HW/SW Partitioning models are translated into a ProVerif specification, and how insecure communications in the models can be detected in the prover. An excerpt of the appearance of the main components are shown here:

6.2.1 Functions

Before a function can be used in the behavior specification of a system, it first must be declared. The declaration describes what arguments the function takes in as input and output. Also, the declaration describes relationships between functions, or if there are any effects from combining functions, for example between a corresponding encryption and decryption function.

Functions are defined for the common cryptographic primitives such as symmetric encryption, symmetric decryption, etc., and modeled as constructor/destructor functions.

For example, here, the symmetric encryption *sencrypt* and symmetric decryption *sdecrypt* functions are defined. For example, *sencrypt* takes in as input two bitstrings, and outputs a bitstring. Then, in addition, to indicate to the prover the relation between symmetric encryption and symmetric decryption, the line starting *reduc forall x* explains that encrypting x with *sencrypt* and then decrypting the encrypted message with *sdecrypt* results in recovering the original data x.

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

6.2.2 Declarations

Declarations specify each channel and variable which will be used in the system. Like functions, they must also be declared before use. The following specifies one channel *ch* and one variable X, which is a bitstring and private (unknown by the attacker) at the start of the system execution. The channel *ch* is a public channel on which communications can be sent. Since *ch* is a public channel, the attacker has complete (read/write) control on this channel.

```
(* Channel *)
free ch: channel.

(* Variables *)
free X__0: bitstring [private].
```

Another channel, *chControl* is a private channel (inaccessible to the attacker), which subprocesses use to signal the start of another subprocess. We will show examples using *chControl* later in this section.

```
(* Control Channel *)
free chControl: channel.
```

6.2.3 Queries

Queries indicate to the prover which security properties, confidentiality, authenticity, and reachability, should be verified. Without queries, the specification would not return any results, as it would not know which attributes, message exchanges, or events needed to be checked.

'Queries Secret' check that an attribute is confidential, meaning that an attacker cannot determine the value of that attribute. 'Queries Event' check the reachability of states. Annotations are placed in a subprocess which correspond to the start of a state as we show later. If the prover finds a trace that 'reaches' that annotation, it determines that the corresponding state is reachable. 'Authenticity' queries check the strong and weak authenticity of data between two states, to confirm that if the process reaches the latter event, then the data in that event must be the same as the data in the earlier event.

The example below shows 1) a query for the Confidentiality of data X, 2) a query to check that the protocol will enter the state 'state' of task T, and 3) a query for the authenticity of data X: that if data X was received by Task T2 at the state 'msgVerified', then it was the same X sent by Task T1 at the state 'sendingMsg'.

```
(* Queries Secret *)
query attacker (new X)
```

```
(* Queries Event *)
query event(enteringState___T__state()).
event enteringState___T__state().

(* Authenticity *)
event authenticity___T1___X___sendingMsg(bitstring).
event authenticity___T2___X___msgVerified(bitstring).
query dummyM: bitstring;
inj-event(authenticity___T2___X___msgVerified (dummyM)) ==>
inj-event(authenticity___T1___X___sendingMsg (dummyM)).
```

6.2.4 Main Process

The main process is the process describing the actions of the complete system. It is the only process started as the beginning of system execution, and starts the other subprocesses. For example, if the system contains two blocks T1 and T2, the main process would be written as shown here, where the main process declares a new session ID, and then starts each task in parallel.

```
process
! (
new sessionID[]: bitstring;
((
T1___0 (sessionID)
) | (
T2___0 (sessionID)
) | (
(
((
T1___start (sessionID)
) | (
T2___start (sessionID)
)))
)))
```

6.2.5 Sub-processes

Sub-processes describe a sub-behavior of a single task in the system. The behavior of a single task is often split into multiple sub-processes, such as T1__start, T1__0, T__1, and etc. In this example, the starting behavior of task T1 is shown in T1__start. After the main process signals that T1__start can begin execution, the task will perform some operations, for example entering the state ‘state1’ (used for reachability queries as described previously), and then indicate that subprocess T1__0 should start. Signaling the start of a sub-process (a sub-process can also start itself) involves sending data such as attributes and the session id along the channel *chControl* as shown here.

```
let T1___start (sessionID: bitstring) =
..
```

```
event enteringState___T1__state1();
out (chControl, chControlEnc ((sessionID, call___T1___0))).


let T1___0 (sessionID: bitstring) =
...
```

6.2.6 Formalizations

These different aspects of a ProVerif specification described above are next described in greater detail and formalized mathematically.

Based on the work in [208], we define the following abbreviations to be used in our formalization. These will also be used to describe the translation and proof.

- Processes: *P,Q* are sub-processes or the main process
- Message: *M* can be sent along channels
- Variables: *X,Y, etc* on which functions can operate, or which can be the value returned by a function
- Public Channel: *ch*, the public channel for communications
- Control Channel: *chControl*, which is a private channel used to control which process is executed next
- Function: *func*, which operate on variables and received messages

Each process is composed of possible actions:

- **Constructor functions** of the form *funcX,Y*
- **Destructor functions** of the form *X=funcY*
- **Write channel operations** of the form *out (ch, M)* or *out (privChannel, M)*
- **Read channel operations** of the form *in (ch, M)* or *in (privChannel, M)*
- **Process controls** which trigger the start of a process, of the form *out (chControl, chControlEnc((sessionID, P, X, Y))*
- **Conditionals** which split a process into different actions, in the form *if condition then P else Q*.

6.2.7 DIPLODOCUS to ProVerif Translation Process

As described in Chapter 4, our methodology and toolkit supports 2 design phases, each within its own design environment. The set of HW/SW Partitioning (or Mapping) models are contained in the DIPLODOCUS environment, and the set of Software Design models are contained within the AVATAR environment.

DIPLODOCUS and AVATAR models differ in many ways, but as we explain in this chapter, DIPLODOCUS models can be translated to AVATAR models without altering the security properties in the model.

The next sections describe how HW/SW Partitioning (DIPLODOCUS) models are translated into ProVerif specifications described above, in a multi-step process shown in Figure 6-1. First, a complete DIPLODOCUS design is translated to an AVATAR model [201], and then, the AVATAR model is in turn translated into a ProVerif specification. The AVATAR-to-ProVerif transformation is described in detail in [209].

To summarize the AVATAR-to-ProVerif transformation, first, each state machine is split into multiple basic blocks that are each translated to a ProVerif process. Avatar concepts are then translated into their ProVerif counterpart. Some of the concepts have no counterparts in ProVerif so the translation must use some workarounds (for loops for instance), or discard AVATAR concepts that ProVerif can not handle (such as time). Processes are linked together by using control *tokens* that signal the start of a sub-process.

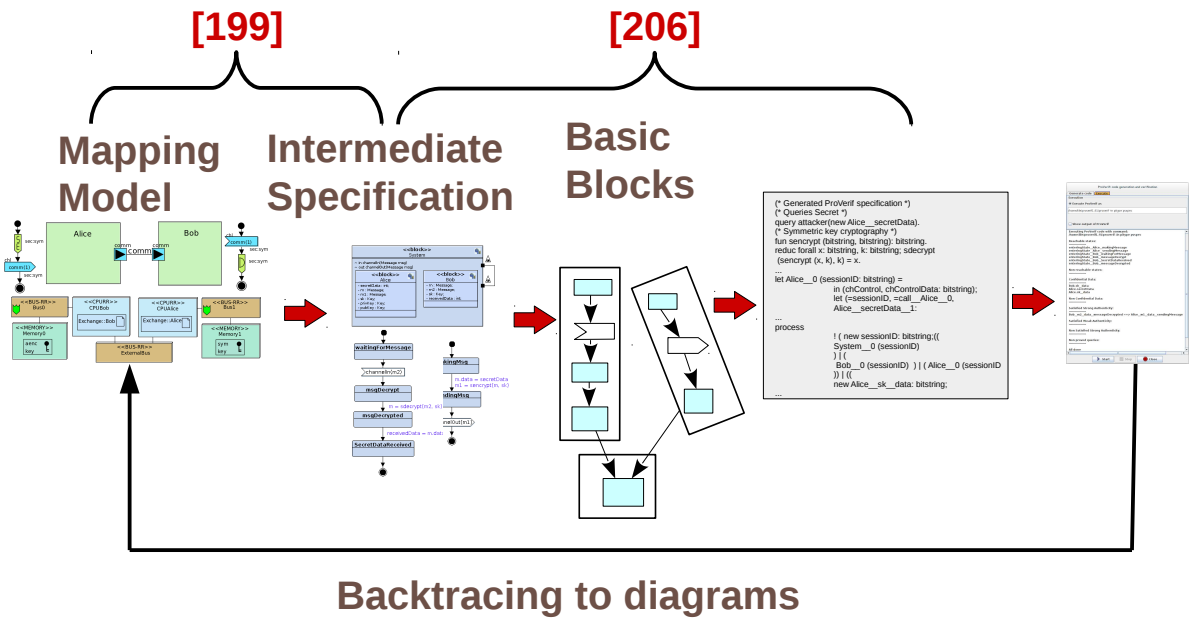


Figure 6-1: DIPLODOCUS to ProVerif Translation process

To describe the DIPLODOCUS->AVATAR translation in greater detail, it is first necessary to formalize the different elements of each. This same translation is used to automatically generate preliminary Software Design models from Mapping models.

6.3 Formalization for Translation

We formalize our models in three parts: 1) DIPLODOCUS Mapping models 2) AVATAR Software Design models 3) DIPLODOCUS to AVATAR translation. By formalizing our models and translation process, we can subsequently prove the correctness of our translation, and also thus explain how a ProVerif specification can be obtained from a DIPLODOCUS Mapping Model.

6.3.1 DIPLODOCUS Formalization

As previously described in Chapter 4, HW/SW Partitioning models include Functional/Application models describing the high-level functionality of the system, Architecture models showing the architecture of the system, and Mapping models, usually enhancements of the architecture models, mapping the tasks and communications in the Functional models onto the Architecture models. A Functional model contains tasks (or functions), and each task has a behavior, which can include basic operations such as computation and communication, and more complex behaviors involving loops, choices, etc. These concepts are described formally below.

A Partitioning P is defined as a set of models $P = (FM, AM, MM)$, with FM a Functional Model, AM an Architecture Model, and MM a Mapping Model.

6.3.1.1 Functional Level

A Functional Model is defined as $FM = (T, Comm)$ where T is a set of Tasks, and $Comm$ is a set of Communications between tasks. A Task t_p is defined as $t_p = (Attr, B)$ with $Attr$ a set of Attributes, and B a behavior.

The Behavior $B = (Ctrl, CommOp, CompOp)$ consists of:

- Control Operators $Ctrl$ – such as loops, choices, etc.
- Communication Operators $CommOp$ – channel read/write, events send/receive, request send/receive
- Complexity operations $CompOp$, who model the complexity of algorithms through the description of a min/max interval of integer/float/custom operations.

Figure 6-2, 6-3, 6-4, and 6-5 show all of the behavior elements along with their formalization.

Among the $CommOp$, we distinguish between channel read/write operators, which we call Data Communication Operators, and send/receive events or requests, which we designate 'Control Communication Operators'. Data Communication Operators send an amount of data, where Control Communication Operators are used for synchronization between tasks and may be used to send/receive attribute values to be used in control operators. For example, an event or request may send the number of times a task should execute a loop. Data Communication Operators are defined by a function $commOp(name, IN/OUT, size)$, where Control Communication Operators are defined by a function $commOp(name, IN/OUT, [attr1, attr2, ...])$, as shown in Figure 6-2.

Control operators $Ctrl$ operate on a sub-behavior $subB$, defined as a subset of the operators of a behavior B. Control operators may be simplified to a *loop* function or $choice$ function.

A choice function $choice([subB1, subB2...], [cond1, cond2, ...])$ takes a set of $subB$ and a set of conditions $cond$, where each sub-behavior $subB$ is a possible option only if its corresponding condition $cond$ evaluates to true. If more than one guard evaluates to true, then during execution, the next operator to be executed will be chosen non-deterministically among all of the corresponding sub-behaviors. Choice functions can be non-deterministic or deterministic, depending if there is only one possible outgoing sub-behavior or one guard condition evaluating to true. For example, select events, choices, and sequences are

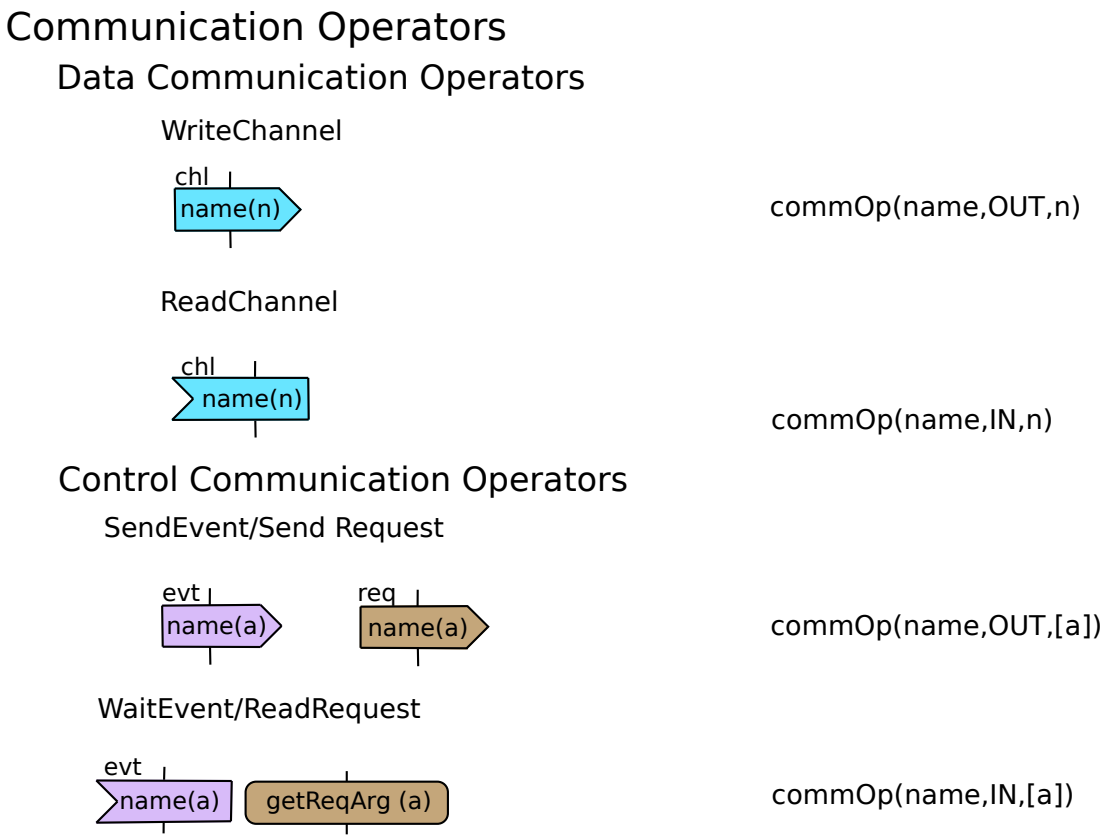
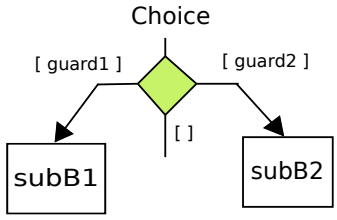


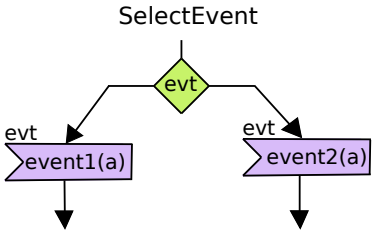
Figure 6-2: Functional Model Communication Behavior Formalization

Control Operators

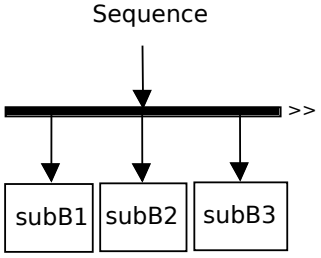
Choice Control Operators



choice([subB1, subB2],[guard1, guard2])



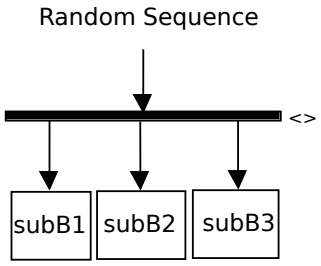
choice([commOp(event1, IN, [a]),commOp(event2,IN, [a])],
[event1 executed?, event2 executed?])



choice([subB1],[true])

next(subB1) = choice([subB2],[true])

next(subB2)=choice([subB3],[true])



choice([subB1,subB2,subB3],[true,true,true])

next(subB1) = choice([$subB2^1$,$subB3^1$],[true,true])

$next(subB2^1) = subB3^{12}$

$next(subB3^1) = subB2^{13}$

next(subB2) = choice([$subB1^2$,$subB3^2$],[true,true])

$next(subB1^2) = subB3^{21}$

$next(subB3^2) = subB1^{23}$

next(subB3) = choice([$subB1^3$,$subB2^3$],[true,true])

$next(subB1^3) = subB2^{31}$

$next(subB2^3) = subB1^{32}$

Figure 6-3: Functional Model Choice Behavior Formalization

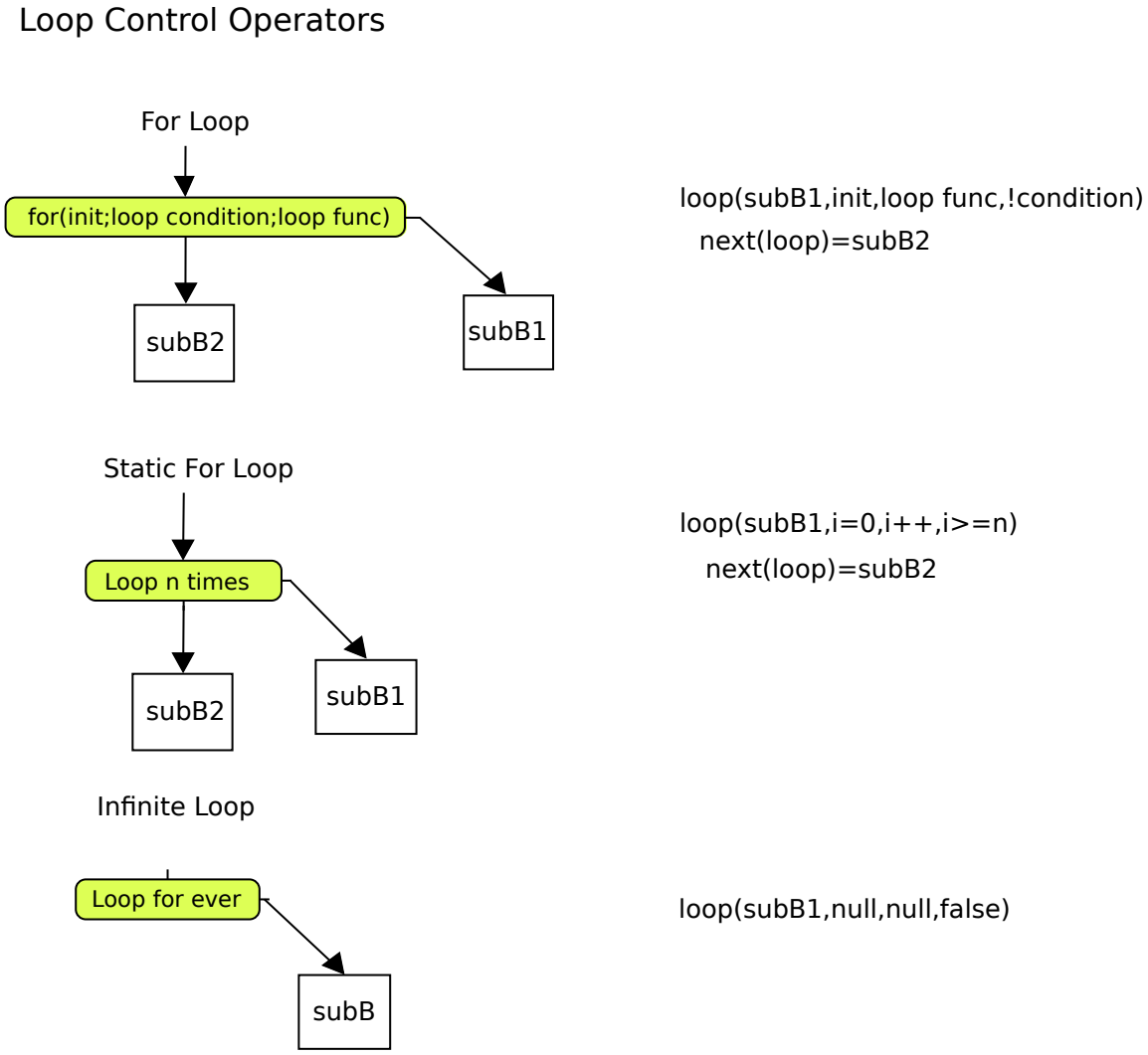


Figure 6-4: Functional Model Loop Behavior Formalization

considered choice functions, as shown in Figure 6-3. For example, a non-random sequence is translated into a series of operators, with first a choice of only taking the first sub-behavior, then next a choice of only taking the second sub-behavior etc. A random sequence is a choice of taking any of the possible sub-behaviors, and then a choice of taking any of the remaining sub-behaviors.

A $loop(subB, init\_func, loop\_func, exit\_cond)$ function takes as input a sub-behavior that it will execute a number of times. At the start of the loop, it executes the initialization function $init\_func$, and on each loop iteration, it performs a function (increment, decrement, etc) $loop\_func$ after each execution of $subB$. After each execution of the loop, the exit condition $exit\_cond$ is checked. If $exit\_cond$ evaluates to false, then the loop repeats: $subB$ and $loop\_func$ is executed again, and then the $exit\_cond$ is checked again. The formalization of the different loops (regular for loop, static for loop, infinite loop) are shown in Figure 6-4.

A regular for loop follows the format with an $init\_func$, $loop\_func$, and $exit\_cond$ where these functions and conditions may be any function or condition. A static for loop must iterate a fixed number of times n, where n is a positive integer and finite. An infinite loop may have any $init\_func$ and $loop\_func$, but the $exit\_cond$ must always evaluate to false.

Complexity Operators $CompOp$ occupy either a given computational complexity (as a function of cycles), or a fixed time delay (in seconds) and may include a function f operating on the attributes of the task, expressed as $compOp(n, Complexity, f(attr1, attr2, ...))$ and $compOp(n, Delay, f(attr1, attr2, ...))$ respectively. Some model operators perform only a function and occupy neither a computational complexity nor a delay, such as random or action states. All of the different operators thus classified are shown in Figure 6-5. Overheads and sizes of messages for the security operators may be taken into account in the calculation of the complexity, and also in the time for these messages to be sent (if the message size is changed after the operation). The exact complexity of security operators and communications is discussed in greater detail in Section 7.4.

6.3.1.2 Mapping Level

Mapping involves allocating tasks onto the architectural model. A task mapped to a processor will be implemented in software, while a task mapped to a hardware accelerator will be implemented in hardware.

The architectural model is a graph of execution nodes (CPUs, Hardware Accelerators), communication nodes (Buses and Bridges), and storage nodes (Memories). Hardware components are highly abstracted: a CPU is defined as a set of parameters such as an average cache-miss ratio, go idle time, context switch penalty, etc.

An Architecture Model

$$AM = (CommNode, StoreNode, ExecNode, link)$$

is built upon abstract Hardware Components: Communication Nodes $CommNode$, Storage Nodes $StoreNode$, Execution Nodes $ExecNode$, and architectural links between Communication Nodes and any other node $link$. $ExecNode$ defines a conversion from Complexities to Cycles, and an execution frequency converting Cycles to seconds, while the times in delay-type Complexity operators are not converted since they were already expressed in physical time. Similarly, $CommNode$ and $StoreNode$ also define a conversion from functional operations (write data, store data, etc) to physical time.

Figure 6-5: Functional Model Complexity Behavior Formalization

6.3.2 AVATAR Formalization

The Software Model $S = (T_S, Comm_S)$ can also be defined as a set of Tasks T_S and Communications $Comm_S$ between tasks. However, the Software Design tasks, behavior, and communications will differ from their corresponding HW/SW Partitioning elements for the same model as we describe next. Overall, HW/SW Partitioning (DIPLODOCUS) models are more abstract, and contain fewer functional details than Software Design (AVATAR) models. On the other hand, HW/SW Partitioning models contain architectural information that is not shown in the AVATAR models.

A Software Task t_s, similar to Partitioning tasks, contains attributes and a behavior diagram, where $t_s = (attr', BD\_S)$. Regarding the behavior of tasks, functional models in the HW/SW Partitioning phase express algorithms as an abstract complexity operation and communications in terms of the size of the exchanged data, where Software Design models describe the implementation of algorithms with a sub-behavior description using attributes, and communications (based on signal exchanges) contain exchanged values. Thus, the set of attributes of software tasks is likely to be enriched both with regards to the partitioning model for algorithms details and communication details.

The communications in AVATAR may be one-to-one, many-to-one, one-to-many, or many-to-many between software tasks. As the set of Software Tasks T_S may be different from the set of Partitioning tasks T, the communications between tasks may also be different, so while $Comm_S$ and $Comm$ should be similar, they are not necessarily identical. For example, if $T1$ and $T2$ exchange data in Partitioning models, but $T1$ is mapped to a Hardware accelerator, then $T1$ will not be present in T_S, and their communication is also present in $Comm$ but not $Comm_S$. It could also be that one task in T is split into multiple tasks in T_S, and the inter-task communications will be added to $Comm_S$.

Software Design functional models contain 'states' which do not exist in HW/SW Partitioning functional models. The state machine diagrams representing the behavior in Software Design models may contain States, designated $state(name, function)$ which may be followed by any number of sub-behaviors. Action states such as random operators may perform a function, such as assigning a random value to an attribute, while simple states perform no operation.

Communication Operators in Software Design no longer distinguish between Control and Data Communications. All communication operators are expressed as $commOp(name, IN/OUT, [attr1, attr2, etc])$, where they are defined by a name, a direction (in or out), and a list of attributes to be sent or received (which may be empty).

For Control Operators, there is no longer an explicit Loop operator, though a transition may point to any state, including itself, so that a loop forms in which $state1 = next(next(...(state1))$.

Choice operators exist as an explicit operator, but they can be formally simplified as a state followed by multiple transitions. We express any such situation with multiple transitions leaving an operator to be $next(state) = (tran1, tran2, etc)$.

While transitions in DIPLODOCUS models only connect one operator to the next, in AVATAR, transitions $tran$ may contain a time function *afterTF*, guard *guard*, and function $func$ operating on attributes, expressed $tran$(*afterTF*, *guard*, $func([attr1, attr2, ..])$.

The relevant modeling operators of Software Design are shown in Figure 6-6.

It is necessary for us to distinguish between Transitions $tran$ vs all other operators: communication-

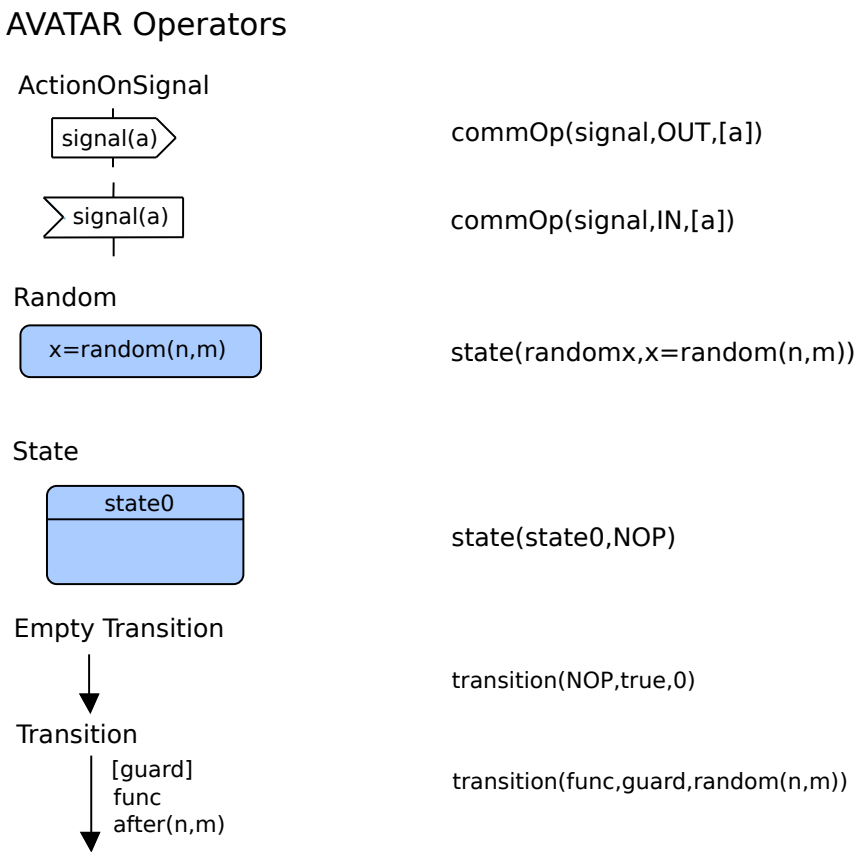


Figure 6-6: Avatar Behavior Formalization

s/states. Communications/states must be followed by a transition, and not another communication/state. Transitions also must be followed by a communication/state, and never another transition. These rules can be expressed as: $tran = next(state)$, $tran = next(comm)$, and $state/comm = next(tran)$.

6.4 DIPLODOCUS to AVATAR Translation Formalization

This section discusses how a DIPLODOCUS Mapping model, including architectural elements, is translated into an equivalent AVATAR model. The architectural elements themselves are not present in Software Design, but some of their properties should be reflected in the AVATAR model, as we discuss in this section. Security Verification is performed only on DIPLODOCUS Mapping models instead of DIPLODOCUS Functional models, for functional models do not model the ability of the attacker to access communications. The mapping to architecture imbues the functional model with vulnerabilities.

There are two possible translations: one which translates DIPLODOCUS Mapping models into their exact AVATAR equivalent, and another which preserves only the equivalence of security properties. It is the latter simplified translation which we use as the first step in the translation to ProVerif, but we first briefly describe the elements present in the full translation only before we describe the translation for security verification.

6.4.1 Full DIPLODOCUS to AVATAR translation

The goal of the full translation is to translate every feature in the DIPLODOCUS model exactly, to be used as a starting AVATAR model for Software Design.

In the full translation to Software Design models, partitioning tasks mapped to processors are to be implemented in software, and are therefore translated to tasks in Software Design. Tasks implemented in hardware are removed, as their design in VHDL for example is supported by other tools. These tasks might be later split or joined manually if desired by the designer, but this preliminary translation is intended to create an equivalent software model.

DIPLODOCUS functional complexity operators may represent an algorithm, or a delay function. The complexity operators, expressed as a function of execution cycles, are translated into either a time function $TF()$ or sub-behavior $subB$ first when mapped upon a processor, which can then be used as the time function in AVATAR. More formally, the transformation relation of functional behavior to software design behavior can be expressed as:

$$B = (CtrlOp, CommOp, CompOp) \rightarrow$$

$$B_S = (CtrlOp', CommOp', TF, subB)$$

The translation from logical Complexities to physical Time Functions depends on the final hardware, approximated as f(n) = n*processor frequency*cycles per computational complexity, assuming a 1-stage pipeline. Complexities and time are ignored in the security proof, but we note this translation as it will be used in the following chapter in Section 7.3, which discusses how complexities in HW/SW Partitioning and physical time in Software Design should relate.

Communications in Mapping are typed 'Non-Blocking Read + Non-Blocking Write' (NBR-NBW), 'Blocking Read + Non-Blocking Write' (BR-NBW), or 'Blocking Read + Blocking Write' (BR-BW). Software Design Communications, however, are typed 'Synchronous' or 'Asynchronous', where Asynchronous communications can be blocking in writing or not. If an infinite FIFO is attached to the asynchronous communication channel, then it never blocks writing, where if a finite FIFO is, then it can block writing. Furthermore, all communication operators in Software Design operate on a single sample, where Read/Write Channel operators may operate on multiple samples. Therefore, we translate each Read/Write Channel(n) into n Read/Write Channel(1) operators.

As none of these qualities (blocking vs non-blocking, synchronous vs asynchronous) are taken into account in ProVerif, for a security translation, we do not take the DIPLODOCUS channel properties into account in our translation for security. However, for the full translation to be used in Model-Checking or generation of a preliminary Software Design model, the channel properties are relevant, so we translate the channel properties to preserve its exact behavior.

6.4.1.1 Translation of Communications

BR-BW channels can be translated into a Asynchronous channel with the same finite sized FIFO used for the communication. BR-NBW channels are similar, expect that they use an infinite FIFO so as to never block writing.

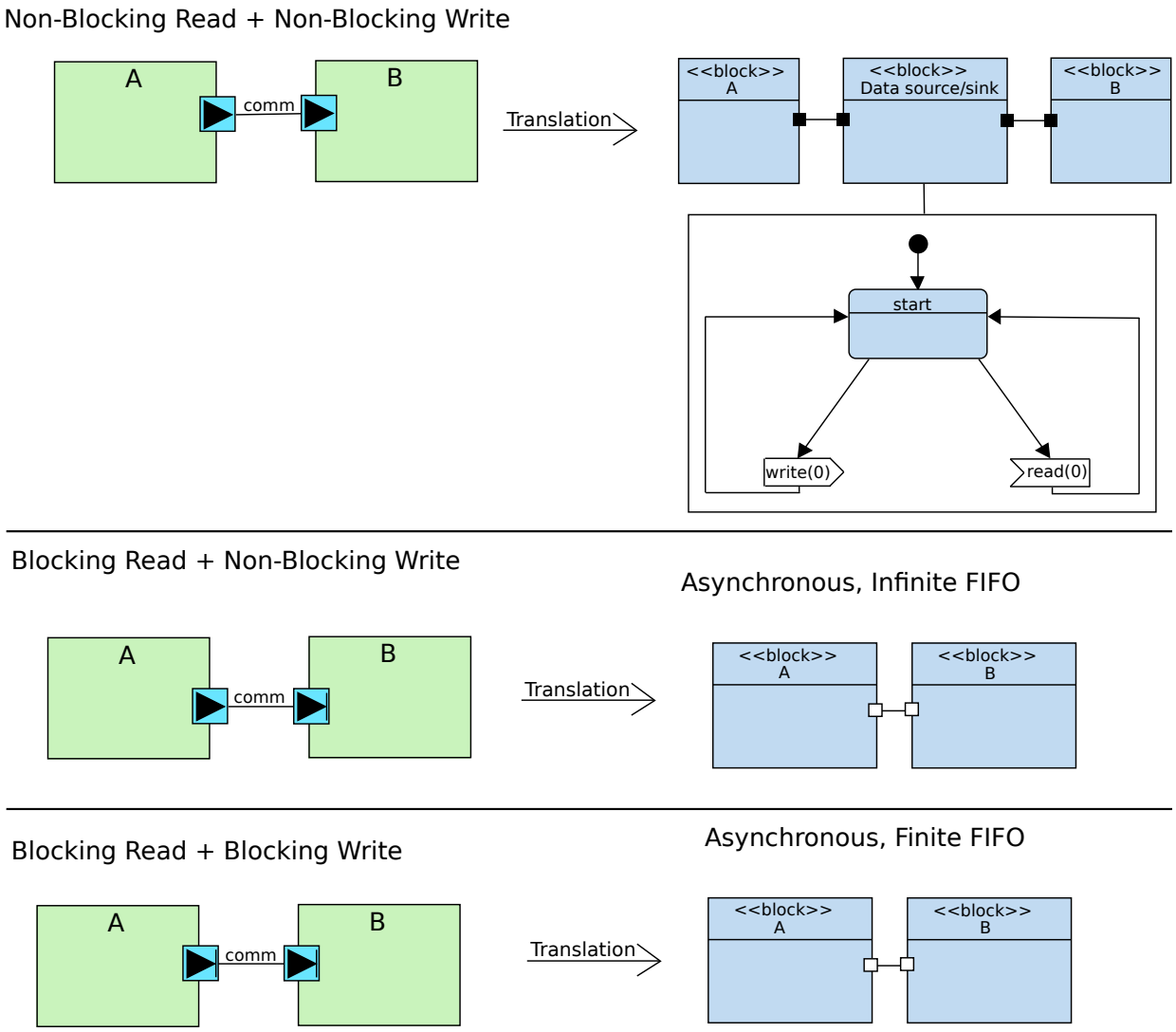


Figure 6-7: Translation of Functional Communications to Software Design Communications

NBR-NBW channels model a data generator in software design. As none of the existing AVATAR Communication types correspond to this channel type, we translate it to contain an intermediate block between the sending and receiving blocks, where the intermediate block is a data source/sink and can always send and receive data. Figure 6-7 shows the translations of the different types of channels.

6.4.2 DIPLODOCUS to AVATAR translation for Security

For security translations, our focus is to check the security properties of critical communications which should be confidential or authentic (such as perception data), and we can ignore the behavior irrelevant to security. For one, we only take into account architecture as it relates to security. All the tasks, regardless of whether they are mapped to be implemented in software or hardware, are translated into AVATAR Software tasks. We need the generated AVATAR model to include the functionality of all hardware-mapped tasks, which may include securing communications, to determine if the overall function of the

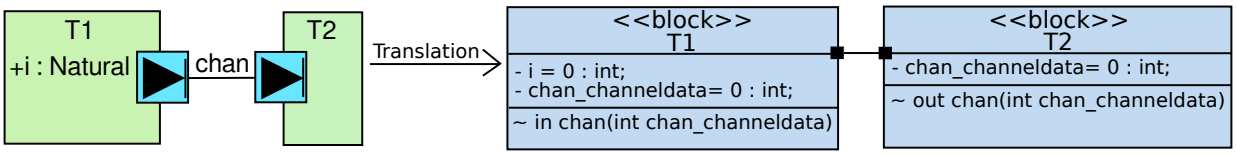


Figure 6-8: Translation of DIPLODOCUS Tasks and Associated Attributes and Communications to AVATAR

system maintains the security properties. For example, even if the full translation would ignore Hardware Security Modules, the translation for security needs to take into account whether they encrypt or decrypt certain data. Therefore, we need to include the behavior of HSMs in the AVATAR model, so that the final ProVerif specification will also take into account the cryptographic operations applied to the data.

The attributes $attr$ in Partitioning task T_P are translated directly into attributes $attr'$ in the corresponding Software Design task T_S. Then, additional attributes are added for each data channel, as Partitioning models do not describe the actual attributes in the communication, while Software Design models describe explicitly the exact data sent. For each channel c, an attribute $c\_channeldata$ is added to the list of attributes for the sending and receiving tasks. Software communications may send multiple attributes together in a single communication, so these attributes may be renamed and split into new attributes as the design is refined.

In addition, new attributes cc and possibly $cc\_encrypted$, are added for each Cryptographic Configuration. Multiple Cryptographic Configurations can be sent along the same channel, so to keep them separate and verify the security of each, we create a new attribute for each, perform the cryptographic operations on it, and then send either the secured or unsecured form along the channel. The exact steps are described in more detail in the translation of Cryptographic Configurations to AVATAR.

Figure 6-8 shows how tasks, and their communications and attributes, are translated from DIPLODOCUS Mapping models to AVATAR Software Design models. Next, we describe how the security properties of these communications are translated.

6.4.2.1 Security of Communications

The security of a communication between tasks in HW/SW Partitioning (DIPLODOCUS) depends on the architecture. As memories are not present in Software Design, the data within the memories are instead present within the attributes of each Software Design Task.

In Software Design, communications between two tasks can be either ‘Public’ or ‘Private’. The hardware on which the communication path is not taken into account in Software Design: either the attacker can access the communication or not, but we do not concern ourselves with the hardware-basis behind this vulnerability. Hardware is modeled at the Software Design phase, so if a communication traversed a specific bus accessible to the attacker, then that communication is simply modeled ‘Public’. On the other hand, in the HW/SW Partitioning phase, architectural details are still relevant. These architecture details must be simplified in order to convert DIPLODOCUS mapping models to AVATAR Software Design models.

For translation to AVATAR, DIPLODOCUS Channels should be classified as ‘Public’ or ‘Private’ based on the properties of the communication path –the set of buses, bridges, and memories, between the two tasks.

If any of the communication nodes along the mapped path are accessible to the attacker, then the attacker can access the data being sent along the path, and the translated channel is considered 'Public'. Otherwise, if all communication nodes are secure, then the channel is considered 'Private'. The architectural components which are traversed are determined as follows.

By default, the communication between tasks is stored on the memory closest to the sending task. The receiving task then reads the communication stored on the memory from the shortest path. The shortest path is defined to be the path crossing the fewest number of architecture components. If there are multiple shortest paths, then the path is selected randomly from among all of the shortest paths.

If however, the communications are mapped to a specific path, then the communications follow that path regardless if there exist shorter ones. In a single architecture and mapping of tasks, the same channel can be translated to a 'Private' or 'Public' communication. For example, Figure 6-9 shows a mapping where there exists a channel 'chData' between Tasks T1 and T2. By the default communication mapping, data is stored in Memory0 by T1 along a secure bus Bus0. Task T2 then accesses the data also along a secure path across Private Bus Bus0. Therefore, the communication is considered 'Private' in AVATAR.

Figure 6-10 shows how instead channel 'chData' is mapped to a different memory and different communication path. In this instance, data is stored in Memory1 by T1 by sending the data along an insecure path through Bus2, Bridge0, and Bus1, all of which are public, and therefore accessible to the attacker. Then, Task T2 also accesses the data in Memory1 along the indicated path through Bus1, Bridge 0, Bus 2, Bridge 1, then Bus 3, again all of which are public. Therefore, the communication is considered 'Public' in AVATAR, as indicated by the attacker symbol on the communication link.

6.4.3 Translation of Operators

Based on the formalization of Partitioning and Software Design Models, we generate the equivalent translation for each operator as described in this subsection. In these diagrams, for the formalization, we omit $next$ operators and assume operations execute in order, unless explicitly stated otherwise for cases of multiple possible $next$ operations, indicated with a choice operator.

As shown in Figure 6-11, Control and Data communications in Partitioning are translated into the equivalent communication operator in Software Design. A new attribute $chan\_channeldata$ is created during the translation for Data communication operators, while the set of attributes sent remains the same for Control communication operators.

Figure 6-12 shows how choice operators are translated. The choice operator in Partitioning (where the next operation is one of multiple possible sub-behavior) is translated into a state, and then a choice: the next operation can be one of two transitions, and then each of those transitions are followed by the translation of each of the sub-behaviors in the choice. The guards of the choice operator are then expressed as the guards on the transitions.

Loop operators are translated into a sequence of states and transitions as shown in Figure 6-13. The choice to exit the loop or not is translated to a choice operator.

Complexity operators are translated into a state and then a transition with a function and time function 6-14. The time function is a translation of the complexities as described above.

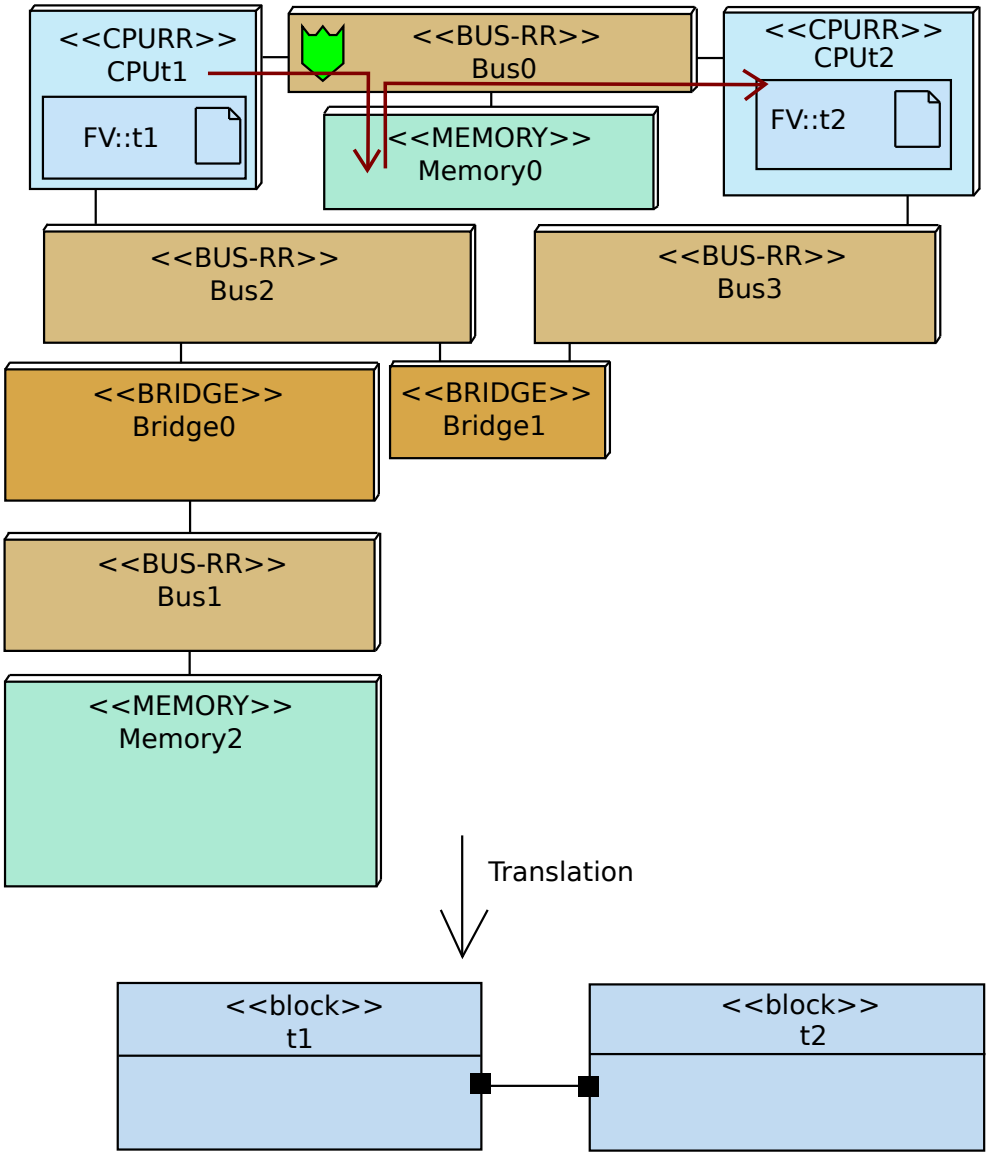


Figure 6-9: Translation of Security of Channels - Secure Communication Mapping

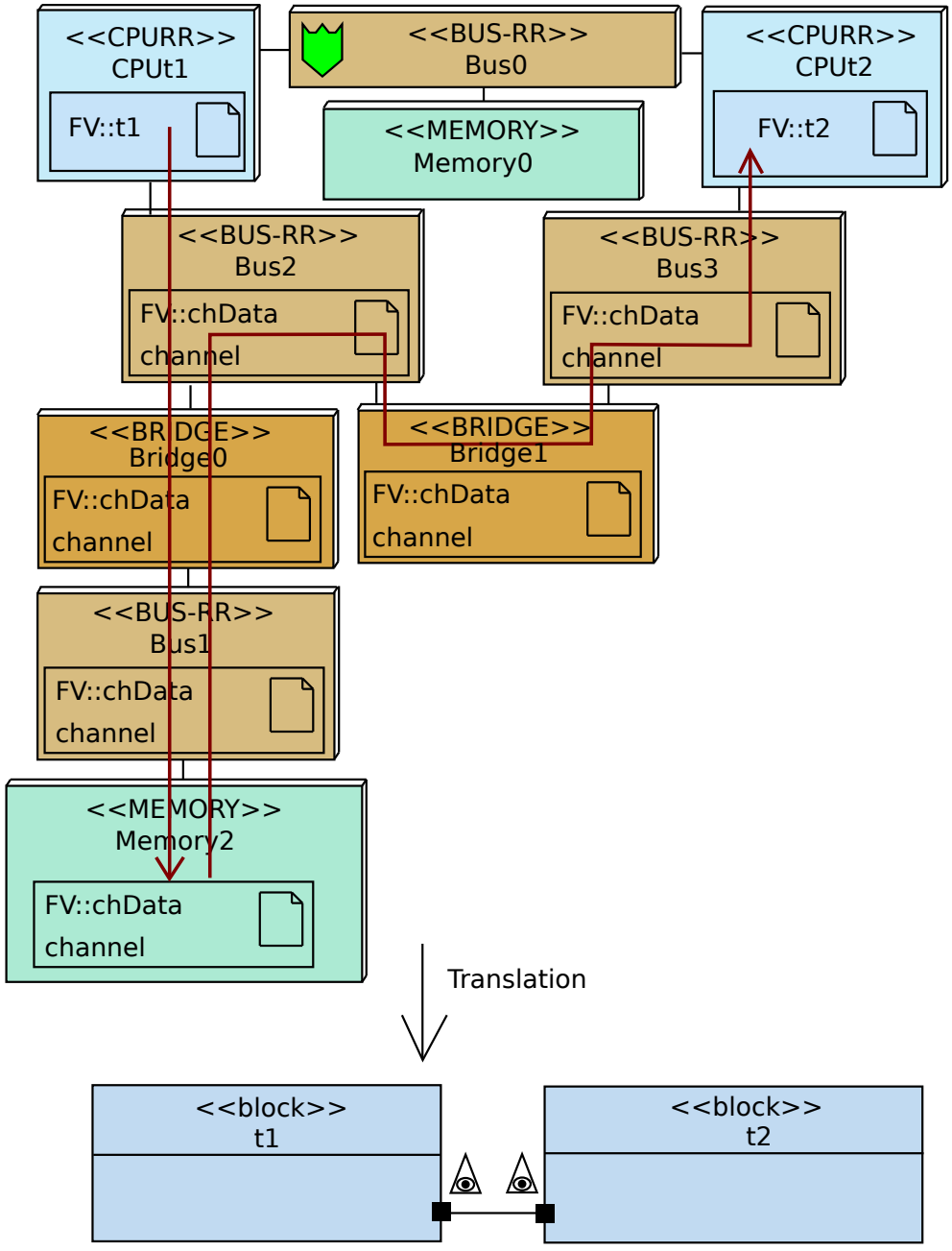


Figure 6-10: Translation of Security of Channels - Insecure Communication Mapping

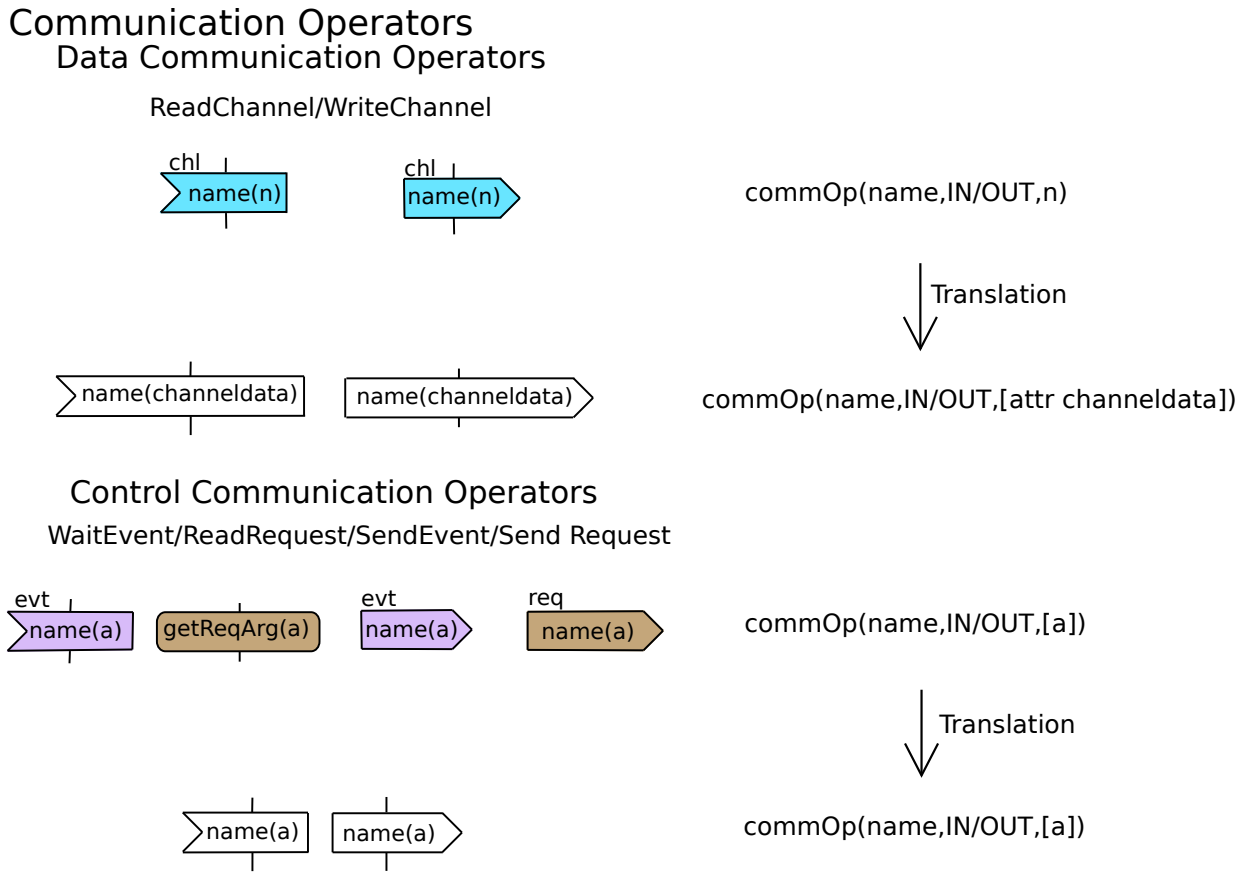


Figure 6-11: Translation of Functional Communication Behavior Elements to Software Design Behavior Elements

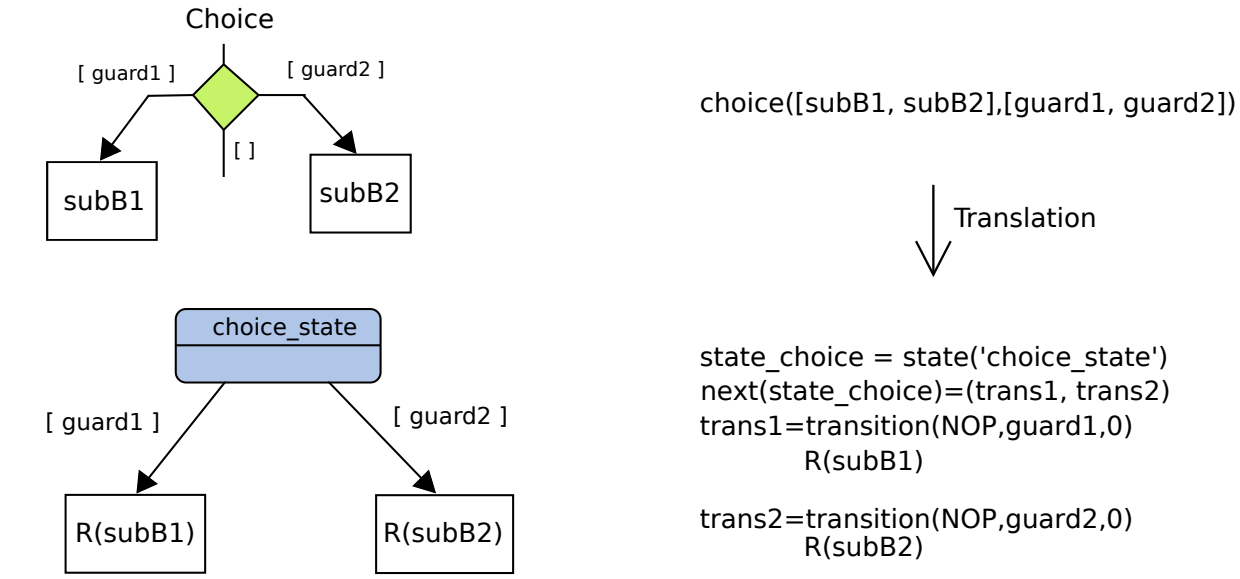


Figure 6-12: Translation of Functional Choice Behavior Elements to Software Design Behavior Elements

6.4.3.1 Translation of Cryptographic Configurations

The detailed translation for each cryptographic operator is shown in 6-15, 6-16, and 6-17.

To explain in greater detail, we examine the translations to AVATAR for all possible situations of sending data secured with a symmetric encryption Cryptographic Configuration shown in 6-15. While the cryptographic operations are not explicitly shown in DIPLODOCUS models, the exact operations are shown in transitions in the AVATAR model.

Basic data encryption

The first case in the figure shows the simple case of encryption and decryption of data without a nonce. As we previously explained, as this instance of the write channel operator is tagged with a Cryptographic Configuration, the data being sent out on the channel $chan$ will be an encrypted form of the new attribute sym, as named after the Cryptographic Configuration, instead of $chan_channeldata$. Queries will also check for if the attacker can recover sym. As shown, the equivalent model in AVATAR involves first encrypting sym with the symmetric encryption function $sencrypt$, using the key key_sym, expressed as $sym\_encrypted = sencrypt(sym, key\_sym)$, and then sending out $sym\_encrypted$ on the channel.

To decrypt the message, the attribute $sym\_encrypted$ is read by the read channel operator, and then the value of sym can be recovered by the symmetric decryption function $sym = sym\_encrypted, sym_key)$, assuming the receiving task has access to the key. If it does not have key_sym, then sym is not recoverable.

Data encryption with nonce

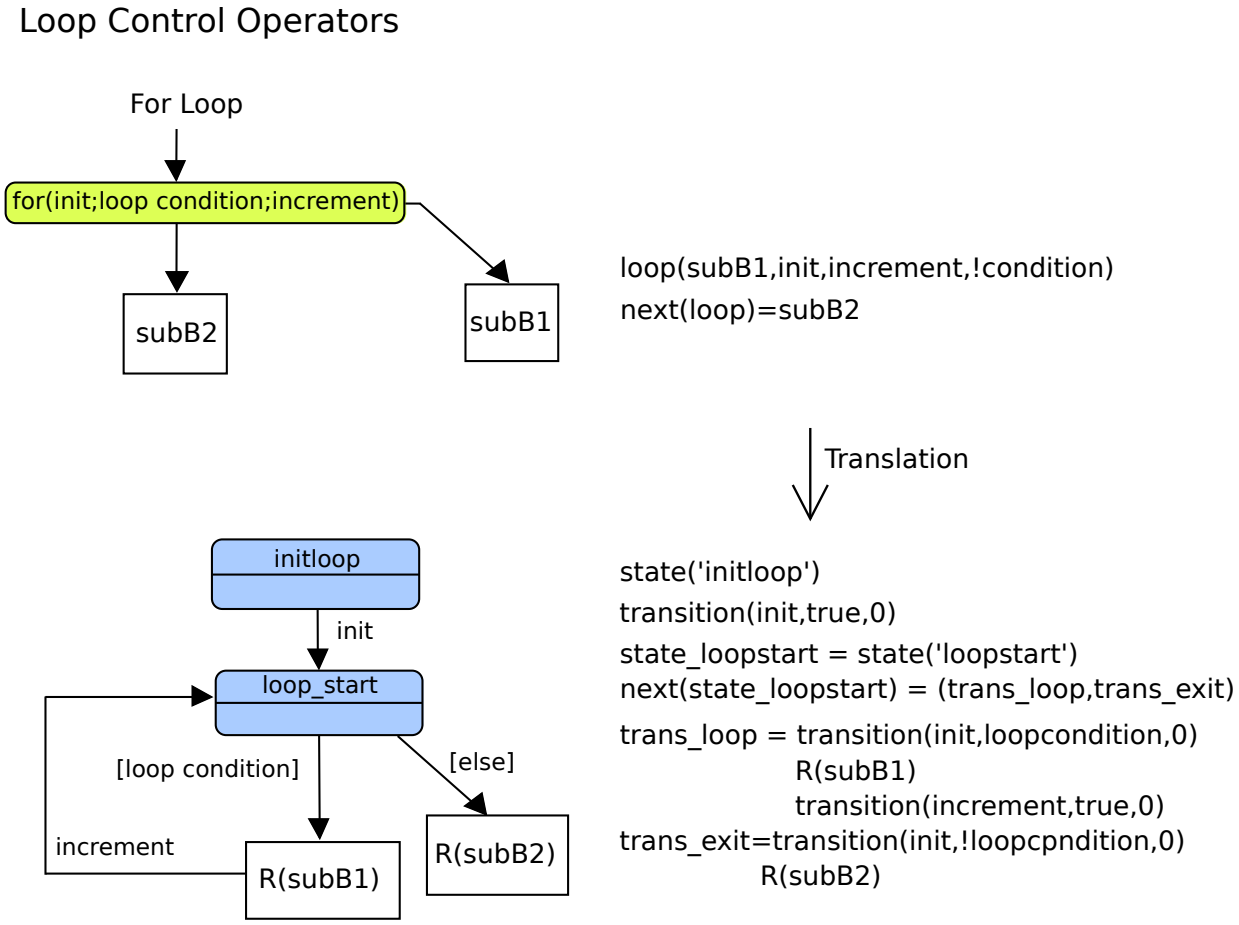


Figure 6-13: Translation of Functional Loop Behavior Elements to Software Design Behavior Elements

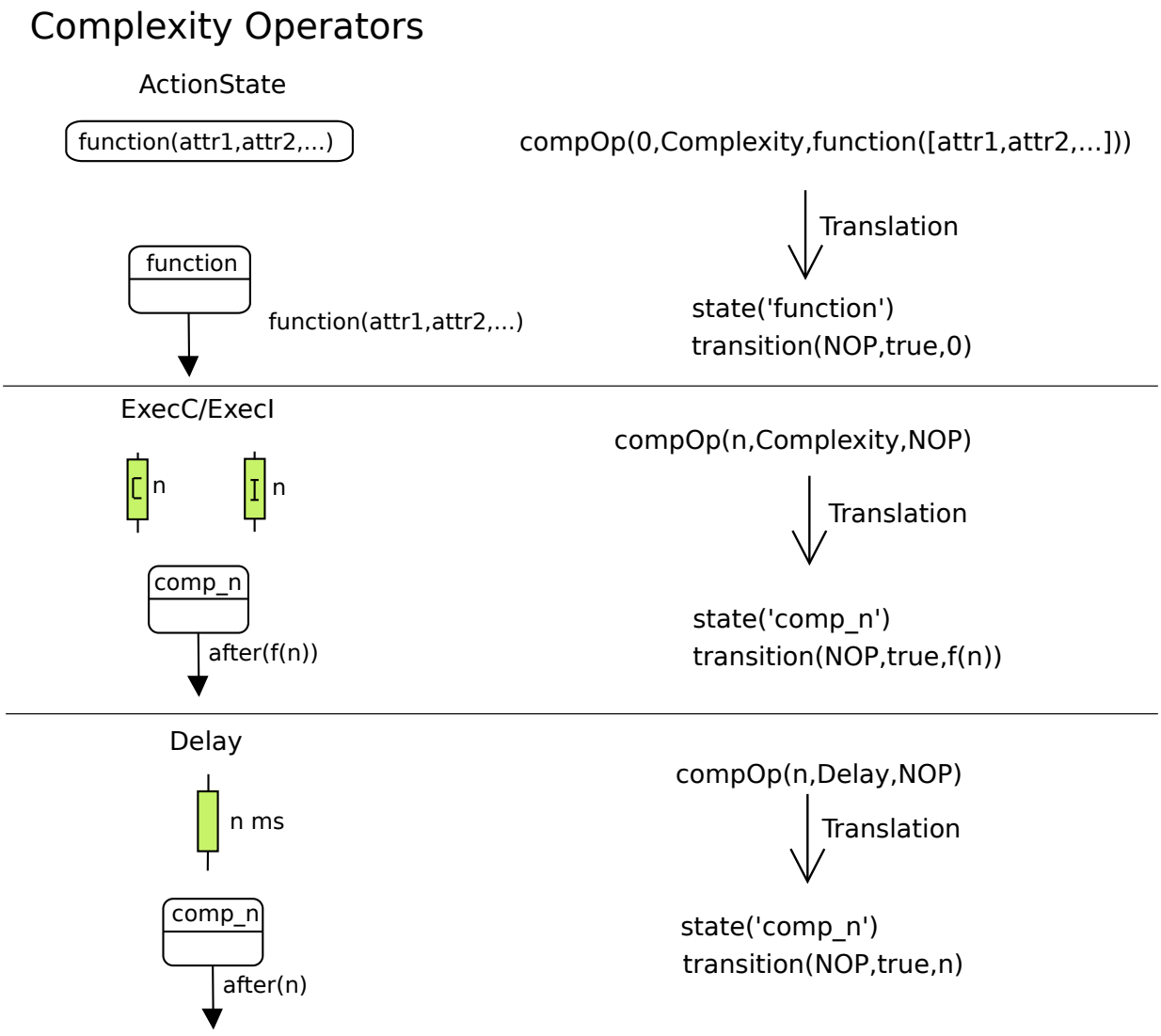


Figure 6-14: Translation of Functional Complexity Behavior Elements to Software Design Behavior Elements

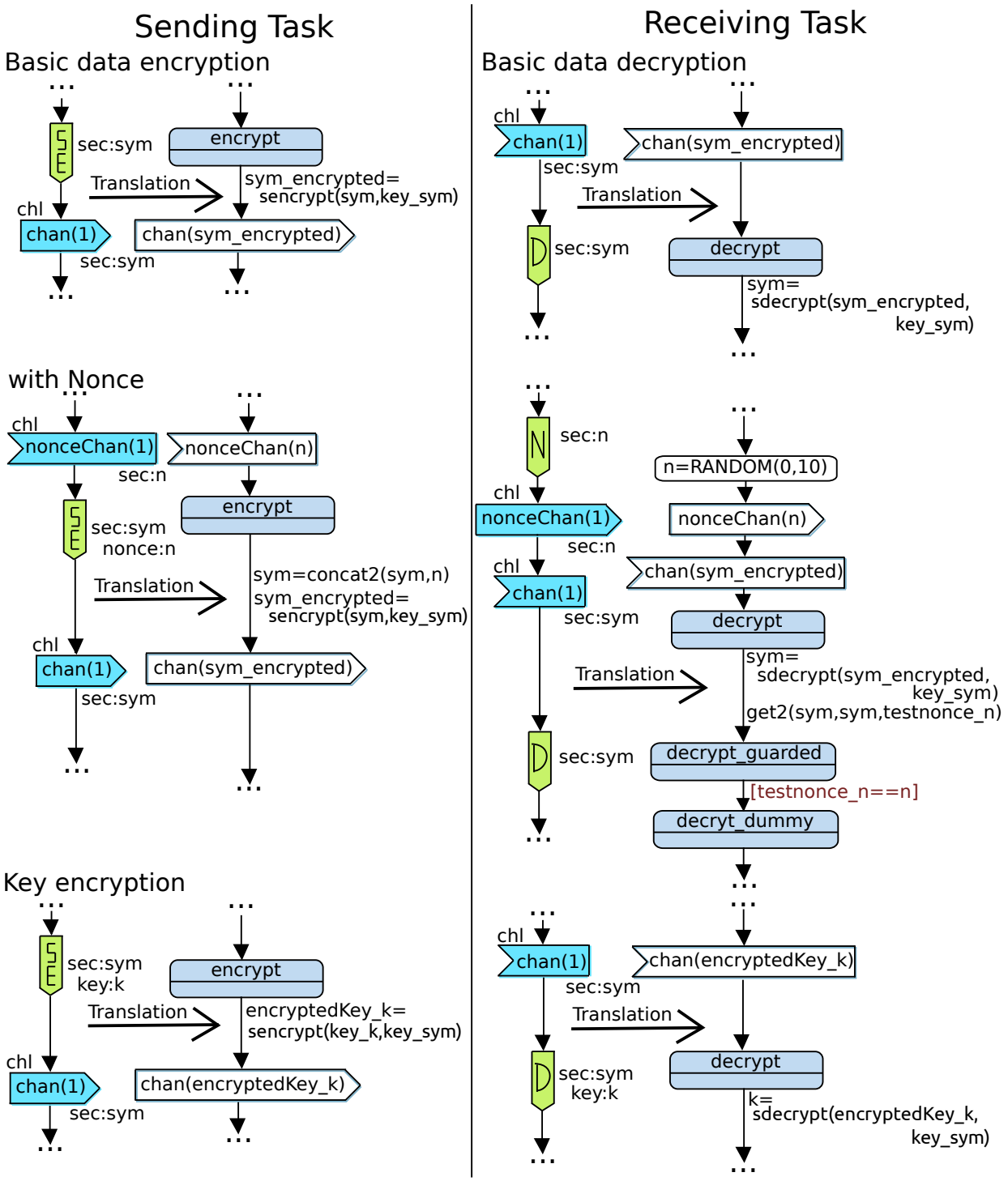


Figure 6-15: Translation of Symmetric Encryption Behavior Elements to Software Design Behavior Elements

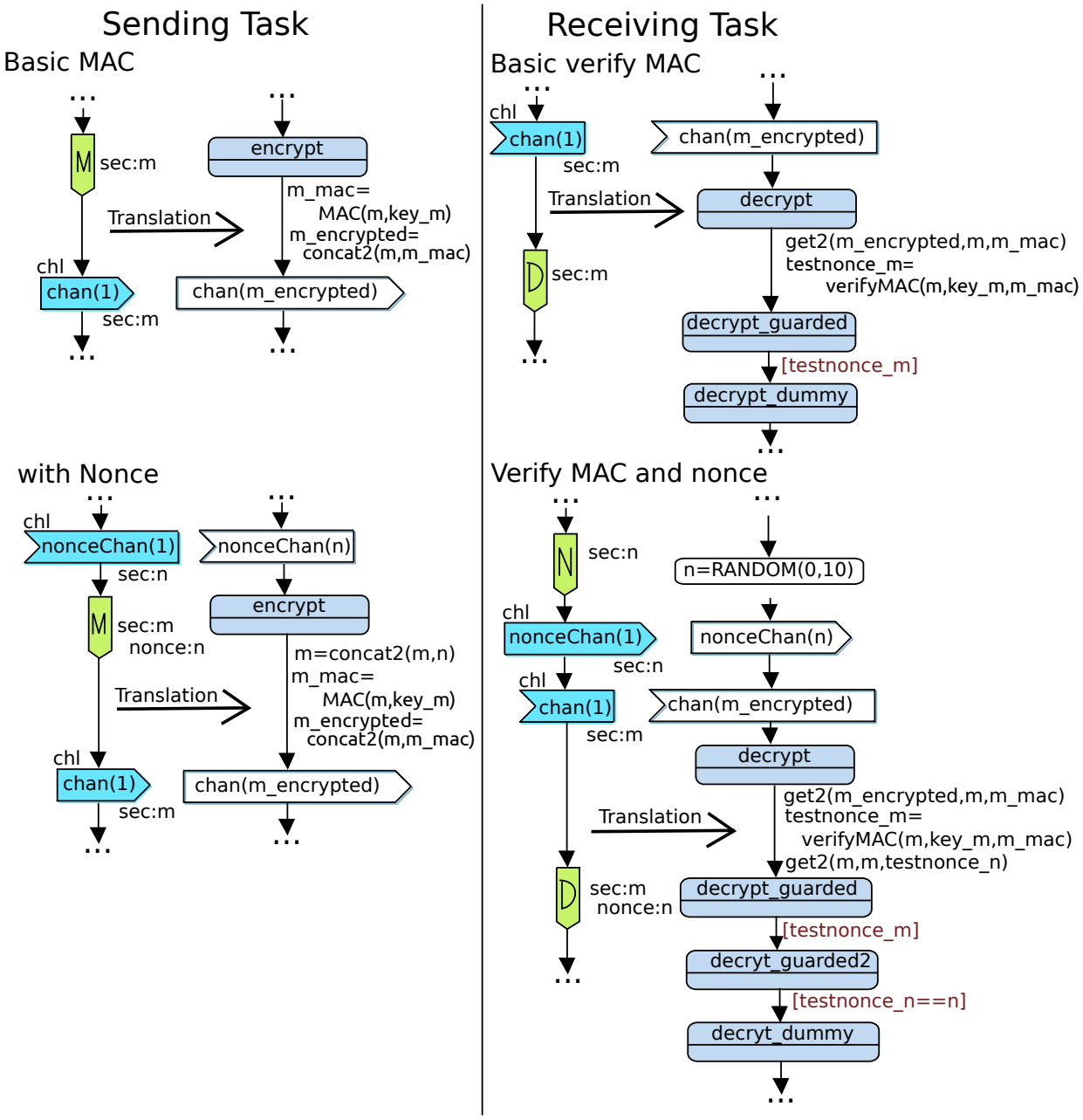


Figure 6-16: Translation of MAC Cryptographic Configuration to Software Design Behavior

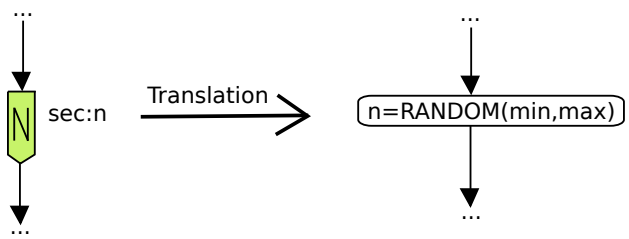


Figure 6-17: Translation of Nonce Cryptographic Configuration to Software Design Behavior

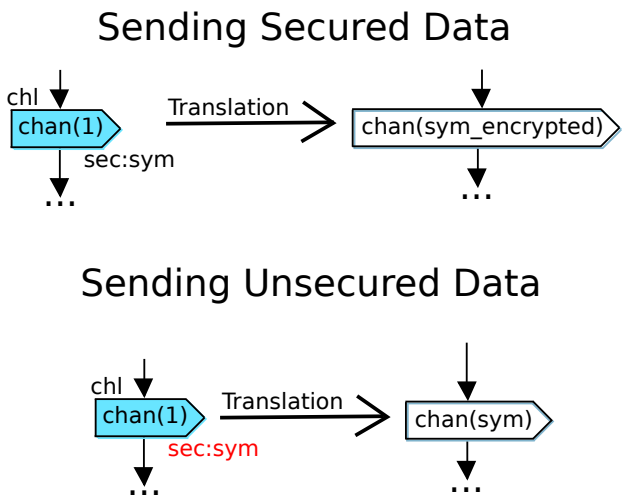


Figure 6-18: Translation of Sending Secured vs Unsecured Data to Software Design Behavior

If a nonce is added, then first the nonce n is exchanged between the two tasks. The sending task concatenates the nonce onto the attribute sym with $sym = concat2(sym, n)$. Next, the combined message is encrypted, and the encrypted message is sent, as described with basic data encryption.

To decrypt sym, then first the combined message is decrypted, and then the message and nonce are separated with the function $get2(sym, sym, testnonce\_n)$. If the message has not been tampered with or replayed, then $testnonce\_n$ should be the same as the original nonce n sent out. If they match, then decryption is finished and sym is accepted. Otherwise, sym is rejected as it has been tampered with.

Key encryption

To distribute keys may include encrypting the key and sending the encrypted message on a channel accessible by the attacker. The key being distributed key_k is encrypted with the symmetric encryption function, and then the encrypted key $encryptedKey_k$ is sent to the receiving task. Then, the receiving task decrypts the key with $k = sdecrypt(encryptedKey_k, key_sym)$. The attacker should not know key_sym, so he/she cannot recover the key key_k either.

Unsecured vs Secured channels

As explained in the previous chapter, there are instances where a task will send or receive the unsecured form of a Cryptographic Configuration, usually in instances where the data is sent between tasks before it is secured, and we need to track the data in its path. Sending unsecured data (across private buses/bridges) often occurs with the use of HSMs. Figure 6-18 shows an example of sending secured vs unsecured data from the Cryptographic Configuration sym, and the corresponding translations. When data is sent in the secured form, as shown in the previous examples, the attribute sent is named $sym_encrypted$, but when it is sent unsecured, it is named sym.

Asymmetric Encryption and MAC Cryptographic Configurations are similarly translated, while Nonces are translated to be a new attribute which takes a random value.

6.4.3.2 Formal Translation

The set of Operators in Partitioning are translated formally into Operators in Software Design, as described below:

1. $commOp(name, IN/OUT, n) \rightarrow commOp(name, IN/OUT, [name\_channeldata])$;
2. $commOp(name, IN/OUT, [attr1, attr2, ...] \rightarrow commOp(name, IN/OUT, [attr1, attr2, ...])$;
3. $compOp(n, Complexity, f(attr1, attr2, ...)) \rightarrow state('comp\_n')$,
   $transition(f(attr1, attr2, ...), true, f(n))$ or $subB$
4. $compOp(n, Delay, f(attr1, attr2, ...)) \rightarrow state('comp\_n', transition(f(attr1, attr2, ...), true, n)$
   or $subB)$
5. $choice([subB1, subB2, ...], [cond1, cond2, ..]) \rightarrow$
   $state('choicestate')$
   $next(state('choicestate')) = (trans1, trans2)$
   $trans1 = transition(NOP, cond1, 0)$;
   $R(subB1)$;
   $trans2 = transition(NOP, cond2, 0)$;
   $R(sub2)$;
6. $loop(subB1, initfunc, incfunc, exitcond); subB2; \rightarrow$
   $state('initloop')$;
   $transition(initfunc, true, 0)$;
   $state('loopstart')$;
   $next(state('loopstart')) = (trans\_loop, trans\_exit)\, trans\_loop = transition(NOP, !exitcond, 0)$;
   $R(subB1)$;
   $transition(incfunc, true, 0)$
   $trans\_exit = transition(NOP, exitcond, 0)$;
   $R(subB2)$;

To express the transformation in greater detail, we write out the pseudocode to transform each HW/SW Partitioning functional operator into AVATAR functional operators. The full translation of each operator is described in Algorithm 1 and 2. For readability, the algorithm is split across two different functions, one for simple communication and computation operators, and another for complex control operators.

```
function translateOperator
input: Mapping model operator op
output: translated Software design operator newOp, modified Software Design Models
if op ∈ simple operators (not choice, loop) then
    //exec operations
    if op ∈ exec then
        newOp= new Avatar State
        transition.after ← translate op complexity into time
    end
    if op ∈ delay then
        newOp= new Avatar State
        transition.after ← delay
    end
    if op in cryptoOperations then
        newOp = new Avatar State
        transition.action ← translate op function
    end
    //channel operations
    if op ∈ write channel then
        channel chan = operator.channel
        newOp= new Avatar Send Signal(chan.name, chanData)
    end
    if op ∈ read channel then
        channel chan = operator.channel
        newOp= new Avatar Receive Signal(chan.name, chanData)
    end
    //Request operations
    if op ∈ sendRequest then
        request = operator.request
        newOp= new Avatar Send Signal(request.name, request.parameters)
    end
    if op ∈ readRequest then
        request = operator.request
        newOp= new Avatar Receive Signal(request.name, request.parameters)
    end
    //Event operations
    if op ∈ sendEvent then
        event = operator.event
        newOp= new Avatar Send Signal(event.name, event.parameters)
    end
    if op ∈ readEvent then
        event = operator.event
        newOp= new Avatar Receive Signal(event.name, event.parameters)
    end
    Add newOp to State Machine Diagram nextOp = translateOperator(op.next))
    Add transition between newOperator and nextOp
    return newOp
end
```

Algorithm 1: Algorithm to translate simple DIPLODOCUS Functional Operators to AVATAR Software Design Operators

```
if op ∈ complex operators then
    if op ∈ choice then
        newOp = new AvatarState(choiceState)
        listOfNextElements ← new list of Avatar Operators
        for Operator next in op.nexts do
            listOfNextElements.add((translateOperator(next))
        end
        for AvatarOperator nextOp in listOfNextElements do
            Add transition between newOp and nextOp
            transition.guard ← op.guard[next]
        end
    end
    if op ∈ sequence then
        for seqOp ∈ op.nexts do
            newOp = translateOperator(seqOp)
        end
        Add transitions between each pair of sequential seqOp
    end
    if op ∈ randomSequence then
        newOp = new AvatarState(choiceState)
        for seqOp ∈ op.nexts do
            newOp2 = translateOperator(seqOp)
            Add transition between newOp and newOp2
            //Create the possible choices on the remaining operators in the randomSequence tmpOp = op
            tmpOp.nexts.remove(seqOp)
            newOp3 = translateOperator(tmpOp)
            Add transition between newOp2 and newOp3
        end
    end
    if op ∈ loop then
        newOp = new AvatarState(Init loop)
        transition.action ← op.init
        newOp2 = new AvatarState(loop start)
        transition2.guard ← loopcondition
        transition2.action ← op.loopfunction
        Add transition 2 between newOp and newOp2
        loopBehavior = translateOperator(op.loopBehavior)
        Add transition between last element in loopBehavior and newOp2
        exitBehavior = translateOperator(op.next)
        Add transition between newOp2 and exitBehavior
    end
end
Add all new operators and transitions
```

Algorithm 2: Algorithm to translate complex DIPLODOCUS Functional Operators to AVATAR Software Design Operators

6.5 Translation to ProVerif

This section describes the 2-step translation from DIPLODOCUS mapping models to ProVerif specifications. Figure 6-19 shows a translation from DIPLODOCUS to AVATAR, and then to ProVerif, for a representative sample of the different modeling elements.

At the highest level, we need to translate the mapping security queries to Proverif queries, then translate each task into one or multiple processes. The basic functions (symmetric encryption, asymmetric encryption, MAC, and etc) are declared in the Functions section at the beginning of each ProVerif specification, and do not need to be translated.

6.5.1 Translation of Queries

Security annotations in DIPLODOCUS indicate which security properties should be verified. These annotations should be placed based on the security requirements defined in the Analysis Phase. Figure 6-20 shows the different annotations, as shown in the indicated circle. Confidentiality and Authenticity annotations are marked on communications in the Functional Component Design diagram, while Reachability annotations are marked on operators in Activity diagrams.

A small grey lock on the sending channel indicates that the Confidentiality for the data sent along the channel should be checked. A split grey lock on the receiving channel indicates that Authenticity of the data sent along that channel should be checked. We place these annotations on the channels instead of channel operators since there may be multiple read/write channel operators for each channel, and it would be tedious to tag each operator.

The reachability of states is also a safety property and can also be verified by TTool's safety checker, and states which should be checked for reachability are instead marked with an annotation 'RL', which stands for 'Reachability' and 'Liveness'.

Each security query marked on a channel is translated into a pragma in AVATAR, and then to the corresponding query in ProVerif, depending on which property is being checked. For example, the three annotations in our example would be translated to:

```
(* Queries Secret *)
query attacker (new chan_chData)

(* Queries Event *)
query event(enteringState___lidar__writelidardata()).
event enteringState___lidar__writelidardata().

(* Authenticity *)
event authenticity___T1___chan_chData___sendingMsg(bitstring).
event authenticity___T2___chan_chData___msgVerified(bitstring).
query dummyM: bitstring;
inj-event(authenticity___T2___chan_chData___msgVerified (dummyM)) ==>
inj-event(authenticity___T1___chan_chData___sendingMsg (dummyM)).
```

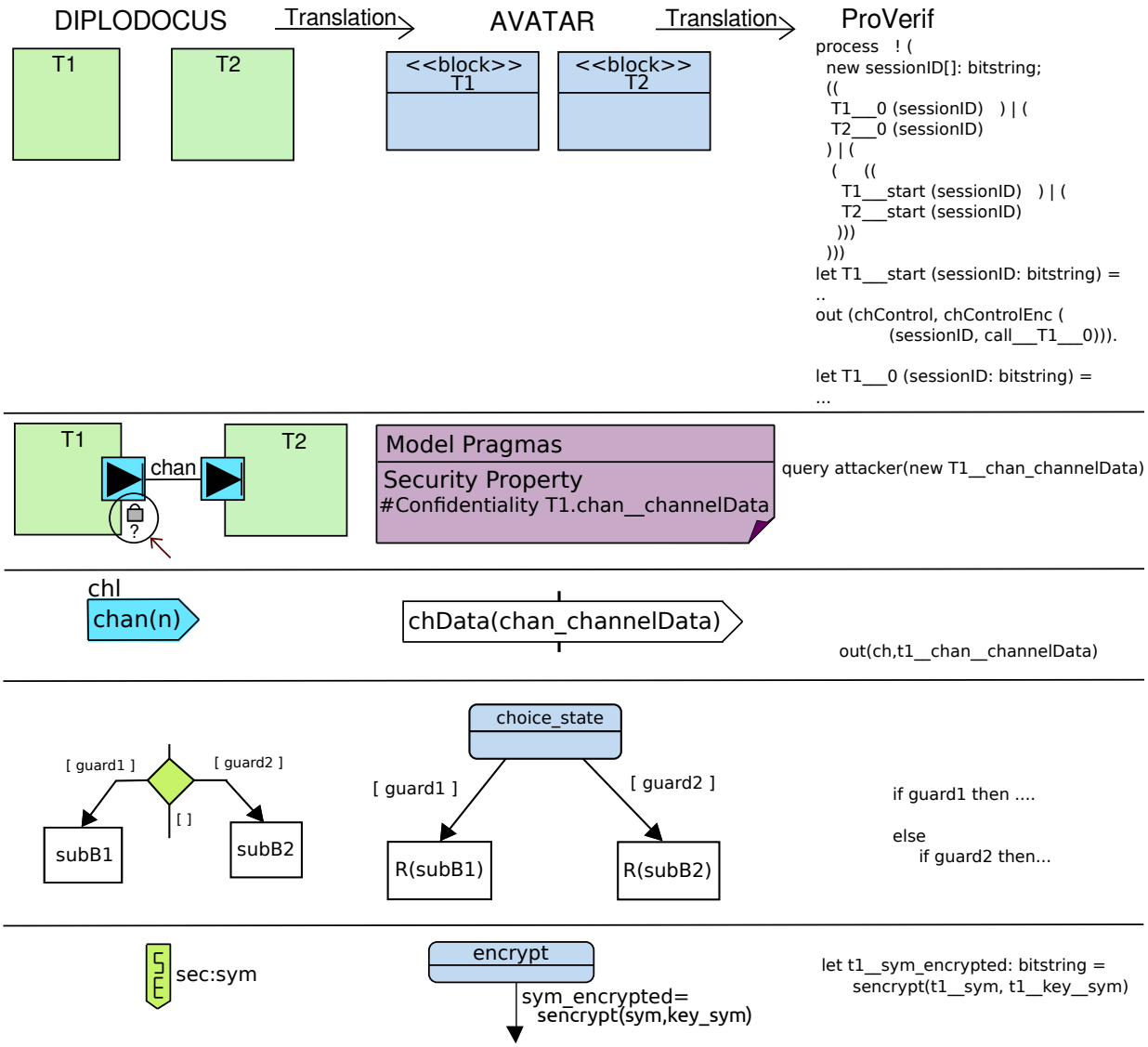


Figure 6-19: Translation of DIPLODOCUS to AVATAR to ProVerif

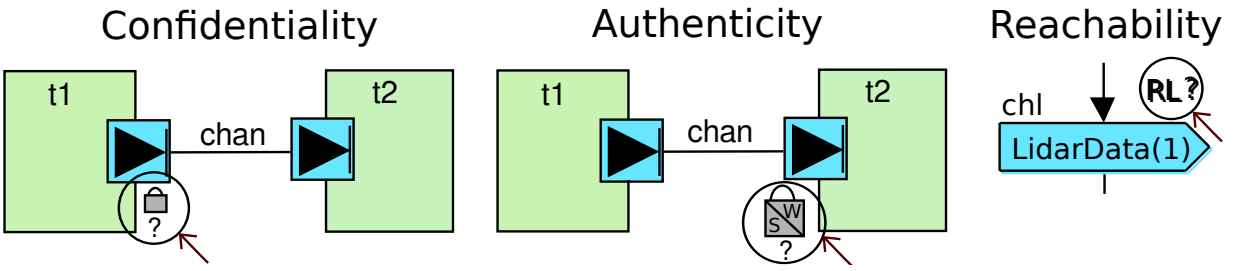


Figure 6-20: Confidentiality, Authenticity, and Reachability Security Annotations

These 3 queries in ProVerif check the confidentiality of attribute *chan_chData*, then the reachability of event *enteringState___lidar__writelidardata())* (corresponding to the LidarData write channel operator), and then the authenticity of attribute *chan_chData*: that if data in state *msgVerified* in task T1 exists, it must be the same as the data *chan_chData* in state *sendingMsg*.

6.5.2 Translation of Tasks

Each task in the model is translated into one of the processes started by the main process, where each Task t in D adds the following lines to P:

```
let t_start(x,...)

let t_0(sessionID: bitstring) =
```

and a line is added to the main process:

```
process
! (
new sessionID[]: bitstring;
((
t___0 (sessionID)
...
```

6.5.3 Translation of Actions

The Activity of each task is translated into a State Machine using the translation described in the previous section, and then the state machine is translated into basic blocks as described in [208], so that a single task in an activity may be split across multiple processes.

As described above, each operator in DIPLODOCUS is translated into one or more AVATAR operators. The AVATAR operators are then translated into actions in a ProVerif specification. As ProVerif has no concept of time, temporal operators are removed, but communications, actions, and control operations are translated. The exact translation of operators involves first applying Algorithm 1 and then the algorithm described in [208].

Thus, in this section, we have formally described how HW/SW Partitioning (DIPLODOCUS) models are transformed in 2 steps into ProVerif text specifications, by describing the individual transformations for each element: queries, tasks, and task behaviors.

6.6 Proof of Correctness

With all of the terms and translations formally defined, we can begin our proof. In this section, we prove that Mapping models are correctly translated into a ProVerif specification. In this case, 'correctly trans-

lated' means that if a security property is violated in the model, then ProVerif should return that the result is verified false. We use proof by induction as we can iterate over the length of the trace and add a new operation in each inductive step.

As [208] proved the correctness of the Avatar-to-ProVerif translation in terms of Confidentiality, we must prove that our DIPLODOCUS-to-AVATAR translation also preserves Confidentiality properties in order to prove that the full DIPLODOCUS-to-ProVerif translation preserves Confidentiality properties.

For our proof, we assume there is a variable X which we wish to assure is not recoverable by the attacker.

Statement: For each DIPLODOCUS trace D of length n, there exists an AVATAR trace S generated by the automatic translation process, and if X is not confidential in D, then it will not be confidential in S. We prove this statement by induction iterating on the length n.

We denote AK as the current set of knowledge of the attacker, including all possible knowledge that can be obtained from AK. For example, if $enc_X = symmetricEncrypt(X, key)$ and $AK = enc_X, key$, then $AK = enc_X, key, X$. At the start of the process, $AK = \emptyset$. The Confidentiality of X is verified *false* if $X \in AK$, and the Confidentiality of X is verified *true* if $x \notin AK$ after expanding AK to contain all discoverable knowledge.

6.6.1 Base case

The Base case is a trace of length 0, or processes with no actions. If no operations occur in D, then no operations occur in S.

No variables, including X, have been transmitted on a public channel, so the attacker has no knowledge in both D and S: $AK = \emptyset$, and therefore X remains confidential in both D and S.

6.6.2 Inductive Step

If a trace of length n will be correctly translated, then we prove that a trace of length of length n+1 will also be correctly translated.

Adding the extra element to generate a trace of length n+1 can involve adding the following actions:

Choice

If the n+1th action is a choice, then the AVATAR trace will also contain a choice. Each branch will maintain the current attacker knowledge, so adding a choice alone maintains the correctness of the translation in terms of confidentiality of X.

Channel Out

There exist multiple possibilities of the possible channel out operations relevant to the Confidentiality of X.

Writing X to a channel in DIPLODOCUS is translated into a Send Signal operation, in the form:
$commOp(ch, OUT, X$
$\rightarrow commOp(ch, OUT, X)$; In both DIPLODOCUS and AVATAR, X is being sent along a public channel.

Sending data along a public channel will increase the knowledge of the attacker. In DIPLODOCUS, if X is sent along a public channel, then $AK_{n+1} = AK_n \cup X$. The operation $commOp(ch, OUT, X)$ in AVATAR will similarly increase the knowledge of the attacker by X.

Sending any data unrelated to X will not affect the knowledge of the attacker AK.

Encryption If instead of sending X, $encX$ is written to a channel, where $encX = sencrypt(X, key_X)$, both the DIPLODOCUS and AVATAR trace will be adding an element $commOP(ch, OUT, encX)$.

In this case, $AK_{n+1} = AK_n \cup encX$. If $key_X \in AK$, then $AK = key_X, encX$, which leads to $AK = key_X, encX, X$. Otherwise, if $AK = encX$, then no additional knowledge about X can be gained at this point.

If instead, the key key_X is sent along the public channel, then both traces will add the element $commOP(ch, OUT, key_X)$.

Attacker knowledge will increase as: $AK_{n+1} = AK_n \cup key_X$. If $encX \in AK$, then $AK = key_X, encX$, which leads to $AK = key_X, encX, X$. Otherwise, if $AK = key$, then no additional knowledge about X can be gained at this point.

The same applies for asymmetric encryption: as long as the attacker does not have the key, then he/she cannot recover X.

Hash/MAC

Hashes are assumed to be 1-way functions, where the original message cannot be recovered from the hash. If $hash(X)$ is sent along a public channel, then $AK = hash(X)$, but $x \notin AK$.

Message Authentication Codes are similarly calculated with a hash function, so that even obtaining the MAC and key will not be enough for an attacker to recover the original message.

In each case, the change in the set of attacker knowledge is identical in both AVATAR and DIPLODOCUS.

Channel In

Reading data in DIPLODOCUS is translated into a Receive Signal operation, in the form: $commOp(ch, IN, X)$ $\rightarrow commOp(ch, IN, X)$

Receiving data that has already been sent along a public channel does not change the knowledge of the attacker, for if X has already been sent, then $X \in AK$, and if X has not been sent, then $X \notin AK$, but the operation $commOP(ch, IN, X)$ cannot disclose X either for the receiving task does not know the value of X.

We assume an empty memory at the start of system execution, so it is impossible to read X if the write X operation has not yet been executed.

An attacker can inject any data he/she wishes, but that impacts the authenticity of the communications, but not confidentiality of data that we examine in this case.

Functions

Functions calculating new attribute values in DIPLODOCUS are translated into actions in AVATAR, using the translation $compOp(n, Complexity, f(attr1, attr2, ...))$

$\rightarrow state('comp\_'n), transition(f(attr1, attr2, ...), true, f(n), 0)$

Calculations within a process involve no output of data along a public channel in both DIPLODOCUS and AVATAR. Therefore, AK remains unchanged in both traces, and the confidentiality of X is unchanged in both traces.

6.6.3 Conclusion

Since the base case and inductive step both hold, our statement above is proven: for each DIPLODOCUS trace D and the corresponding generated AVATAR trace S, the confidentiality of any variable X will be the same in both traces. Therefore, we have proved that the DIPLODOCUS-to-Avatar translation preserves the property of Confidentiality.

As we will discuss in the Perspectives in Section 8.3, a proof that our model transformation preserves the property of Authenticity should be developed in the future.

6.7 ProVerif Results

The security results returned by the ProVerif prover indicate if each query is verified true, verified false, or cannot be verified, as shown on the window in Figure 6-21. These results are back-traced onto the DIPLODOCUS Component Diagrams for user convenience, and remain on the diagrams for the user to refer to as they reconsider and modify the models. This capability was first added as described [209]. Before the start of this thesis, a user had to read textual ProVerif output, and then remember it as he/she modified the models. For larger models, it may be difficult to remember the exact security verification results for each query.

The security annotations, in the form of grey locks, as described in Section 6.5.1, are colored after security verification to indicate if their corresponding security property is satisfied (green) or not satisfied (red). If there is an error with the prover and the results cannot be determined, then the lock remains grey.

For confidentiality results, data exchanged across a channel vulnerable to an attacker is indicated by a crossed-out red lock. However, data proved confidential is indicated by a green lock. Weak Authenticity and Strong Authenticity are indicated by the split lock with the half-locks colored green for verified true and half-locks colored red and crossed out for verified false. The two halves of the lock corresponding to authenticity can be colored differently, for example, if weak authenticity is verified true but strong authenticity is verified false.

Since a single functional model may have multiple associated mappings, the name of the mapping is displayed next to the security annotations, in order to indicate which mapping these results correspond to.

For example, we previously mentioned that an attacker tampering with internal communications could prevent the braking system from correctly functioning. We demonstrate how ProVerif verification works by checking the security of our Autonomous Vehicle system, focusing on ensuring the authenticity of the communications between the Navigation and Perception units. In the current architecture, shown in Figure 6-22, the Perception and Navigation units communicate across an insecure bus. As expected, Perception data is not verified authentic.

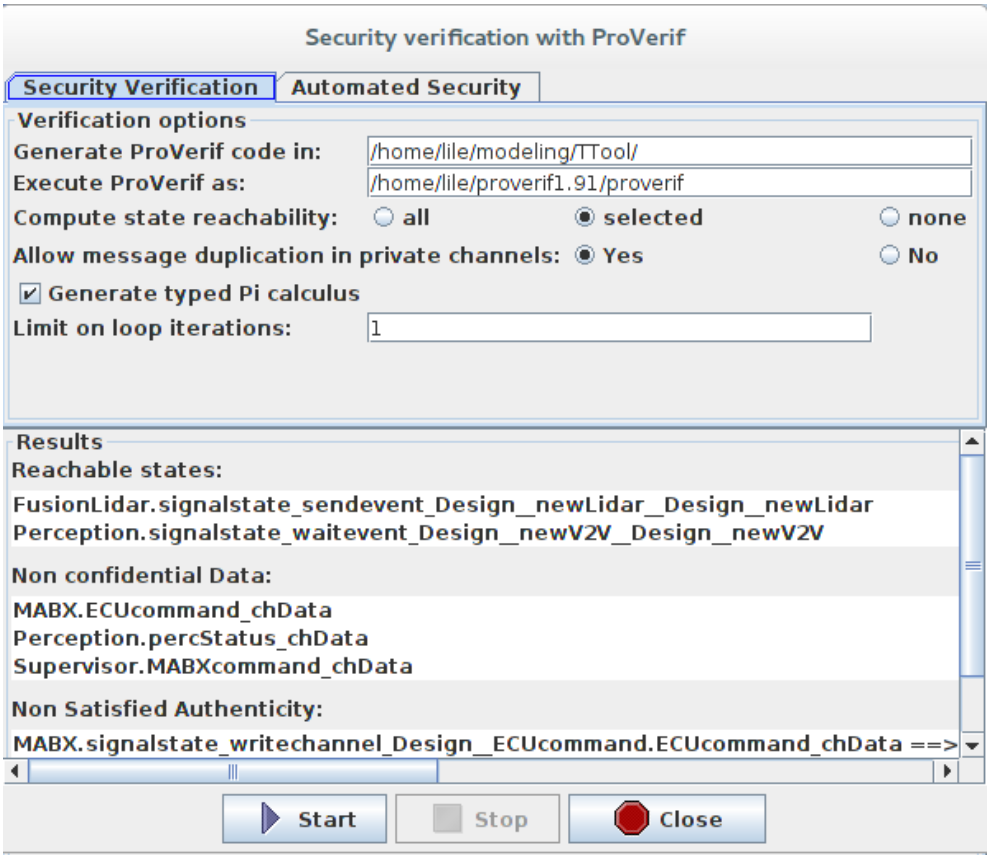


Figure 6-21: ProVerif Verification Results Output

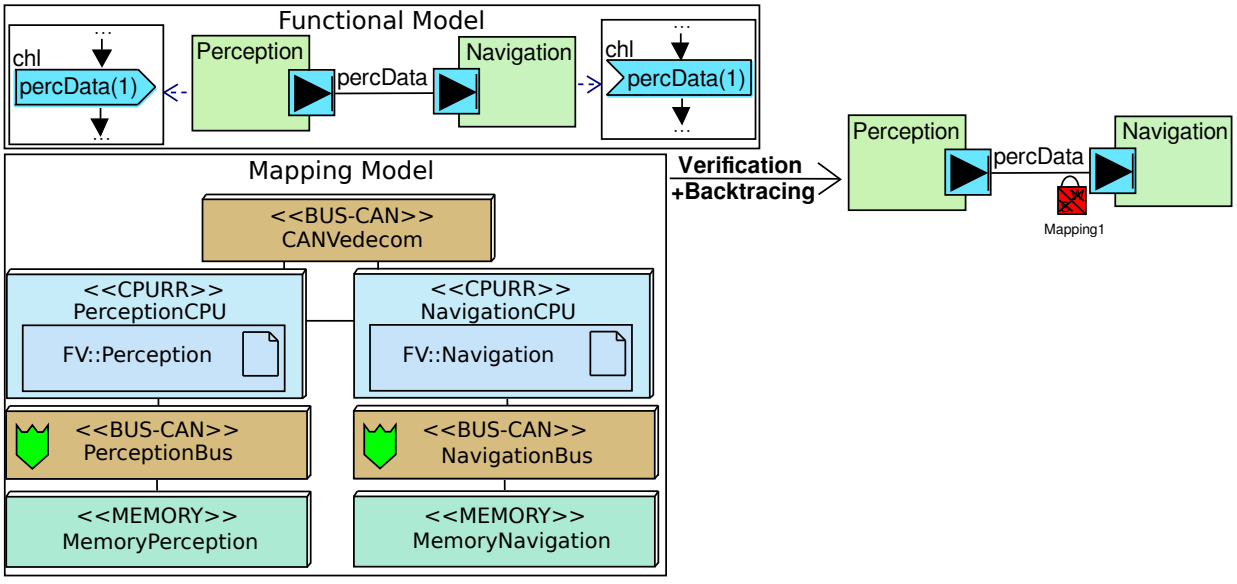


Figure 6-22: Verification Results for Default Mapping

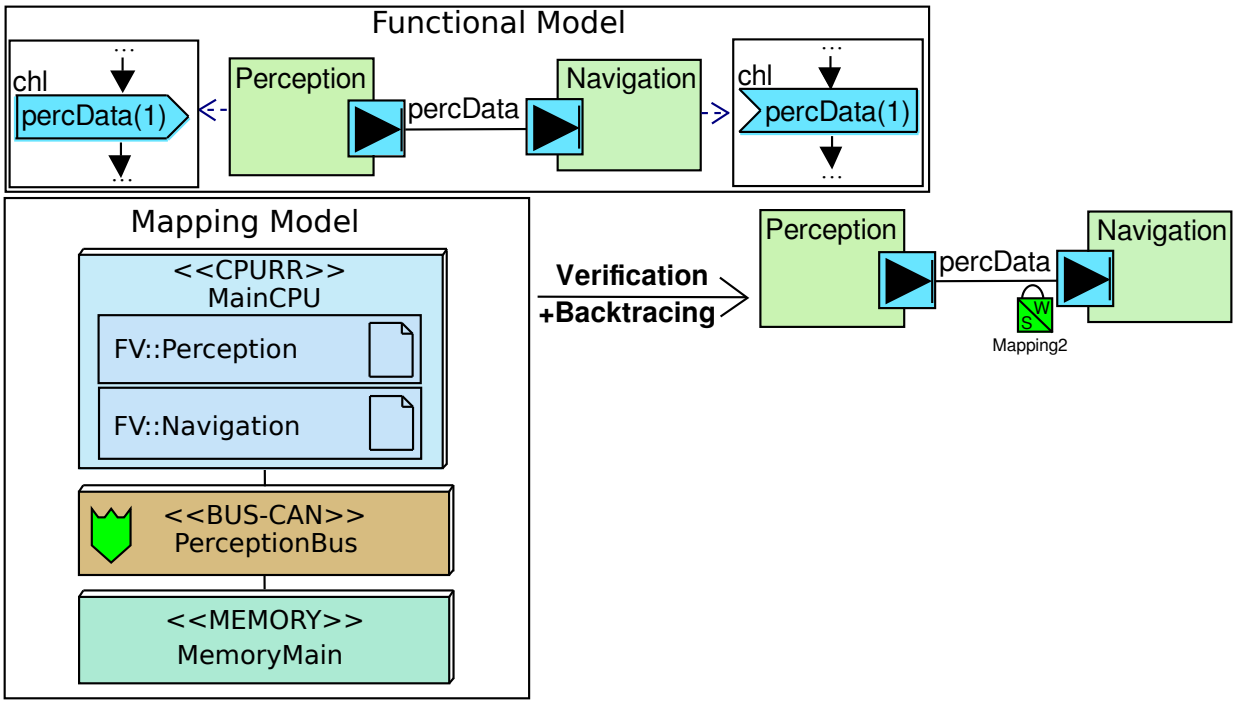


Figure 6-23: Verification Results for Modified Mapping with Perception and Navigation Tasks mapped to same CPU

However, if the architecture is modified as shown in 6-23, where Perception and Navigation are mapped to the same CPU, and there exists a secure path to memory, then Perception data is verified authentic. In the next section, we discuss how our tookit can automatically add Cryptographic Configurations to represent security protocols to secure data regardless of the architecture.

6.8 Automatic Generation

As previously noted, software engineers are often not experts in security. In the HW/SW Partitioning phase, we therefore offer automatic generation capabilities, including adding Cryptographic Configurations with associated security operators and Hardware Security Modules. Figure 6-24 shows the automatic generation window, with its many possible generation modes and options. At each generation, the toolkit can add security operators (and optionally using a HSM to perform security operations for selected tasks), or map all keys to a memory that is accessible securely.

The security representations are added based on the security properties that must be fulfilled for each channel. The automatically generated secured model is only to provide an estimate of the performance properties of the secured system, and the exact security protocols should be described in greater detail in the Software Design phase. In certain cases, there are multiple methods to satisfy a security property, and we take only one of these options to simplify the algorithm. For example, by default, all encryption operators are set to symmetric encryption, for simplicity in mapping keys.

6.8.1 Security Requirements

The security requirements should be first indicated with the security annotations (grey locks) on the Component Diagram. It is assumed that if the user marks that there should be a query (Confidentiality and/or Authenticity) of data sent along a certain channel, then he/she desires that channel data fulfill those security properties. Based on the properties indicated, different types of cryptographic operators are added to the sending and receiving task. Next, all data operations for that channel are tagged to show that it is secured data from that Cryptographic Configuration being sent.

6.8.2 Addition of Security Operators

In the first option, the toolkit can add security operators based on the selected security property to ensure and the based on the security requirements of each channel. The user selects if he/she wishes to ensure confidentiality and/or weak authenticity or strong authenticity for all channels marked with security annotations. For example, if the user only wishes to add operators to ensure confidentiality, then the toolkit will ignore the requirements on authenticity and only add the encryption operators to channels marked with the security annotation indicating that the data on them must be confidential. However, if the user also indicates that he/she wishes to ensure weak (and optionally strong) authenticity, then the toolkit will add MACs and/or nonces to channels which have been marked that their communications should be authentic. The detailed operators which need to be added are described further in this section. The mapping of keys for each operator added can then be performed automatically with the key mapping algorithm to be described later.

In addition, for the operators being added, estimated times to perform encryption, decryption, calculate a MAC, etc, and the overhead, can be manually set in lieu of using the default options.

Algorithm 3 and 4 show how security operators are automatically added to a mapping model based on ProVerif verification results. Part 1 parses the verification results to determine which channels need to be secured, and part 2 takes the list of channels and currently-violated security properties, and modifies the activity diagrams by adding the appropriate security operators to the activity diagrams of each task associated with the channel.

6.8.2.1 Confidentiality only

If the only property that needs to be fulfilled is confidentiality, then the data being sent along the channel must be encrypted to be unreadable by the attacker, and it must also be decryptable by the receiver. Furthermore, the associated keys should have been securely shared and stored, though we do handle this with . Figure 6-25 shows the operators added: the message should be encrypted before being sent, and then decrypted after being received.

6.8.2.2 Authenticity only

Ensuring Authenticity prevents the attacker from injecting custom data, which can be carried out by adding a sort of additional information to the message. Similar to how a checksum can check if the data within

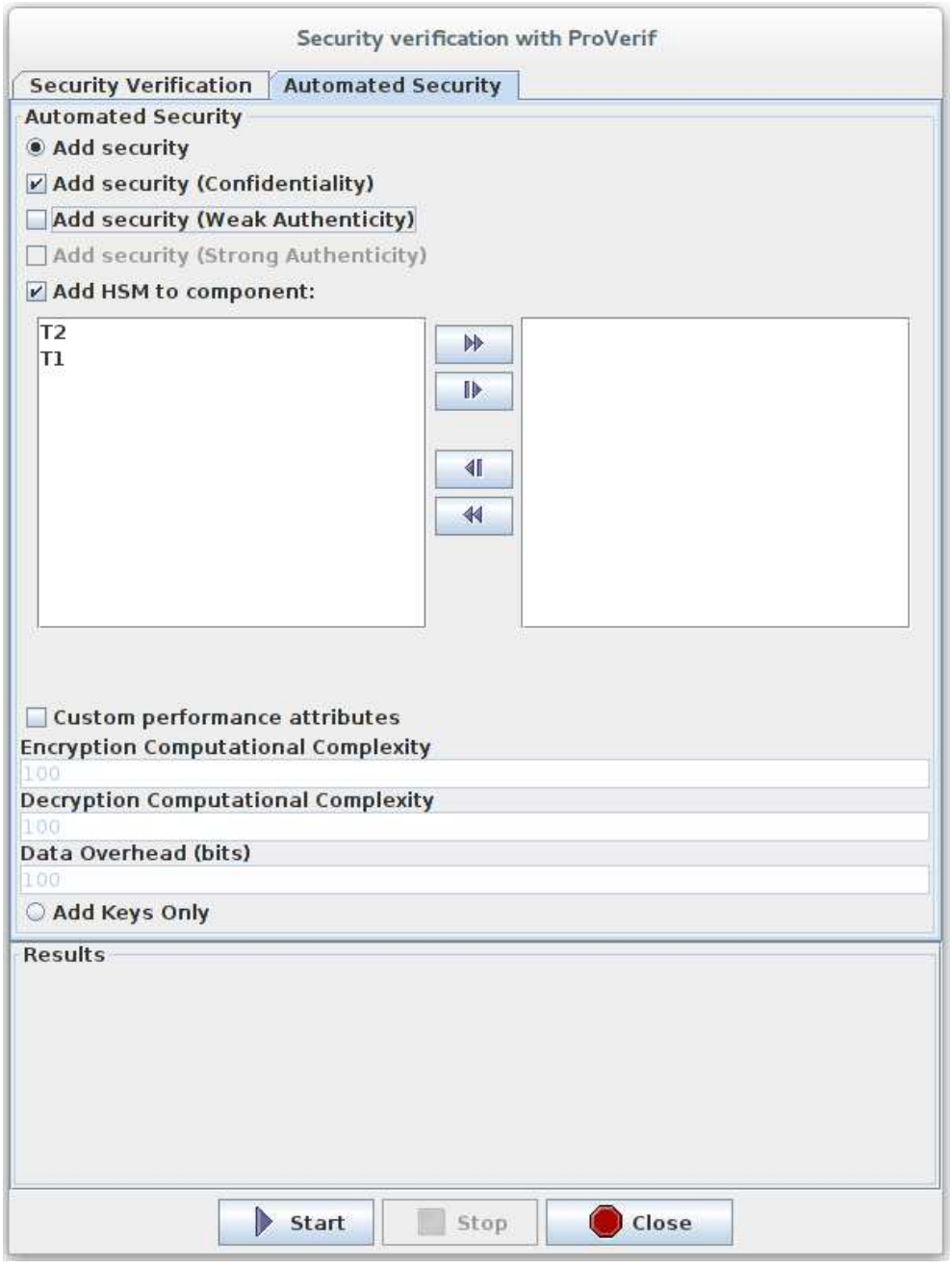


Figure 6-24: Window for Automatic generation of security

```
input: ProVerif verification results, insecure mapping model
options: addWeakAuthenticity? addStrongAuthenticity?
//Confidentiality is added by default
//Strong authenticity can be added only if weak authenticity is also added
output: new generated secured mapping model
struct chanInfo : [channel, securityProperties(strong authenticity, weak authenticity, )]
toSecure = new list of chanInfo
for channel in model do
    secProperties ← new list of security properties
    if channel confidentiality = 'Verified False' and channel.checkConfidentiality then
        secProperties.add(Confidentiality)
    end
    if addWeakAuthenticity and channel.checkAuthenticity then
        if channel authenticity = 'Verified False' then
            secProperties.add(Weak Authenticity)
            if addStrongAuthenticity then
                secProperties.add(Strong Authenticity)
            end
        end
    end
    chanInfo.add(new chanInfo(channel, secProperties))
end
```

Algorithm 3: Algorithm to Modify Diagrams to automatically add security operators Part 1

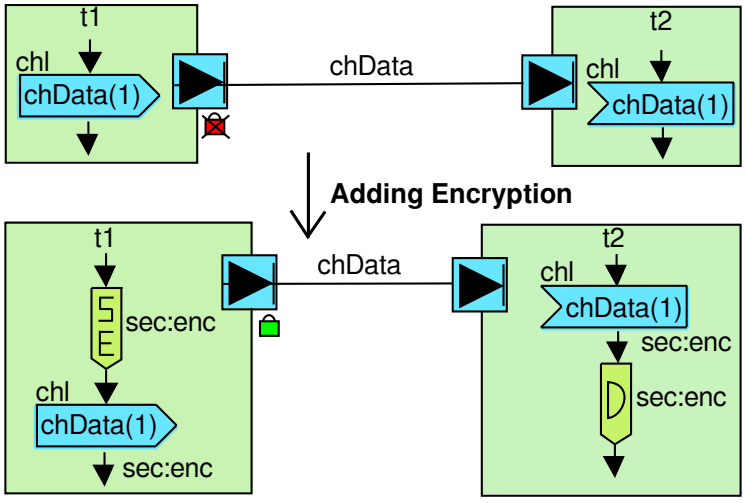


Figure 6-25: Automatic generation of security operators to ensure confidentiality

```
for channel in chanInfo do
    if Confidentiality, Authenticity ∈ channel.securityProperties then
        if StrongAuthenticity ∈ channel.securityProperties then
            Add new nonce channel between channel.destinationTask → channel.originTask
            Add send nonce to channel.destinationTask
            Add receive nonce to channel.originTask
        end
        foreach Write Channel Operator do
            Add Encryption Operator before write operator
            if addStrongAuthenticity then
                Add nonce to Encryption Operator
            end
        end
        foreach Read Channel Operator do
            Add Decryption Operator after read operator
        end
    end
    else if Authenticity ∈ channel.securityProperties then
        if StrongAuthenticity ∈ channel.securityProperties then
            Add new nonce channel between channel.destinationTask → channel.originTask
            Add send nonce to channel.destinationTask
            Add receive nonce to channel.originTask
        end
        foreach Write Channel Operator do
            Add MAC Operator before write operator
            if addStrongAuthenticity then
                Add nonce to Encryption Operator
            end
        end
        foreach Read Channel Operator do
            Add MAC Verification Operator after read operator
        end
    end
    else if Confidentiality ∈ channel.securityProperties then
        foreach Write Channel Operator do
            Add Encryption Operator before write operator
        end
        foreach Read Channel Operator do
            Add Decryption Operator after read operator
        end
    end
end
```

Algorithm 4: Algorithm to Modify Diagrams to automatically add security operators Part 2

has been modified due to lossy transmissions, Message Authentication Codes can be calculated by only authorized individuals with a key and can be used to detect if a message has been modified.

Message Authentication Codes are generated from a message and cryptographic key. The MAC can be concatenated onto a message before being sent. The receiver then splits the combined message into the original message and the MAC, calculates the MAC based on the shared secret key, and checks if the calculated and original MAC match. If the attacker tampers with a message secured only by encryption, it is assumed that the message will not decrypt correctly, but it is also possible that the decrypted message can be accepted by the receiver. Therefore, it is recommended to use Message Authentication Codes to ensure authenticity of messages [186].

As shown in Figure 6-26, the MAC operator calculates the MAC and concatenates it to the message, and the corresponding 'Decrypt' operator for a MAC serves to verify the MAC, and reject the received data if the MAC does not match.

If however Strong Authenticity must be guaranteed, or that the system should resist replay attacks, then a MAC alone is insufficient, since a replayed message will match its MAC as it was sent by an authorized sender. Instead, a nonce, sequence number or timestamp should be provided for each message. As shown in the bottom of Figure 6-26, the receiver first sends the sender an unique nonce before each message transmission. The sender concatenates the nonce onto each message, and then calculates the MAC of the combined message. The receiver then again checks that the MAC matches the message, and also checks if the nonce matches the one expressly sent for this message. If the attacker replays a message, then the nonce will not match and the message can be rejected.

6.8.2.3 Confidentiality and Authenticity

Confidentiality and Authenticity can be ensured together by adding a nonce and then encrypting the combined message, as shown in Figure 6-27. Adding a MAC alone would not ensure confidentiality of the message, and encrypting the message alone would not ensure authenticity. While Encrypt-then-MAC could also ensure Authenticity, we chose to exchange a nonce and then encrypt the message as it was simpler.

6.8.3 HSM Generation

Instead of adding the security operators within each task, it is possible to indicate that all security operations should be performed with a HSM instead. After the user selects which tasks should have a HSM added, our tookit modifies the modeling diagrams to add the HSM, including modifying the task's activity diagrams to send data to the HSM, generating the HSM's activity diagram, etc. A single Hardware Security Module is added to each processor which executes at least one of the designated tasks. If multiple tasks mapped to a single CPU are designated to have a HSM added to them, then only a single HSM will be added.

For each HSM to be added to perform security operations for one or more tasks, first, the architectural diagram is modified to add a Hardware Accelerator and memory, with a connecting private bus, as shown in the previous chapter in Figure 5-10.

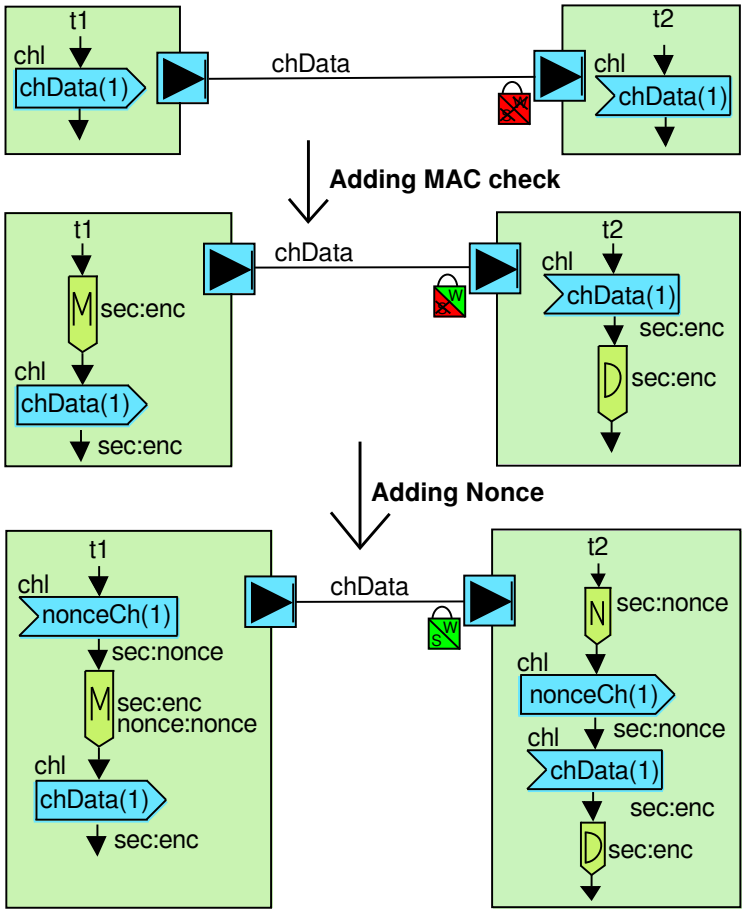


Figure 6-26: Automatic generation of security operators to ensure Weak and Strong Authenticity

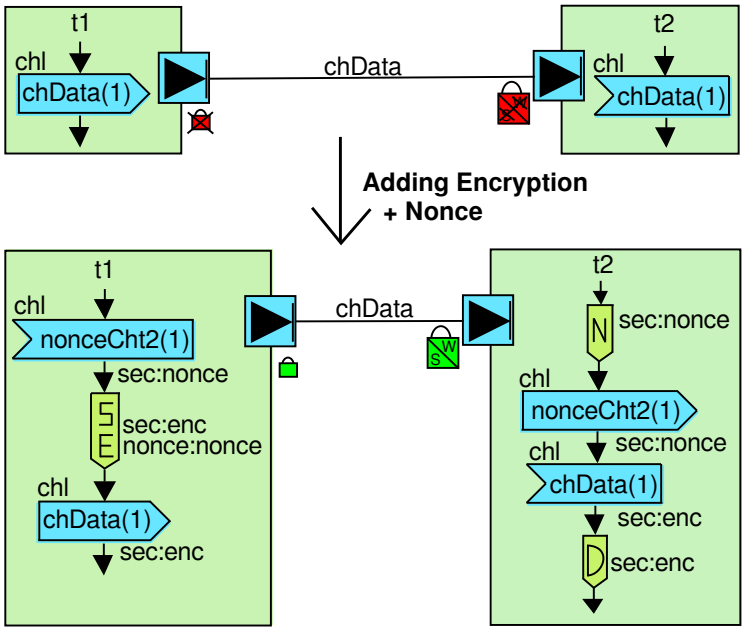


Figure 6-27: Automatic generation of security operators to ensure Confidentiality and Authenticity

Next, we determine which data needs to be secured by the HSM. For each channel, we use a modified version of Algorithm 3 to determine which security properties need to be satisfied. The activity diagrams of the sending task T1 and receiving task T2 are then modified.

The component diagram is modified to add a new task HSM. For each data communication which needs to be secured, a 'data_channel' and 'retData_channel' is added between the HSM task and its associated original task. Before each secure exchange of data between T1 and T2, T1 first sends the data to secure along 'data_channel'. The associated security operators from 4 are then added to the HSM. Channel operators are then added to the HSM and T1 so the HSM can return the data to T2 along 'retData_channel'. Finally, a decrypt operator is added to T2 after the reception of the data.

6.8.4 Automatic Key Mapping

With multiple Cryptographic Configurations, it may become tedious to the map all of the keys to memory. Our toolkit therefore can find every Cryptographic Configuration used by a task, and then, depending on the type of the Cryptographic Configuration, map each applicable key to a memory that the task can securely access. For Cryptographic Configurations of type symmetric encryption or MAC, both the sending and receiving task will need to be able to access the key. For asymmetric encryption, however, all the sending tasks will need the public key while only the receiving task will need to access the private key.

If a key is sent along a bus accessible to an attacker, then the key would be known to the attacker, so we wish to avoid sending keys along public buses. Our key mapping algorithm, shown in Algorithm 5, iterates over each Cryptographic Configuration used, and checks each task if it uses that Cryptographic Configuration. For each task which needs the key, the algorithm searches for securely accessible memories from the processor to which it is mapped. The algorithm traverses all possible private buses and bridges using breadth-first search, until it finds a memory. The key is then mapped to that memory. If all possible secure paths are searched and no memories are found, then a warning is issued saying it is impossible to map keys for that task.

6.8.5 Automatic Generation for Case Study

As previously described in Section 6.7, our current architecture and mapping does not ensure the authenticity of Perception data sent from the Perception task to the Navigation task. Using automatic generation, we indicate that we wish to ensure the strong authenticity of Perception data, and generate the following new model.

Figure 6-28 shows how security operators are added to the activity of the Perception task and Navigation task, and the resulting verification results. As previously described, adding a MAC and nonce can be used to ensure strong and weak authenticity, even if perception data is sent a bus accessible to the attacker.

If however, we remove the memory connected along a secure path, and instead provide one memory along the public bus for the Navigation and Perception tasks as shown in Figure 6-29, the cryptographic key used to calculate MACs becomes accessible to the attacker when either accesses it. Then, after the Nonce is sent along a public channel, the attacker can concatenate the nonce to his/her own forged message, calculate the MAC with the recovered key, concatenate the MAC to the message, and then send it to the Navigation task. As the MAC will match since it was calculated with the correct key, and the nonce will also match, then the Navigation task will accept this forged data message and authenticity does not hold.

```
function mapKeys
input: Mapping Model
output: modified Mapping Model
for Cryptographic Config cc in model do
    for Task t in model do
        if cc ∈ t then
            //Task uses Cryptographic Config, so find closest secure memory
            proc ← Processor containing t toVisit ← empty list of architecture communication
              components
            toVisit.add(All private buses/bridges accessible from proc)
            while toVisit is not empty do
                currentNode = toVisit.pop(0)
                //Remove first element of toVisit
                if currentNode ∈ Memory then
                    Map key to currentNode
                    break
                end
                toVisit.add(All private buses/bridges accessible from proc)
            end
            if no key mapped then
                Warn 'Cannot map key cc for Task t' due to lack of secure memory
            end
        end
    end
end
```

Algorithm 5: Key Mapping Algorithm

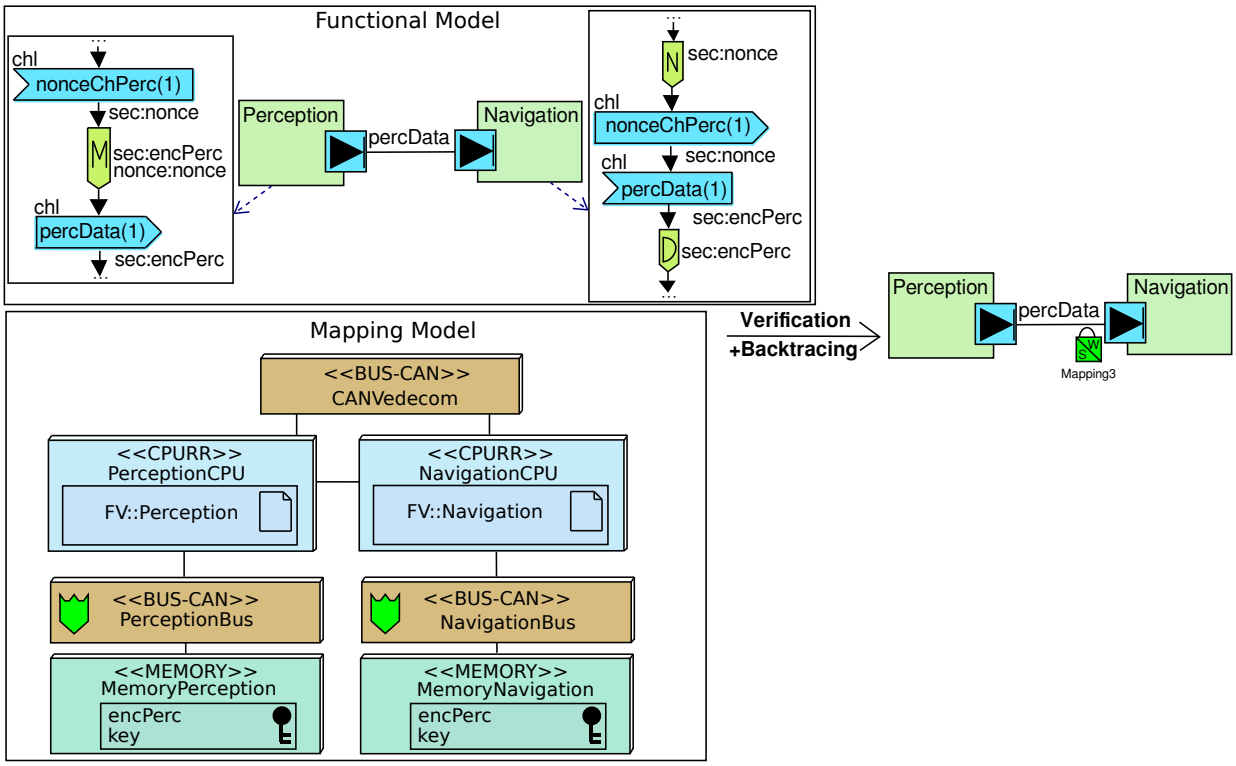


Figure 6-28: Verification Results for Mapping with Security Operators added

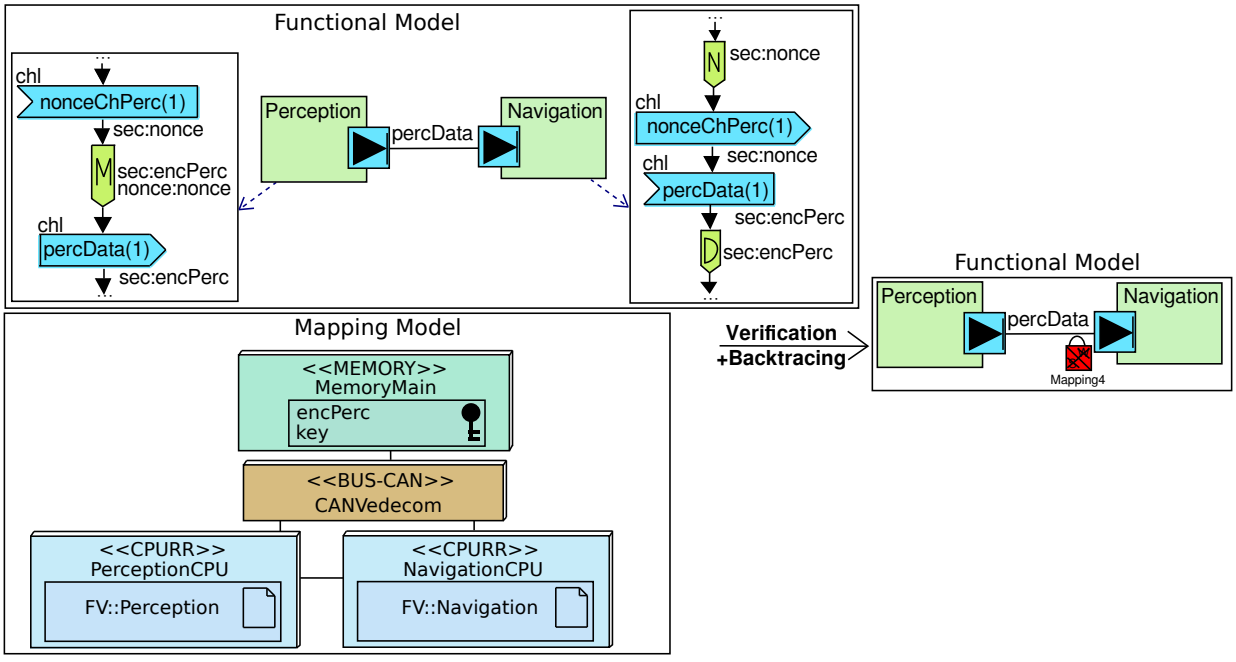


Figure 6-29: Verification Results for Mapping with Security Operators and Insecure Memory Access

6.9 Conclusion

This chapter described the security verification of HW/SW Partitioning models, first by formalizing the model transformation process and proving its correctness, and then by describing how results are interpreted and displayed on the models, and also how our toolkit can take those results to automatically add security operators to correct the unsatisfied security properties. We demonstrate how slight changes to an architecture can result in different security verification results, and thus the importance of adding memories providing secure access to cryptographic keys.

Mapping models are translated into Proverif specifications, and the security annotations are translated into queries which indicate to the prover which security properties we are interested in. Using a recognized prover ProVerif, those indicated security properties are then verified formally. The results of the security verification can then be used to generate a secured model fulfilling all the security properties marked on the model.

However, adding security causes a negative impact on performance. The time to encrypt/decrypt/calculate MACs may cause the system to no longer respond to critical events in time. The next chapter describes the importance of calculating latencies between events.

Chapter 7

Performance Evaluation

> "I wish it need not have happened in my time," said Frodo.
> "So do I," said Gandalf, "and so do all who live to see such times. But that is not for them to decide. All we have to decide is what to do with the time that is given us." –J.R.R. Tolkien, The Fellowship of the Ring

7.1 Introduction

Performance metrics are commonly used to assess the suitability of an architecture and mapping, often supported by automatic Design Space Exploration [226, 257]. The performance of a system is characterized by its timings and percentage usage times of architectural components. As previously discussed, performance is one of the factors impacting the safety of certain systems.

Many real-time safety-critical systems, such as autonomous cars, interact continuously with the environment and users. New input from the outside world is processed by the system, which then effectuates a response observable in the real world. The timing of such responses can greatly impact functionality and safety.

Even if the system is functionally correct, and implemented safety functions ensure safe behavior, those safety functions are inefficacious if they cannot execute in time. Therefore, it is important to ensure the ability of an embedded system to respond to events in a timely manner [218]. Autonomous vehicles especially need to rapidly respond to changes in road and traffic conditions, such as to traffic lights, changes in speed limits, unexpected stopped vehicles, and obstacles in the road. The system must calculate the needed response in time for the vehicle to react correctly, such as braking before reaching a red light, or changing lanes before the vehicle reaches the end of a newly shut down lane.

Timing requirements can vary across real-time systems, where some events must execute before or after a given time, or for less strict systems, the average response time should be below a set time [212]. Even in non-safety-critical systems, a user will not wish to purchase or continue to use a slow or unresponsive device [146]. In other systems, it important to determine the timing profile of a sub-system for it to interact predictably with other components in a system [311].

Latencies also relate to our works on checking the assumptions taken at higher levels of abstraction. As a model should be consistent across multiple levels of abstraction, the latencies measured at each level of abstraction should also correspond, as discrepancies in the measured latencies indicate errors in the abstraction or translations across models. Decisions on architecture and mapping made at the higher level of abstraction are based on estimations of metrics that should be validated on the more refined software design models [118, 119].

In addition, other performance metrics are often related to the latencies measured. The timing characteristics of a system can also be affected by the usage, or load, on processors or communication buses [99]. If a processor is too highly charged, it can lead to undesired delays of lower priority tasks.

In the previous chapters, we discussed countermeasures to ensure the security of a system. The addition of security operations, however, induces a negative effect on the performance of a system. In this chapter, we quantitatively evaluate that performance impact on the set of secured and unsecured mappings. The evaluation involves both assessing the load of the architectural components and verifying the timing of the system. In Chapter 4, we defined one of our requirements to be for the autonomous vehicle can brake in time to avoid an obstacle detected by sensors. This timing requirement serves as a running example for how we perform latency analysis on our model.

7.2 Latency Analysis

Latency analysis is performed in multiple steps. First, the quantitative requirement on latency must be defined. Next, the text requirement must be translated to determine which modeling elements the critical events refer to, after which latencies can be measured in simulation. The results are then conveniently displayed on the modeling diagrams.

7.2.1 Latency Requirements

Before latencies can be measured, we must first define which are the timing requirements of the system. In Section 4.3, we previously described the needs of our section, including latency requirements. Latency requirements are first expressed in Requirement Diagrams as a refined requirement of a safety requirement. The high-level requirement ‘The system should be safe’ is refined to ‘System latencies should preserve safe function’, and further into ‘When an obstacle is detected, a braking order must be issued in time to avoid the obstacle.’

It next becomes necessary to determine the exact latency requirements. Braking distance is calculated as the distance traveled by the vehicle during a response time added to the braking distance of the vehicle. In our simplified example, we assume the maximum distance that an obstacle can be detected is 80 meters, and the vehicle is traveling at a constant 100 km/h in dry road conditions. Braking distance is therefore approximated to be 50 meters [52]. Therefore, the reaction distance is 30 meters, and the maximum time to issue the braking order is 1.08 seconds, though the reaction time should be ideally faster [111].

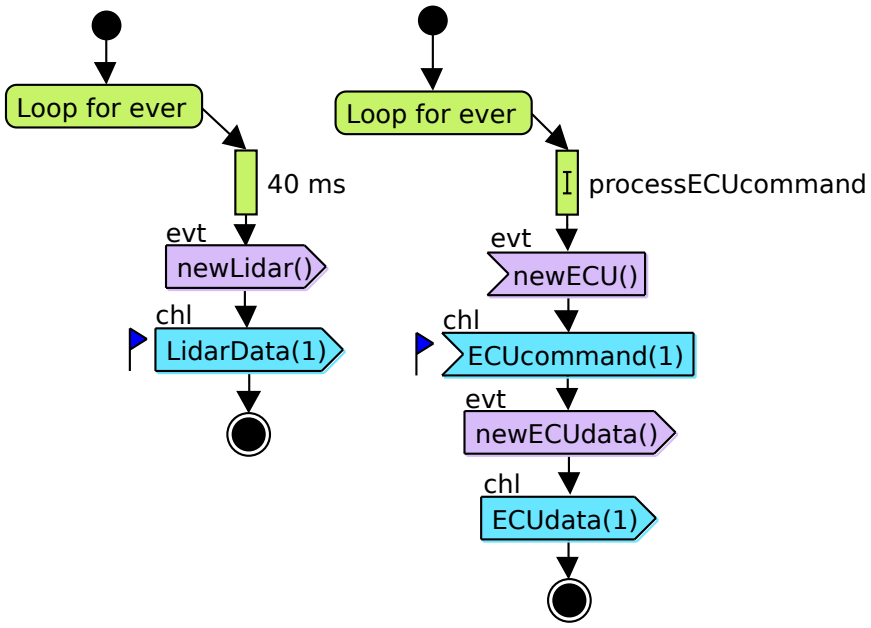


Figure 7-1: Mapping Operators tagged with Latency Checkpoints

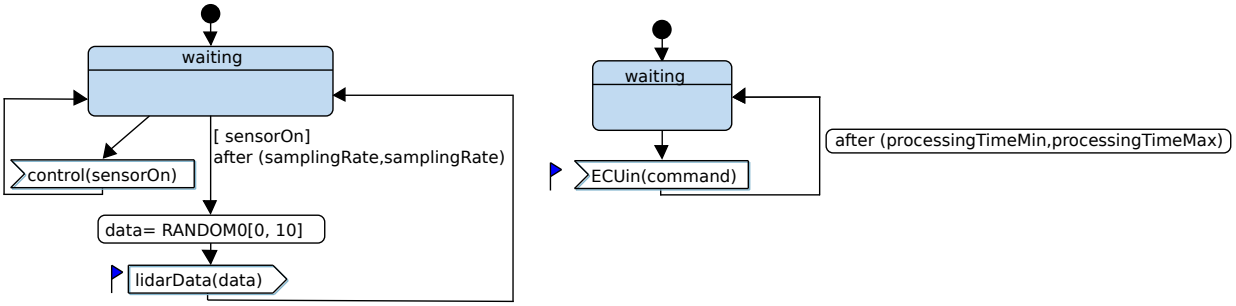


Figure 7-2: Software Design Operators tagged with Latency Checkpoints

7.2.2 Latency Annotations

With the timing requirements defined, the relevant operators must be selected on the modeling diagrams. The events referenced by the requirements should be translated to their equivalent activity diagram event. For example, the obstacle detection event corresponds with the start of sending Lidar Data to the Perception unit, and the issue of a braking order to the vehicle corresponds with the reception of the ECU command by the ECU Gateway. We are assuming that the response time of the ECUs is negligible.

Each relevant operator is tagged as a 'Latency checkpoint', displayed as a blue flag, as shown in Figure 7-1 and 7-2. During simulation, all the possible latency checkpoints can be selected in the Latency window, as shown in Figure 7-3. The min, max, average, and standard deviation of the set of latency values between the two events is displayed for the current simulation run.

In the Software Design Phase, latency queries can also be written in Performance Pragmas, as shown in Figure 7-5. Each pragma statement may check if all instance of operators op1 and op2 meet a certain criteria, such as being less than or greater than a set number. Without a set number provided, statements

may also instead ask for the average latency value.

7.2.3 Latency Analysis

Latencies are measured by averaging the values measured over multiple simulations. The simulation engine, based in C++, is described in [181, 182]. Each architectural component then schedules operations by the tasks or communications mapped on it, which then execute as a transaction over a given time. Each simulation run is stored as a set of transactions executed on the set of architectural components.

Simulations can be run automatically to completion, or interactively in user-defined increments. Users can step through a simulation based on a number of transactions, time units, or commands, or they can run until the next memory access or bus transfer. A step-by-step interactive simulation can help the user examine the detailed execution of a system. While a user could note the execution times of important operators and calculate latencies manually, it would be time consuming to manually record and calculate these latencies, especially if we wish to check the latencies over many iterations.

Before the start of my thesis, latencies could only be manually calculated by recording the execution times during an interactive simulation, or from an execution trace file as shown:

data.txt

```
========= Scheduling for device: SupervisorCPU_0 =========
Supervisor: SelectEvent t:9550034 l:1 vl:1 Ch: newPerc__newPerc
Supervisor: Read 4,percData t:9550035 l:4 vl:4 Ch: percData
Supervisor: Execi 1100000 t:9550039 l:1100000 vl:1100000
========= Scheduling for device: ECU_CPU_0 =========
ECU: Wait newECU__newECU t:18650070 l:1 vl:1 Ch: newECU__newECU
ECU: Read 4,ECUcommand t:18650071 l:4 vl:4 Ch: ECUcommand
ECU: Send newECUdata__newECUdata t:18650075 l:1 vl:1 Ch: newECUdata__newECUdata
ECU: Write 4,ECUdata t:18650076 l:1 vl:4 Ch: ECUdata
========= Scheduling for device: MABX_CPU_0 =========
MABX: Wait newMABX__newMABX t:10650040 l:1 vl:1 Ch: newMABX__newMABX
MABX: Read 4,MABXcommand t:10650041 l:4 vl:4 Ch: MABXcommand
MABX: Send newStatus__newStatus t:10650045 l:1 vl:1 Ch: newStatus__newStatus
========= Scheduling for device: PerceptionCPU_0 =========
Perception: SelectEvent t:8000001 l:5 vl:1 Ch: newLidar__newLidar
Perception: Read 4,LidarData t:8000006 l:5 vl:1 Ch: LidarData
Perception: Read 4,LidarData t:8000012 l:5 vl:1 Ch: LidarData
Perception: Read 4,LidarData t:8000018 l:5 vl:1 Ch: LidarData
Perception: Read 4,LidarData t:8000024 l:5 vl:1 Ch: LidarData
Perception: Execi 160000 t:8000029 l:800000 vl:160000
Perception: Execi 150000 t:8800029 l:750000 vl:150000
Perception: Send newPerc__newPerc t:9550029 l:5 vl:1 Ch: newPerc__newPerc
Perception: Write 4,percData t:9550034 l:1 vl:4 Ch: percData
Perception: Notified newECUdata__newECUdata t:9550035 l:5 vl:1 Ch: newECUdata__newECUdata
Perception: Notified newStatus__newStatus t:9550040 l:5 vl:1 Ch: newStatus__newStatus
```

Extracting the execution times for each operator could be performed by hand or even programmatically, except that multiple operators of the same name are not distinguishable in the trace file. Multiple operators

in an activity diagram can all be read operators for the same channel, for example, and are only distinguishable within the simulation engine by their id. While it would be possible to modify the simulation engine and print the ids in the trace, we decided it would be more helpful to the user to calculate latencies within the simulation window automatically.

Since all of the data required to calculate latencies between operators (operator id + execution time tuples) already exists within the simulation engine, we only needed to extract them into the display window. Each hardware node stores a list of all transactions. A new command was added to request that each node print out the list of transactions which matched the requested ids, and the output transactions were then sent to the simulation user interface for processing. The list of transactions is then processed as shown in Algorithm 6.

For user convenience, we provide a panel for the user to indicate which latencies to measure, which our toolkit then automatically calculates. All the operators marked with latency checkpoints are listed in a drop-down menu, and the user can select between which two operators should latencies be measured.

Given operators $op1$ and $op2$, where occurrences O of the operators $O(op1) = [t1_evt1, t2_evt1, ...]$, and $O(op2) = [t1_evt2, t2_evt2, ...]$ respectively, the list of latencies l_min is calculated as described in Algorithm 6.

input: occurrences of operators op1 and op2 output: set of all latencies between op1 and op2
intialize l={}
for *t_op1 in O(op1)* **do**
  min_latency = MAX_INTEGER;
  for *t_op2 in O(op2)* **do**
    if *t_op2 - t_op1 > 0 t̂_op2 - t_op1 < min_latency* **then**
      min_latency = (t_op2 - t_op1)
    end
  end
end
l.add(min_latency)

Algorithm 6: Algorithm to Calculate Latency for Operators (op1, op2)

As shown in Figure 7-3, the minimum, maximum, average, and standard deviation of the latency values measured in the current simulation run is displayed. These values are measured as cycles relative to the main clock, and can converted to real time in seconds based on the processor frequencies.

7.2.4 Backtracing Latencies

Backtracing latencies directly onto activity and state machine diagrams better helps the user modify the design as it may be difficult to remember all of the different measured latencies for larger designs. These annotations also explicitly mark if requirements are not met or there is an incoherence in the design across different levels of abstraction.

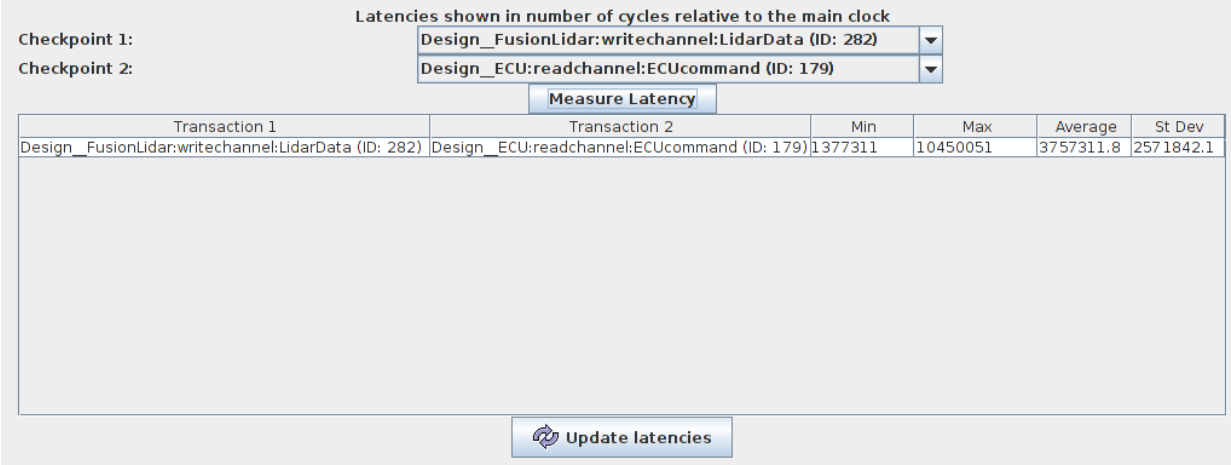


Figure 7-3: Latency Measurement Panel

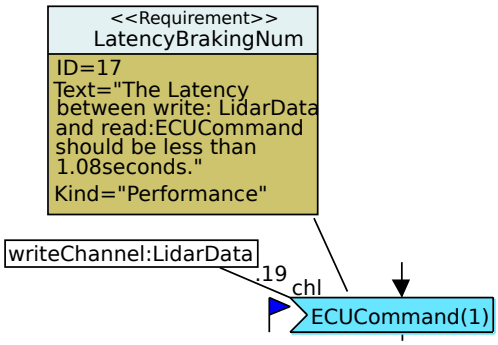


Figure 7-4: Mapping Operator marked with latency measurement and linked Requirement

7.2.4.1 HW/SW Partitioning

Latencies are backtraced to be displayed for each operator. For each latency measurement, the name of the other operator is displayed, along with average value of the latency measurement in seconds.

Operators marked with latency checkpoints can also be directly linked to requirements. If the requirement is formatted in the form "The latency between Operator 1 and Operator 2 should be less than x seconds', and then linked to Operator 1 or Operator 2, the fulfillment of this requirement is automatically displayed after simulation. Latencies that violate timing requirements are marked in red. Figure 7-4 shows the linked requirement for braking, and since it is satisfied, the latency is marked in black instead of red.

This idea of automatically interpreting requirements as verification annotations or queries should be extended to other requirements, as it could be convenient for the designer to have all of the queries set up, and prevent the designer from forgetting to take into account one of the requirements. This idea will be further discussed as Future Work.

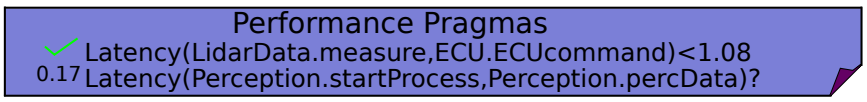


Figure 7-5: Performance Pragma with Latency results

7.2.4.2 Software Design

As the design is refined, the same latency measurements should be checked on the Software Design task.

In Software Design diagrams, both the pragma and latency checkpoints can be annotated. Send/receive signal operators can also be linked to communication operators from the HW/SW Partitioning Phase, so that should the measured latencies differ by over fixed percentage, the labeled latency is also marked in red.

Figure 7-5 shows the pragma annotated with the possible results after simulation. A green check mark indicates that the latency meets requirements, while a red cross indicates that the measured latency does not. Latency queries are annotated with the average latency value.

7.3 Relating Latencies across Levels of Abstraction

The two main levels of abstraction handled in our methodology are at the HW/SW Partitioning phase, and the Software Design phase. In the HW/SW Partitioning Phase, algorithms are abstracted as a Computational Complexity, which can be converted to real time based on the frequency of the hardware component on which it is executed. When the software design model is subsequently developed, it refines the partitioning functional model by adding details of the implementation of algorithms. The algorithm execution time should be similar to the estimated computational complexity. Figure 7-6 shows how latencies are calculated in the HW/SW Partitioning and Software Design Phase.

If a design has been modeled consistently across different levels of abstraction, then the latencies should be similar. A large discrepancy may mean that execution time was not correctly translated from computational complexities, or other performance issues [119]. For example, simulations in Software Design ignore architecture, and do not take into account potential processor and bus contentions that could delay the execution of operations or reception of communications. One potential solution to better take into account the hardware platform is to use the SocLib library [297] for prototyping, as described in [119].

To detect such incoherences, Software Design operators which are latency checkpoints are linked to their corresponding HW/SW Partitioning operator. After simulation on both levels, the latency from HW/SW Partitioning, which is measured in cycles, is converted to actual time, and checked against the latency from Software Design, which is measured in seconds. If the times differ by more than 10%, then the latency is marked in red to indicate an error.

In our example, we examine the measured braking time to determine if the obstacle detection and trajectory calculation algorithm execution times are correctly translated. The average latency, 3757311 cycles, is divided by the clock frequency 20MHz to be equal to 0.19 seconds.

In our preliminary Software Design, the average latency is measured to be 0.12 seconds. As shown in 7-7,

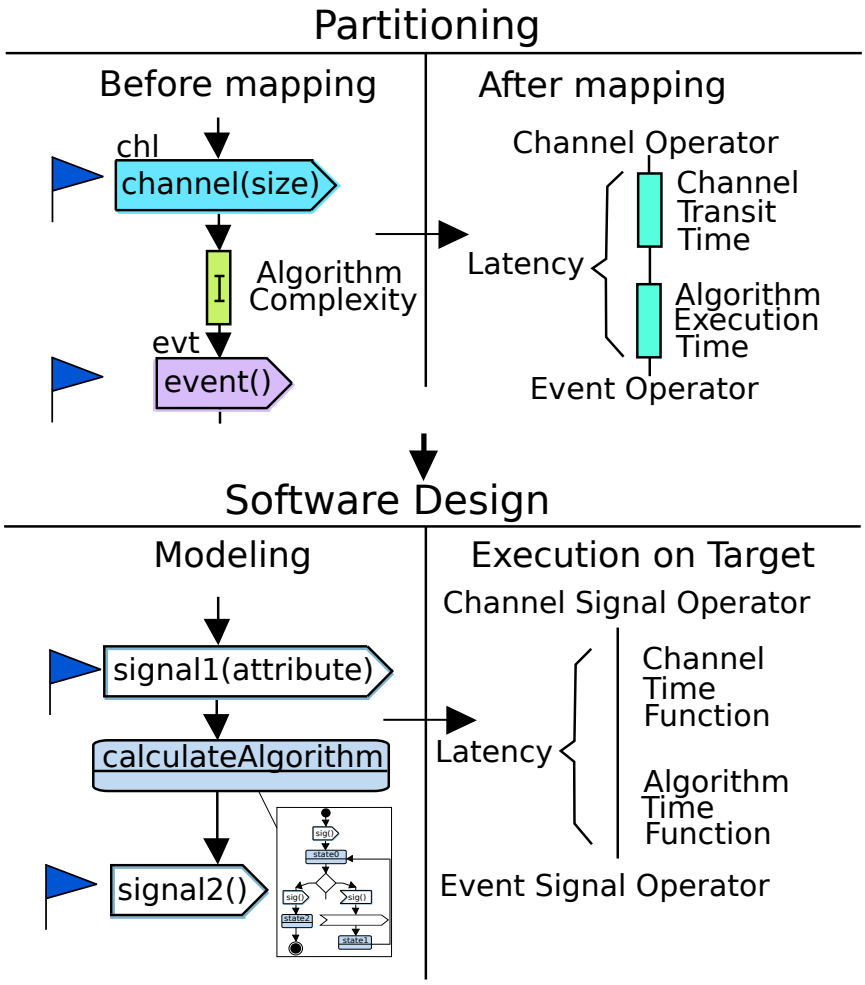


Figure 7-6: Latencies across Mapping vs Software Design

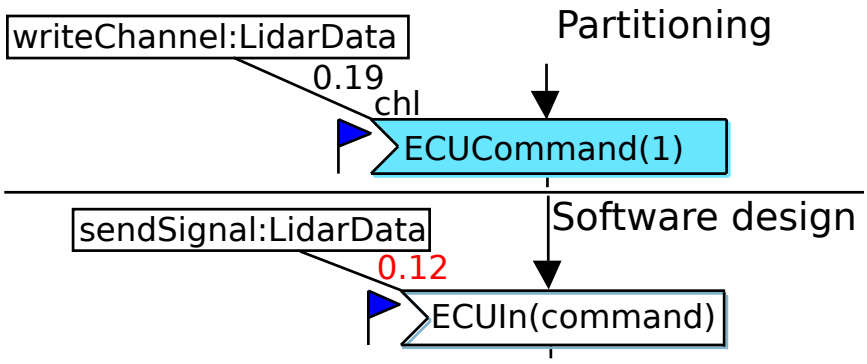


Figure 7-7: Incoherence detected in latency measured between HW/SW Partitioning and Software Design

this latency is marked in red. As the corresponding latencies are significantly different, we should re-check both our models to find where the discrepancy lies.

7.4 Performance Impact due to adding Security

Adding security protocols are expected to lead to degraded performance, due to the additional time required to send longer messages (from added nonces or MACs), exchange keys, and execute security algorithms. The new capability added to measure latencies between events can help us determine if adding security will delay critical events.

We examine the performance of the secured and unsecured Autonomous Vehicle models described in the previous chapters. The default mapping, from Figure 6-22, includes no security mechanisms and does not ensure the security of Perception data. Mapping 2, from Figure 6-23, maps both the Navigation and Perception tasks to a single CPU to ensure Authenticity of Perception data. Mapping 3, from Figure 6-28, uses the same Architecture and Mapping as Mapping 1, but adds security protocols to ensure Authenticity of Perception data. As Hardware Security Modules have been proposed as a solution to mitigate the performance impact due to security protocols, we should examine if they indeed can improve performance for a secured model. Mapping 4, from Figure 5-10, includes security protocols as in Mapping 3, but instead uses a HSM to encrypt perception data.

As described in Section 4.5, the ability to measure the load of each architectural component, or percentage of time a component is active, was previously available in TTool. We also should examine the reaction time of the system to brake following the detection of an obstacle, to check if our secured systems can still react in time to avoid collisions with obstacles. Figure 7-8 shows the load of the different architectural elements on Mapping 1, the default mapping. As all of the components besides the Perception and Navigation CPUs are active less than 1% of the time, we examine only their loads across the other mappings.

Table 7.1 shows the performance results for each mapping, including the max, average, and standard deviation of the braking latencies measured, and the load of the Perception and Navigation CPUs.

The Perception and Navigation tasks are computation intensive, and as expected, mapping both to a single processor results in a highly loaded CPU and increased latency. Adding security (Mapping 3) very slightly increases the average and maximum braking latency. However, the addition of a single Hardware Security Module in this case does not result in significant improvements to performance, as theoretically expected.

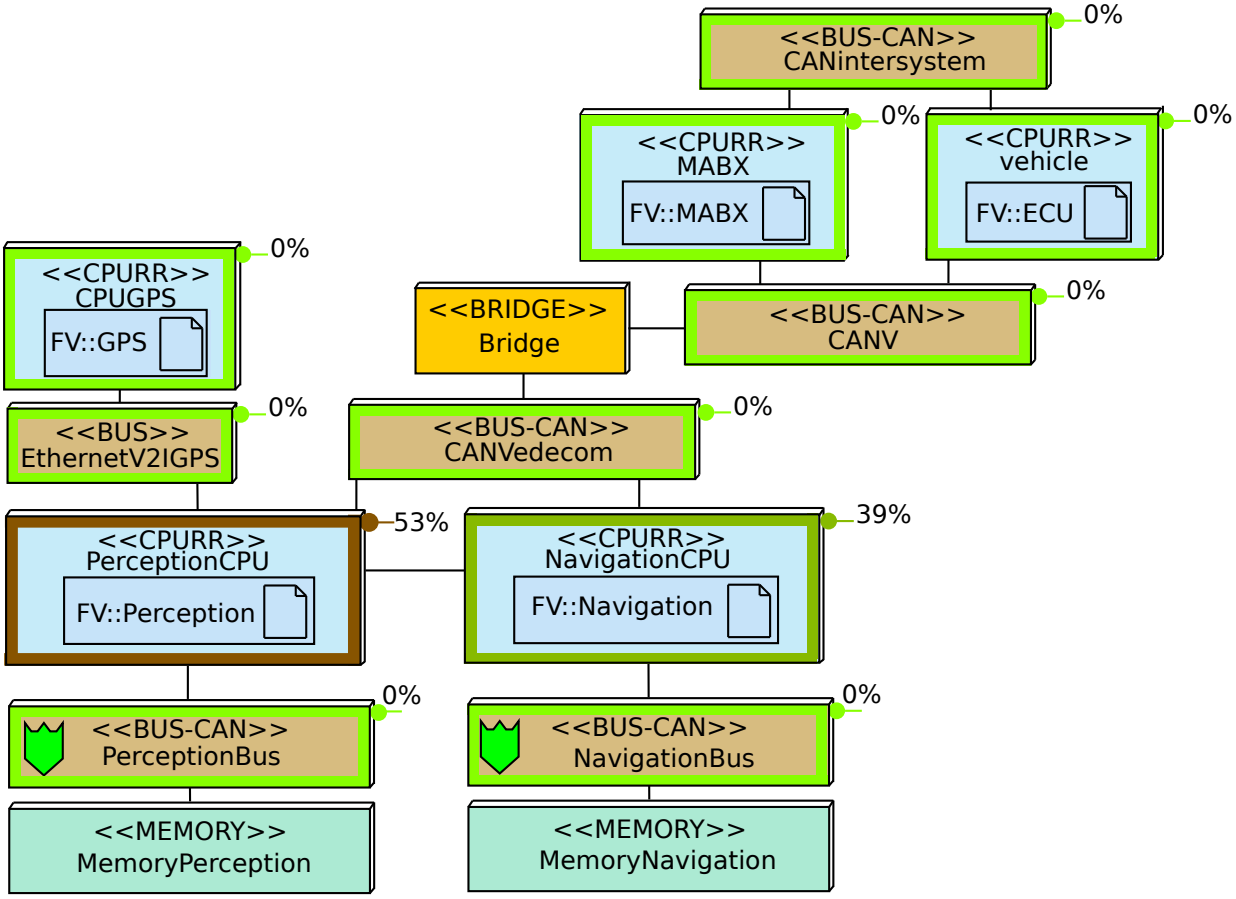


Figure 7-8: Performance results for Mapping 1

Table 7.1: Performance Results over Mappings

Mapping	Braking Latency (s)			Load (%)	
	Max	Average	St. Dev	Perception CPU	Navigation CPU
Mapping 1 (Default)	0.52	0.19	0.13	53	39
Mapping 2 (Nav + Perc on same CPU)	1.15	0.45	0.28	93	-
Mapping 3 (+Security Protocols)	0.53	0.21	0.27	47	35
Mapping 4 (+Security Protocols +HSM)	0.57	0.20	0.15	50	36

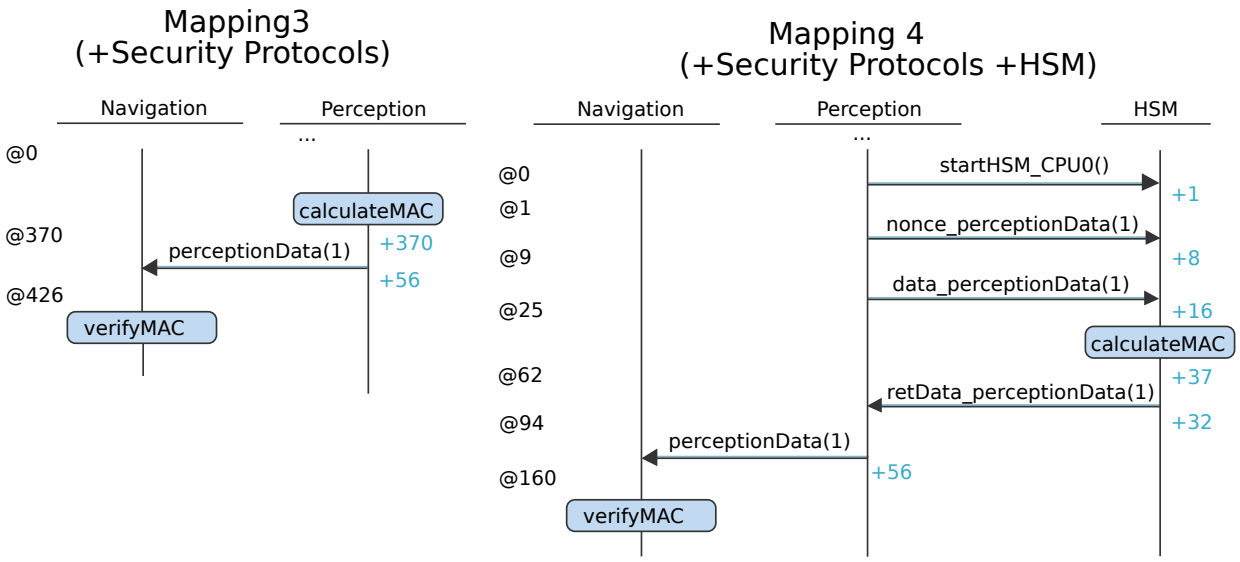


Figure 7-9: Simulation Trace for Performing Security Operations in Perception task or HSM

While the Hardware Security Module performs the security protocols significantly faster than a regular processor, the processor must send the communications to the HSM and then receive the encrypted communications to send. In this case, the majority of the operations performed are calculating the obstacles and trajectory, and the encryption operations are insignificant in comparison. In fact, if the amount of data is trivial, We assumed that the hardware accelerator performs security operations 25x faster than the regular processor, a somewhat optimistic assumption based on the experimental results in [56, 344].

Our estimated data size for perception data is 16 bytes, and data size for the nonce is set to 8 bytes which should be sufficient [63]. With the SHA-256 block size of 512 bits, the operation is performed in a single instance of 370 cycles. Figure 7-9 shows the simulation trace for if the MAC is calculated by the Perception task, or instead by the associated HSM, and annotated with the number of cycles to perform each operation. We assume a bus transfer speed of 1 byte/cycle, with the total transfer time for each communication depending on the message size. For example, the final perception data message sent is 24 bytes (message + nonce) + 32 bytes (MAC) [174]. Synchronization events, such as starting the HSM, are assumed to take 1 cycle. The speedup in these 266 cycles is insignificant, especially as we assumed the perception and navigation algorithms executed in far more than 370 cycles.

If we examine a system in which security algorithms occupy the majority of the execution time, we may find that HSMs have a greater impact on performance. Our future work should examine if these results are indeed accurate by testing on a prototype, and thus, model HSMs more realistically based on experimental data.

7.5 Conclusion

Performance metrics, such as load and latencies, are vital characteristics of a system's behavior. Not only should a system behave correctly, but it must also execute its behaviors in time to respond to safety critical events. Processes can be further delayed if a single hardware component is in charge of executing too many intensive processes or transporting large amounts of data communications.

In our previous work checking the performance impact due to security [201], our timing analysis was limited to comparing total execution cycles of setting the main task to run a fixed number of times. Through our new capability to measure latency, we were able to examine a critical safety property:obstacle response time. The added latency measurements allows for a more precise assessment, as we determine the response times exactly instead of possibly measuring the latencies for events that were not important. For example, measuring the total run time of our example would include the inactive time of the sensors between data collection.

Thus, we evaluated the performance of various configurations of our system, with different mappings and security capabilities, and noted the performance impact of securing a single communication, with the security protocols performed by processors or by a Hardware Security Module. By better examining performance of various mappings, we can better select which will meet all of our requirements.

Chapter 8

Conclusion and Perspectives

> "The future of the safety movement is not so much dependent upon the invention of safety devices as on the improvement of methods of educating people to the ideal of caution and safety."
> - Walter Dill Scott

Connected embedded systems, such as future autonomous vehicles, are expected to bring many conveniences to our lives. Not only does internet connectivity bring entertainment for passengers on a long trip, but it provides us access to the wealth of knowledge available: an exact location with comprehensive driving instructions, locations of restaurants or gas stations for rest stops, and traffic conditions all around us to better help us avoid gridlocks. Drivers no longer need to read a paper map or find detours by hand. Additional safety features, such as automatic braking or parking assist, can further prevent collisions.

However, by adopting these connected and automated features, we cede an amount of control of our vehicle, and instead rely more strongly on software. Accidents in a completely autonomous vehicle will be due to system errors instead of driver errors. Therefore, these systems should be assured to function correctly and safely. As accidents involving autopilot have shown, these systems cannot yet handle all road conditions, and sensors and processing algorithms may fail to correctly perceive the environment correctly [180, 300, 323]. The safety of these systems should be assured through a comprehensive verification process.

The safety of embedded systems relies on multiple aspects. First, the system should be free of software bugs that might cause it to deadlock and stop responding, potentially resulting in dangerous situations such as the vehicle driving straight off the road or stopping on the freeway. Next, the system should be secure against hackers, so that hackers cannot modify the system code, as in Miller and Valasek's hack [223], and inject their own commands into the system. For example, a hacker who can gain control of another's vehicle could provoke a fatal accident, such as turning hard into incoming traffic. A safe system must also take into account its limitations and compensate for them. Since cameras do not function well in low lighting, the system should rely primarily on the other sensors at night. Finally, the safety of the system depends on performance, or timings. Hardware components should not be so highly charged that they cannot perform functions reliably, and critical events, such as obstacle avoidance, should not be delayed so that they execute beyond a strict deadline.

8.1 Integration of full Safety and Security Features into Autonomous Vehicle Model

Throughout this thesis, we demonstrated each of the new capabilities of our approach to improve the safety and security of the Autonomous Vehicle in our running example. We have explained some of the possible countermeasures to add, but we conclude by demonstrating how to integrate a preliminary set we consider necessary to avoid most of the possible hazards.

We conclude by integrating all of the relevant safety and security countermeasures into a single model, and then evaluating its overall safety, security, and performance. In previous chapters, we described that perception data should be ensured authentic to prevent an attacker from injecting false perception data. We extend this requirement to other other critical internal communications as well, such as vehicle status, MABX commands, ECU data, and ECU commands. In addition, authenticity of sensors should be guaranteed. The EVITA project discussed that the system should detect if a genuine sensor was replaced with a faulty or modified sensor without authorization by a malicious individual, and recommended an authentication process to detect that each component was authentic [274]. The EVITA project [274] also suggested that functional internal communications should confidential. Understanding the format of the communications could allow an attacker to reverse-engineering software or recover intellectual property [166, 289]. Therefore, we also require the critical internal communications be verified Confidential. Securing the system requires adding the security countermeasures described in Chapter 5 and 6. As the Navigation task must decrypt or encrypt multiple communications, more than in our previous study in Section 7.4, it may observe a greater performance improvement when supplied with a Hardware Security Module. As in Chapter 5, we use a Firewall to protect against the attacker injection of code.

In addition, as discussed earlier, redundancy of protocols and hardware can better prevent system failure even in the event of a single protocol flaw, lost message, or processor fault. Since the calculation of obstacles is critical, we duplicate that function. The Perception task is then split into a preliminary Perception task which accepts the inputs from the sensors, and then the duplicated second task which calculates the location of each obstacle and the corresponding level of confidence. The Navigation task must then take the 2 sets of Perception data and determine if they are coherent. If the data differ too greatly, then there is likely an error in the system. Another safety check which should be important is on sensor data. Coherence and Plausibility checks can detect and discard sensor data which has been falsified by an attacker, poor data due to sensor limitations, or erroneous data generated by a faulty sensor.

Figures 8-1 and 8-2 show the Functional and Mapping models of the autonomous vehicle respectively, with the described safety and security countermeasures added, along with the security verification results confirming the confidentiality and authenticity of critical internal communications. This model with improved safety and security should protect against many of the faults and attacks described in Chapter 2, such as sensor limitations [180, 216, 229, 295], manipulated sensors [255, 294, 346], and attacker-injected commands and data [67, 223, 337]. This model does not include the security mechanisms for the V2X (pseudonym generator, etc), security protocols on the CAN bus, tamper-resistance mechanisms for sensors (filtering, etc), vulnerabilities such as open ports, or coding details. Future safety and security tests should be performed on first the detailed software design model, and then a developed prototype.

We repeat our performance evaluation from Section 7.4 to check the performance impact due to the additions. We examine both braking latency and the load on the Perception and Navigation CPU(s) as before. Table 8.1 compares the performance results from our base model (with no security or safety counter-

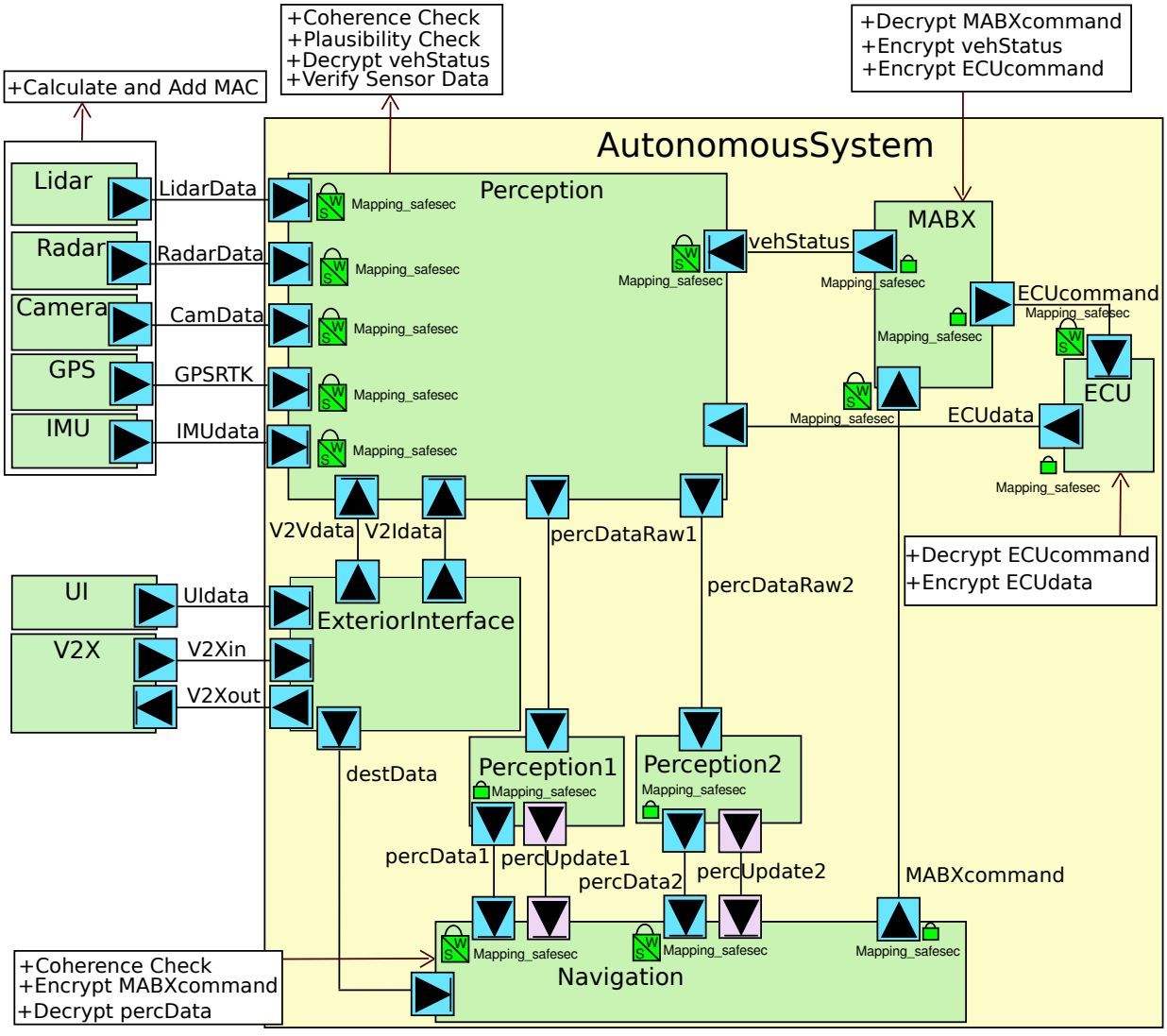


Figure 8-1: Full Application Model with Safety and Security Countermeasures

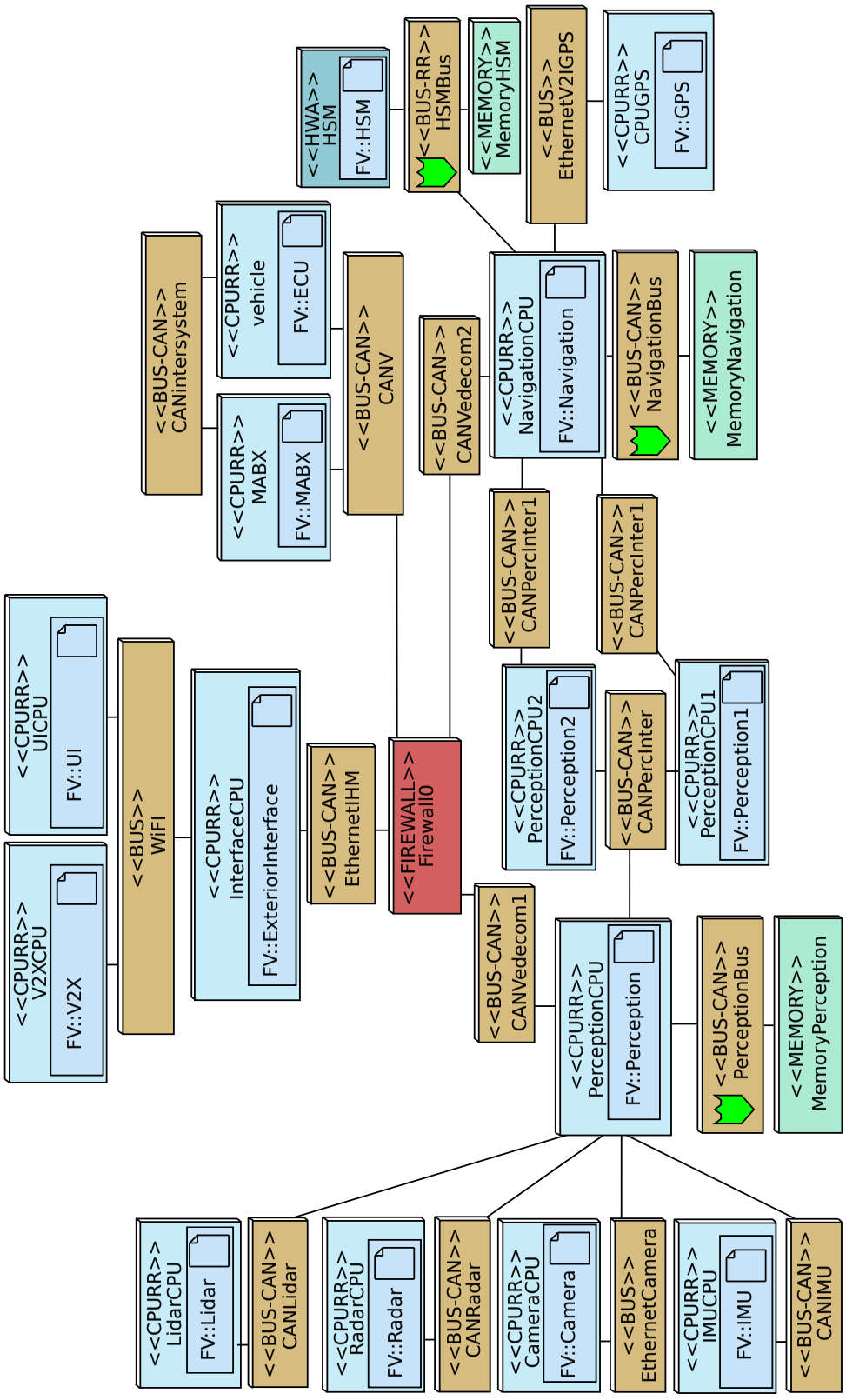


Figure 8-2: Full Mapping Model with Safety and Security Countermeasures

Table 8.1: Performance Results Comparison of Default vs Safe and Secured Mapping

Mapping	Braking Latency (s)			Load (%)			
	Max	Average	St. Dev	Perc CPU	Perc CPU1	Perc CPU2	Nav CPU
Default Mapping	0.52	0.19	0.13	53			39
+Safe +Sec Mapping	1.13	0.46	0.10	4	10	10	42
+Safe +Sec Mapping (Improved Perf.)	0.99	0.39	0.15	5	7	7	36

measures) and our new final safe and secure model. As expected, the braking latency was significantly increased, though the loads on the processors decreased, likely due to breaking up the Perception tasks and idle time waiting for another task to send or receive communications. At this point, the reaction time to brake for an obstacle is while on average less than the 1.08 seconds required, the maximum braking latency exceeds the allowable reaction time. To improve this latency, we could consider further modifying the mapping, such as using better processors, or improving the runtime of the different algorithms. In this case, while this first set of safety and security countermeasures should improve the security of our autonomous vehicle, their addition degraded its performance to which point consequently it could no longer be guaranteed safe.

Therefore, we needed to modify our system so that it would satisfy performance requirements again. We could decrease the braking latency using various approaches, such as using more efficient processors, more efficient security protocols, optimized algorithms, and etc. According to our model, the Perception tasks' algorithms are the most computation intensive, and therefore we targeted them for improvement. As shown in Table 8.1, in the performance results for Safe and Secure Mapping (Improved Performance), the system would satisfy the timing requirements if the computation complexity could be reduced by 10%. This assumption should be validated after re-designing the algorithm in detail in the Software Design phase, aligning with our methodology based on iterations of modeling, verification, and reconsideration steps, which are repeated until all system requirements are satisfied.

8.2 Contributions

This thesis discussed safe and secure design of such systems from a Model-Driven Engineering perspective. Modeling a system helps designers systematically consider all of the system needs, and detect issues in the modeling phases before purchase of hardware or time-intensive development of code. Repairing problems early in these phases thus prevents costly rework or patching products after mass production and distribution.

We started with a summary of the issues faced by autonomous and connected vehicles, and discussed how modeling and verification could support the design of such systems. We presented the capabilities of other methodologies and toolkits, but as none satisfied all of our design needs, we proposed a new methodology. Our new methodology enhances the SysML-Sec Methodology [13] to consider security during the HW/SW Partitioning Phase, allowing for the modeling of attacker capabilities to target an architecture and architecture-based countermeasures, and improved performance analysis by examining the latencies between critical events.

These ideas were actualized and integrated in the modeling toolkit TTool [15]. Throughout this thesis,

we have demonstrated the new capabilities of our toolkit, in modeling, verification, and automatic generation using the Vedecom Autonomous Vehicle as our case study. We have demonstrated how our HW/SW Partitioning models can now be better analyzed to determine the attacker impact, whether through code injection or message tampering, guiding us to add countermeasures automatically such as Hardware Security Modules. Furthermore, we used our new latency measurement capabilities to both check the coherence of models across different levels of abstraction and to more precisely analyze the performance of secured vs unsecured models than in previous publications [201]. The latter analysis helped us better determine the performance impact due to security, and also demonstrated the limitations of HSMs. Without these modeling and verification capabilities, the analysis in the HW/SW Partitioning phase may fail to take into account security requirements and the performance impact of security operations, leading to selection of a non-ideal architecture/mapping. With these new modeling and analysis features, we proposed an architecture/mapping model of the Autonomous Vehicle system with safety and security mechanisms included, and then determined its satisfaction of safety, security, and performance requirements. In conclusion, these improved modeling and verification capabilities help select a mapping satisfying safety, security, and performance requirements, and better provide a better foundation for the subsequent Software Design phase.

8.3 Perspectives

The ultimate goal of this thesis was to determine how to design safe and secure systems. We have provided a methodology and supporting toolkit which should be easy to use, but still cannot address all aspects of safe and secure design.

While this thesis has proposed new aspects for design of embedded systems, there exist significant future work, beyond modeling and verification. One important future research direction is to investigate the practical aspects of security in embedded systems, to determine if our modeling adequately represents systems and their potential vulnerabilities.

8.3.1 Security for Embedded Systems in Practice

As this thesis is primarily theoretical, future work should involve determining if our modeling approach is sufficient to represent security vulnerabilities and countermeasures, and our supported verification tools can indeed detect actual security flaws in real-world systems. Our HW/SW Partitioning phase, for example, is highly abstract, and we should ensure that a secure model is indeed secure in practice. For example, many of the demonstrated hacks [67, 223] involved entering the system through an open port, which we should determine if this possibility is correctly represented by modifiable code and public buses.

8.3.2 Accurate Representation of Countermeasures

Theoretically, Hardware Security Modules should greatly improve performance. However, experimental results have shown variable improvements, with the HSM from the EVITA project showing a range of experimental results on different security algorithms, from no improvement to a 25x speedup [344], and [56] showing 2-60x speedup for the security functions, resulting in 2-9x speedups of the entire algorithm.

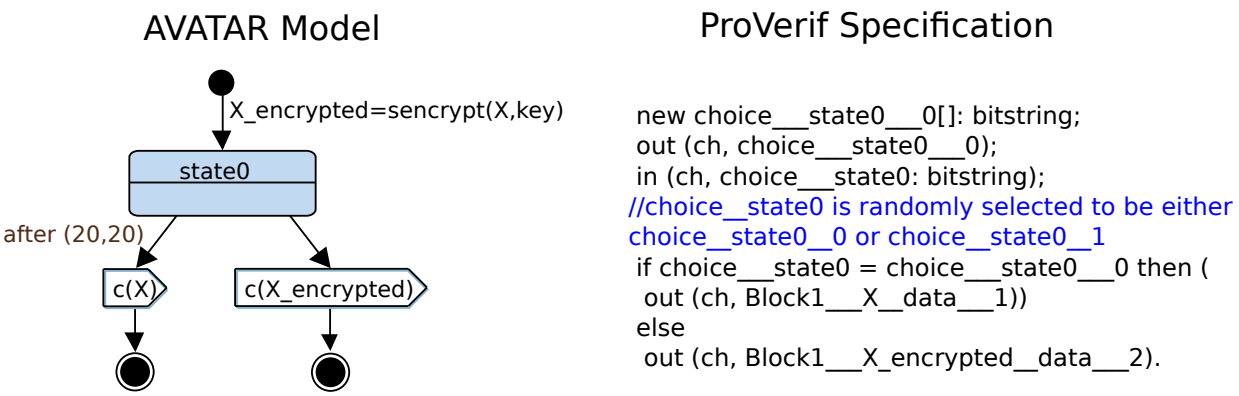


Figure 8-3: AVATAR Model translated incorrectly to ProVerif due to removal of time

We should investigate with actual Hardware Security Modules to see their impact of performance, and thus, more accurately represent HSMs.

Other potential future work enhance our ideas on modeling and verification, including future development for our toolkit.

8.3.3 Full Automatic Generation of Countermeasures

The ultimate automatic generation tool would take in the set of requirements, a base model, and then add all countermeasures to fulfill all of these requirements. At this point, our current automatic security generation tool adds security operators and Hardware Security Modules to fulfill marked security properties. These additions, however, may cause the model to no longer meet timing and performance requirements. Given an accurate set of information regarding the execution time of different encryption algorithms, our tool could suggest which algorithm would fulfill timing requirements, or suggest that a Hardware Security Module be added. However, in our case study, we determined that Hardware Security Modules were not effective due to the minimal security operations performed, and our toolkit should be able to better analyze the model to determine when adding HSMs would be effective.

8.3.4 Security Modeling and Verification

Currently, our security verification focuses on determining if an attacker can access specific sensitive data or modify program code. For all that ProVerif has served our purpose, it is possible that other security provers may bring their own advantages. For example, only limited mathematical notation can be used in ProVerif.

This thesis touches upon tamper-proof program execution, but decided that no hardware or software countermeasure could definitively allow us to classify a task as tamper-proof. However, the future developers of TTool should consider keeping track of future research in this direction.

8.3.5 Time in ProVerif

We currently disregard all temporal operators for the translation to ProVerif, as the security prover has no concept of time. However, in certain cases, this simplification can lead to the prover wrongfully determining that sensitive data is recoverable by the attacker when in fact it is not. For example, Figure 8-3 shows an Avatar State Machine Diagram where the next action is to send the data X after t seconds or the encrypted form of X, along a channel c which we assume public. By removing the $after(t)$ action, ProVerif interprets the behavior of the system to be sending X or X_encrypted, for which the Confidentiality of X is proved false. However, in AVATAR, the system will take the first available action, which would be to send X_encrypted. Therefore, in reality, X would never be sent along the public channel, so the Confidentiality of X is actually true in the model. Future work should consider how to best add the concepts of unreachability due to time into our translation.

8.3.6 Safety Countermeasure Modeling

While TTool supports the modeling of safety countermeasures like Redundancy and Coherence checks, unlike security countermeasures, they cannot yet be added automatically like security countermeasures can. The automatic addition of these common countermeasures would reduce design time and manual work, making modeling more efficient.

8.3.7 Safety and Security Analysis Diagrams

As discussed in Chapter 3, various works have proposed combining safety and security analysis. While in our example, fault trees and attack trees can be modeled separately, certain faults may make an attack more likely, and our modeling diagrams should investigate how to connect them.

8.3.8 Relationship between Safety, Security, and Performance

Considering the interactions between Safety and Security is important to fulfill both types of requirements. We have discussed some aspects of how adding countermeasures affects each, but there is more analysis and research to be done, especially on how to better automate the process. Furthermore, while our security analyzer can determine which security properties on data and communications are violated, it would be helpful to extrapolate the ultimate effect on the system safety based on the security analysis results.

8.3.9 System Resilience

While our current safety verification can check the correctness of our design and if it avoids certain unsafe situations, we do not yet support fault resistance analysis. Currently, during Software Design, our toolkit can simulate the effect of lossy communications, we propose expanding these fault simulations to allow for faulty architectural components, such as lossy buses or failing processors, and faulty communications, such as across error-prone channels.

8.3.10 Vulnerability Modeling

TTool's current modeling of vulnerabilities are limited to communications accessible to an attacker and tasks which can be modified by an attacker. Works like [93, 195] offered more detailed vulnerability modeling. Based on specific commercial components selected, these tools looked up the list of discovered vulnerabilities for each component. This idea could better guide the development of attack trees with the exact vulnerabilities and corresponding attack steps that can be carried out on the system.

8.3.11 Improved Connections between Phases

This paper noted that security requirements, attacks, and faults in the Analysis phase can be related to certain countermeasures and verifications in the later phases. However, these links are not yet complete, and with the exception of limited latency requirements, requirements are not yet automatically translated into verification annotations or pragmas. This automation is another proposed area of future work.

8.3.12 Integration of Security Verification Results

Currently, Confidentiality and Authenticity properties are formally checked with ProVerif, while Availability properties are checked informally with Attacker Scenarios and our simulation engine. While ProVerif does not handle verification of Availability or consider performance parameters, and our simulation engine is not yet formal and ignores cryptographic operations, we could look into methods of better harmonizing these two analysis methods. We could potentially either find another prover considering all 3 properties, or adapt either our simulator to take into account security properties, or extend ProVerif to consider Availability.

8.3.13 Proof of Correctness for Authenticity

In this thesis and [208], we have proved that the model transformations to ProVerif specifications maintain that if a variable is not Confidential in the model, then it will be verified non confidential in the ProVerif specification. In future work, we should also prove the correctness of the transformation for both the Strong and Weak Authenticity properties.

8.3.14 Attack Probabilities

Security properties in TTool are currently either proved true or false, or unproved, does not take into account brute force or side channel attacks, and assumes that keys cannot be guessed. However, as other works have provided times for attack and attack probabilities based on parameters such as key size, security algorithm, etc. Many agencies and researchers have provided recommendations for key size [36, 196], and as encryption/decryption time varies with key size, it would be helpful to indicate to the user if their selected key size/algorithm is acceptable security practice, or if a brute force attack break it too quickly [108]. However, it is worth noting that recommended key size, algorithms, and password cracking times change over time [30, 105], and they may need to be updated yearly.

Chapter 9

Resume

La vérité vaut bien qu'on passe quelques années sans la trouver.
–Jules Renard

9.1 Introduction

Les systèmes embarqués et les dispositifs connectés sont de plus en plus répandus dans notre vie quotidienne [96]. Ces systèmes, qui comportent des composants matériels et logiciels, contiennent un ordinateur "intégré" dans l'appareil et sont conçus pour exécuter une seule fonction dédiée [25]. La connectivité omniprésente de ces objets a amélioré notre vie quotidienne, avec l'ajout de commodités dans notre maison [263], et l'amélioration de la sécurité dans nos trajets quotidiens (connectivité Internet, surveillance des voitures, freinage d'urgence, etc) [342]. Malgré tous les avantages dus à ces dispositifs connectés, leurs dysfonctionnements posent des impacts graves sur la vie privée ou la sécurité personnelle.

Par exemple, des machines de radiothérapie devraient aider à traiter le cancer, mais les bogues logiciels ont rendu malades ou tué des dizaines de patients [121, 198]. Des autres produits quotidiens ont été rappelés en raison de risques sur la sûreté [6, 176]. Des failles de sécurité ont également été trouvées sur des appareils médicaux, comme la pompe à médicaments Symbiq de Hospira [153]. Les chercheurs ont démontré comment prendre le contrôle des voitures connectées grâce à la connectivité cellulaire et wifi [67, 223], et des drones grâce à une connexion à distance non sécurisée [268]. Les attaques ont également visé d'importants systèmes industriels, comme en témoignent les attaques Stuxnet, Flame et Duqu [219]. Tous ces exemples démontrent les risques de sûreté posés par les failles ou les vulnérabilités des systèmes embarqués et connectés.

La sûreté est définie comme l'évitement de situations qui peuvent causer des pertes telles que des blessures corporelles, des maladies, des dommages matériels, des dommages financiers, etc. Un système devrait être exempt de défauts, qui sont définis comme des états système indésirables dus soit à des commandes incorrectes ou à l'absence de la commande correcte, soit à une défaillance, qui est définie comme l'incapacité du système ou de l'élément du système à remplir sa fonction prévue [62]. Dans notre contexte, nous divisons la sûreté du système en multiples aspects : la sûreté conventionnelle qui consiste à éviter les dysfonction-

nements entraînant des pertes, comme les errurs de programmation de l'appareil Therac-25 qui pourraient occasionner une dose fatale de rayonnement aux patients, et la sûreté de la fonctionnalité prévue qui consiste à éviter les pertes dues aux conditions environnementales, même dans un système sans défaut, comme le mauvais fonctionnement des capteurs dans des conditions météorologiques défavorables [31, 154, 198].

Les systèmes en temps réel impliquent un logiciel de contrôle continu qui commande des composants physiques, qui fonctionnent avec des contraintes de temps pour une fonction correcte [123]. Dans certains systèmes en temps réel, la performance peut également être essentielle à la sûreté [64, 243]. Les événements critiques ne devraient pas être retardés en raison d'un conflit d'accès au bus ou au processeur d'un système embarqué, car de telles contentions peuvent créer des situations potentiellement dangereuses comme des dommages au système ou aux utilisateurs [9, 191].

De plus, la sûreté d'un système dépend de sa sécurité. Selon le chercheur en sécurité Charlie Miller, qui a démontré le piratage à distance d'une Jeep à travers le réseau cellulaire, "Vous ne pouvez pas avoir la sûreté sans sécurité" [128]. Même si un système a été conçu pour être complètement sûr, si un pirate accède au système, il pourrait changer la fonctionnalité entièrement d'un système qui ne met en oeuvre des contrôles de sûreté.

La conception de systèmes embarqués sûrs est compliquée en raison de leurs nombreuses exigences et de la présence de composants matériels et logiciels [145]. Non seulement nous devons nous assurer que le système se comportera toujours en toute sûreté et qu'il est protégé contre les attaquants, mais nous devons également tenir compte de la performance en temps réel pour les dispositifs critiques du point de vue de la synchronisation, de la mémoire, de la durée de vie des dispositifs, du coût et de la taille de l'architecture, de la fiabilité et enfin de la consommation d'énergie pour les systèmes sur batterie [25, 184].

La conception de systèmes sécurisés est compliquée par le manque d'expertise en sécurité des développeurs [20,284,306,306], et le fait que les mécanismes de sécurité sont souvent ajoutés une fois le système déjà sur le marché [98]. En même temps, la conception de systèmes sûrs est compliquée par la nécessité d'assurer à la fois l'exactitude fonctionnelle du logiciel dans une variété d'environnements et la capacité du matériel à supporter les fonctions de sécurité [144].

De nombreuses solutions pour assurer la sûreté et la sécurité ont été proposées, telles que la modélisation, les tests, diverses méthodologies et le respect de normes industrielles. Une solution, la modélisation systématique et la vérification formelle, peut aider à détecter les défauts plus tôt, à préciser le système et à mieux analyser l'ensemble du système, ce que les tests individuels ne peuvent pas faire [290]. Réparer les défauts logiciels plus tôt dans le processus de conception coûte moins cher qu'après la production de masse [137]. La vérification formelle et la simulation peuvent être effectuées sur les modèles plus abstraits pour valider la conception [91], car les systèmes complets peuvent être trop grands pour être modélisés et prendre trop de temps à vérifier, souvent décrits comme le problème de l'explosion d'état [61]. Les concepteurs affinent ensuite de façon itérative les modèles abstraits jusqu'à ce que les modèles incluent tous les détails importants [340]. Les modèles finaux peuvent alors être automatiquement traduits en code généré [192, 331], ce qui assure la corrélation entre le système et les modèles, et facilite le processus de développement logiciel [290].

Ma thèse, financée par l'Institut Vedecom, l'institut de recherche pour le développement de véhicules durables et autonomes, étudie la conception de systèmes embarqués sûrs et sécurisés. La conception de leur véhicule autonome doit tenir compte d'une multitude d'exigences, notamment la sûreté des occupants et des autres passants. L'une des étapes critiques du développement est de décider du matériel et des logiciels de haut niveau, et du partitionnement du logiciel avec le matériel (déterminer les composants

matériels où les fonctions sont exécutées) [217, 236, 316].

Comme présenté dans cette thèse, il existe plusieurs méthodologies et outils de conception, chacune se concentrant sur certains aspects de la conception ou des domaines spécifiques, mais aucune ne supporte les outils de vérification nécessaires et gère la modélisation de la sécurité requise, en particulier lors de la sélection d'une architecture et des allocations de fonctions sur l'architecture. Nous faisons donc évoluer la méthodologie SysML-Sec et l'outil de support TTool pour mieux répondre à ces besoins [13, 15]. Comme avantage de notre approche, le fait de garder l'ensemble de la modélisation dans une seule boîte à outils permet de s'assurer qu'il n'y a qu'un seul ensemble de modèles et de minimiser le nombre de reconsidérations à chaque changement [157]. En même temps, nous assurons la facilité d'utilisation de notre boîte à outils, une qualité essentielle pour assurer son adoption par les concepteurs [235]. Nous discutons aussi de nos efforts pour une assurer une présentation claire des résultats de vérification au niveau des modèles, afin de faire gagner du temps au concepteur en identifiant automatiquement les exigences auxquelles un modèle ne répond pas.

Les contributions de cette thèse sont la proposition d'une nouvelle methodologie de modélisation et de verification, dont la modelisation de sécurité pendant le phase d'allocation logicielle/matérielle, et les mesures de latences (performance).

9.2 Contexte

9.2.1 Sûreté et Sécurité des Voitures Autonomes/Connectés

L'introduction des voitures autonomes devrait réduire les accidents, faciliter la circulation, diminuer la pollution, offrir des services de transport aux personnes handicapées, aux personnes âgées et aux enfants, et aux autres qui ne peuvent pas conduire, et changer l'essence même de nos trajets quotidiens [101]. Contrairement aux véhicules conventionnels, les véhicules autonomes s'appuieront entièrement sur des logiciels et des capteurs, au lieu de prendre des décisions humaines potentiellement erronées pour le contrôle. Pour assurer la sécurité des passagers et des autres personnes à proximité, les constructeurs doivent assurer le fonctionnement sûr et sécurisé du logiciel du véhicule.

Comme pour les autres dispositifs embarqués, les véhicules n'ont pas été exempts de défauts de sécurité et de sûreté, comme cela a été démontré ces années. Pire encore, alors que la connectivité accrue des véhicules a offert de nouvelles facilité pour la sûreté, elle a également créé des possibilités d'attaque pour les pirates informatiques.

Le tableau 9.1 résume les risques pour la sécurité et la sûreté pour les véhicules connectés et autonomes.

9.2.2 Contre-mesures proposées

De nombreux projets de recherche proposent de prendre en compte ces problèmes: nous les présentons en fonction des contre-mesures possibles. Bien qu'il existe d'innombrables méthodes pour assurer la sûreté d'un système, nous nous concentrons sur les principales méthodes utilisées dans le prototype Vedecom.

Les contrôles de cohérence peuvent aider à garantir le bon fonctionnement d'un système malgré des données de capteurs falsifiées ou erronées. S'il y a une discordance significative dans les capteurs, alors il est

Table 9.1: Tableau de Risques dans les Voitures Autonomes/Connectées

Risque	Attaque/Défaut	Reférence
Contrôle à distance du véhicule par l'attaquant	Réseaux Wifi/3G network	[67, 223, 337]
Contrôle du véhicule par l'attaquant	Unité télématique attachée au port OBD-II	[109, 188, 345]
Arrêt des fonctions de sécurité	Attaque par déni de service sur le bus CAN	[246]
Données des capteurs falsifiées	Camera/Lidar/Radar	[255, 294, 346]
Signaux GPS falsifiés	GPS	[149]
Atteinte à la vie privée	Appli Smartphone App/Capteur Pneu	[150, 159]
Mauvaise interprétation des panneaux de signalisation routière	Modification de panneau	[122]
Mauvaises Données Lidar/Radar	Neige / Pluie	[216, 229]
Mauvaises Données Caméra	Ténèbres / éblouissement solaire	[180, 295]
Perte de Signaux GPS	Bâtiments / Tunnels	[66]
Mauvaise infrastructure routière	Marquage des voies endommagées / Panne d'électricité	[216]

possible qu'un attaquant usurpe l'un des capteurs, ou qu'un des capteurs ne fonctionne pas correctement, et donc l'utilisateur devrait être averti.

Les contrôles de plausibilité tiennent compte de la plage possible des valeurs et des données historiques pour filtrer les données d'entrée. Par exemple, un saut extrême dans la position actuelle à partir du GPS dans un court intervalle est impossible, et est probablement dû à un dysfonctionnement ou à un problème de réception. Le boîtier MABX, qui convertit les commandes du véhicule pour les ECUs, effectue également un filtrage pour des raisons de sûreté. Les commandes du véhicule peuvent être acceptées ou rejetées en fonction de l'accélération maximale autorisée, du freinage, du virage, de la vitesse actuelle, etc.

La redondance des fonctions vitales et des capteurs permet d'assurer le fonctionnement du système dans les composants critiques. Même en cas de dysfonctionnement d'un composant, le véhicule doit continuer à fonctionner en toute sûreté ou entrer en mode de sûreté en avertissant les occupants et en se dirigeant vers un arrêt sûr. La fonction principale de la voiture autonome est l'unité Perception, qui recueille toutes les données des capteurs et génère l'ensemble des obstacles dans la zone environnante. Quelle que soit la rigueur des tests, il existe de nombreuses combinaisons de mesures de capteurs et d'obstacles, et tout algorithme de perception peut encore avoir des défauts. De plus, en cas de défaillance du processeur ou du bus de communication, aucune donnée de perception ne serait envoyée au superviseur, ce qui empêcherait le système de continuer à fonctionner.

Nos contre-mesures de sécurité primaires protègent notre système ou nos données contre un attaquant. Certaines communications internes du système peuvent être accessibles à l'attaquant, il est donc important de s'assurer que l'attaquant ne peut pas récupérer ou altérer des données importantes qui pourraient donner lieu à des situations indésirables. Les mécanismes de cryptage empêchent un attaquant de récupérer (et de comprendre) certaines données. Les codes d'authentification de message peuvent être ajoutés à un message afin que le destinataire puisse déterminer que le message n'a pas été modifié. Les horodatages et les nonces peuvent également être utilisés pour empêcher que des messages en double ne soient reçus et

acceptés lors d'une rediffusion.

Des modules de sécurité matérielle (HSM) ont été suggérés dans les projets européen EVITA, PRESERVE, SEVECOM et d'autres travaux sur la sécurité des systèmes embarqués [92]. Ils définissent des protocoles de sécurité et protègent les clés cryptographiques qu'ils contiennent, conformément à l'exigence proposée de mémoires sécurisées pour la cryptographie. En outre, ils peuvent contenir des accélérateurs cryptographiques qui effectuent le chiffrement plus rapidement que les processeurs ordinaires. Cependant, il s'agit d'un composant matériel supplémentaire ajouté au système. Les modules de sécurité du matériel commercial peuvent être basés sur ARM Trust Zone, Infineon Aurix HSM, etc, dont certains sont compatibles avec les spécifications issues d'EVITA [293].

Les pare-feu séparent les différents sous-systèmes avec des niveaux de sécurité différents. Ils peuvent isoler les communications non fiables pour les empêcher d'accéder à des systèmes internes critiques. Les pare-feu peuvent être des pare-feu matériels ou logiciels, ou les deux. [70] a discuté de la façon dont les pare-feu peuvent être mis en œuvre dans le matériel afin de protéger un circuit intégré.

Les systèmes de détection d'intrusion ou d'anomalie peuvent détecter des attaques ou des dysfonctionnements de composants [173]. Si une attaque est détectée, le système peut avertir l'utilisateur ou entrer en mode failsafe. Des produits commerciaux sont également disponibles pour les constructeurs automobiles, comme le logiciel de détection d'anomalie de Symantec [45].

Une autre considération importante en matière de sécurité est d'assurer l'intégrité du code, de sorte que le logiciel lui-même ne peut pas être modifié. Comme certaines attaques démontrées ont impliqué la modification du code système, les techniques de signature de code et les environnements d'exécution sécurisée devraient prévenir ces attaques.

Bien qu'il existe d'autres mécanismes et protocoles de sécurité plus spécifiques, cette thèse se concentre sur la façon de modéliser abstraitement la sécurité pour la sélection d'une architecture sûre et sécurisée.

9.2.3 Effets secondaires des contre-mesures pour la sûreté, la sécurité et la performance

Une voiture autonome est un système dans lequel la sûreté est particulièrement critique, car tout dysfonctionnement peut entraîner de graves dommages financiers ou corporels. Sa conception implique donc un examenation attentif des différents mécanismes permettant d'améliorer la sûreté du système. Cependant, nous notons que l'ajout de ces contre-mesures peut avoir des effets secondaires, car la correction d'un défaut peut entraîner une cascade malheureuse de corrections supplémentaires.

Par exemple, l'ajout du chiffrement ou de l'authentification des données améliore la sécurité d'un système et devrait améliorer la sûreté en prévenant les comportements dangereux induits par les attaquants. Cependant, le temps supplémentaire pour sécuriser les données dégrade les performances et peut retarder les événements critiques pour la sécurité.

Même les pare-feu, qui filtrent les communications et devraient empêcher les communications générées par des pirates informatiques, continueront d'appliquer un certain délai aux communications. Comme la protection des données ou le filtrage entraînera un retard, il y aura un effet négatif sur la performance et un effet inconnu sur la sécurité.

Pour nos contre-mesures de sécurité, un contrôle de cohérence peut empêcher l'impact d'un attaquant s'il détecte une incohérence entre les données injectées et les données correctes, mais seulement si l'attaquant

ne peut pas accéder facilement à tous les composants en même temps. Il peut donc être utile de sécuriser les données avec des algorithmes de cryptage et des clés différentes pour empêcher un attaquant d'accéder facilement aux deux ensembles de données. En outre, le retard dû au contrôle de cohérence peut affecter les performances.

Les modes de sûreté peuvent s'activer lorsque le système détecte un problème de sûreté, comme une défaillance matérielle, ou un problème de sécurité, comme une attaque. Bien qu'ils soient destinés à améliorer la sûreté du système, leur effet dépend également de leur mise en œuvre, car le mode dégradé peut impliquer la suppression de certains protocoles de sécurité, ce qui rend le système moins sûr en fin de compte.

De nombreuses caractéristiques supplémentaires pour les véhicules connectés devraient améliorer la sûreté, comme le freinage automatique, les systèmes V2X qui peuvent signaler si une voiture avant freine, etc. Cependant, cette connectivité accrue a eu un impact négatif sur la sécurité de ces systèmes, en ajoutant de nouvelles voies d'attaque. Aucun piratage ne peut être effectué sur un système complètement isolé.

En prenant en compte les limites des capteurs (notamment en raison de conditions environnementales), nous nous rendons compte qu'une voiture autonome ne peut pas dépendre d'un seul capteur pour percevoir le monde qui l'entoure. Fourniture de capteurs supplémentaires améliore notre algorithme de perception, mais reçoit et traite toutes les données, puis le calcul des cohérences utilise des cycles d'horloge supplémentaires. De la même manière, d'autres contrôles de sécurité, tels que la surveillance ou les timers de type "watchdog" nécessitent également du matériel ou des logiciels supplémentaires, et peuvent avoir un impact sur les performances [280].

Ainsi, l'effet global sur le système de l'ajout d'une seule contre-mesure n'est souvent pas clair. Cette interaction entre la sécurité et la sûreté démontre les complications liées à l'ajout de mécanismes visant à améliorer la sécurité et la sûreté. Le fait que l'ajout d'une contre-mesure pour s'assurer qu'une exigence peut avoir mené à la violation d'une autre exigence démontre l'importance des itérations de modélisation et de vérification jusqu'à ce que le système soit vérifié et que toutes les exigences soient satisfaites.

Pour bien ajouter les contre-mesures, qu'elles soient réalisées en logiciel ou matériel, on a besoin de méthodologies et d'outils avec des capacités multiples. La conception complète d'un système se fait en plusieurs phases à plusieurs niveaux d'abstraction [262], car les systèmes embarqués du monde réel peuvent être trop complexes pour être conçus tous à la fois [278, 283]. Avant qu'un système puisse être conçu, il est nécessaire de décrire les exigences du système, de considérer quelles attaques et quels facteurs environnementaux le système devrait prendre en compte. Lors de la conception des systèmes embarqués, le matériel et les logiciels doivent être conçus, d'autant plus que nous devons évaluer l'emplacement et l'impact des contre-mesures qui peuvent être matérielles ou logicielles [143, 313].

Il faut aussi soutenir la verification formelle de propriétés de sûreté et sécurité, notament les propriétés de sûreté de "deadlock", Accessibilité, Vivacité, et Performance, et les propriétés de sécurité de Confidentialité, Authenticité, et Disponibilité.

9.2.4 Travail Connexe

Il y avait des autres outils et méthodologies pour le modélisation de systèmes embarqués, comme [24, 104, 170, 271, 331], mais ils se concernent plus le verification fonctionelle et performance. Entre ceux qui se

concerne la sécurité, ils sont plutôt sur la analyse seulement [187, 256, 264], en modélisation de logiciel et pas matériel [168, 206], ou ils concernent seulement quelques attaques [205].

Toutes ces autres méthodologies de conception et les outils offrent les capacites de la modélisation et de la vérification avec une gamme d'approches, mais aucune d'entre elles ne supporte le processus de conception complet nécessaire à la conception de systèmes embarqués sûrs et sécurisés, tels que le véhicule autonome de Vedecom. Bien que la méthodologie SysML-Sec n'ait pas la capacité de répondre à certains des besoins, comme une meilleure analyse du temps et une meilleure modélisation en matière de sécurité, elle a suffisamment pris en compte pour que nous puissions l'adapter et l'améliorer afin de tenir compte de tous nos besoins en matière de modélisation et de vérification. Le reste de la thèse traite de la façon dont nous avons ajouté les principales capacités manquantes.

9.3 Méthodologie

La figure 9-1 présente une vue d'ensemble de notre méthodologie. Tout d'abord, nous modélisons les exigences de notre système, en ce qui concerne la fonctionnalité, la sûreté et la sécurité. Pour mieux déterminer les besoins, nous modélisons ensemble les attaques et les défauts potentiels contre qui le système devrait protéger. Les arbres d'attaque décrivent les attaques possibles auxquelles le système peut être confronté, et les arbres de défaillance décrivent comment les défaillances du système peuvent entraîner des pertes possibles. Les étapes d'attaque détaillées d'un arbre d'attaque, par exemple, Les exigences, les défaillances et les attaques doivent être modélisées ensemble, car les attaques et les échecs possibles peuvent nous amener à développer de nouvelles exigences. Les contre-mesures aux attaques et aux pannes, en particulier, peuvent être directement traduites en exigences.

Après avoir terminé notre analyse d'exigences, nous modélisons le système à différents niveaux de détail. Les modèles de partitionnement matériel/logiciel décrivent l'architecture et les modèles de haut niveau fonctionnel, et ils montrent quelle composante architecturale exécute quelle fonction : cette étape est également appelée "allocation". Les premiers modèles peuvent être abstraits et ignorer les détails et ne décrire que les propriétés de haut niveau. Une fois que le système est suffisamment modélisé, la simulation et la vérification formelle déterminent si les exigences relatives à l'allocation (en termes de sûreté, de sécurité et de performance) sont satisfaites. En fonction des résultats de la vérification, les modèles de partitionnement logiciel/matériel peuvent être modifiés et des contre-mesures de sûreté et de sécurité pertinentes peuvent être ajoutées. Les contre-mesures à ce stade comprennent le matériel redondant, les pare-feu et les opérateurs de sécurité des données abstraites. Des contre-mesures fonctionnant à un niveau inférieur d'abstraction sont ajoutées dans la phase suivante, telles que celles qui fonctionnent sur des valeurs exactes des communications de données. Les éléments de conception et les résultats de vérification peuvent être liés aux diagrammes de phase d'analyse, tels que les résultats de synchronisation liés à une exigence de synchronisation, ou la confirmation que certaines données sont sécurisées peut être utilisée pour signifier qu'une attaque est impossible. Les résultats des vérifications de sécurité déterminent également comment les scénarios d'attaque peuvent être ajoutés explicitement aux diagrammes.

Une fois l'architecture et l'allocation déterminées, nous modélisons le comportement détaillé du système pendant la phase de conception logicielle. Les modèles de conception logicielle préliminaires peuvent également être générés automatiquement à partir des modèles de partitionnement HW/SW. Dans cette étape, les algorithmes et le comportement du système sont modélisés plus en détail. Une vérification formelle vérifie ensuite à nouveau le système pour s'assurer qu'il répond à des exigences plus précises.

Dans la conception logicielle, les requêtes de vérification sont écrites en pragma, qui sont des notes sur les diagrammes de modélisation [16, 252]. La vérification de sûreté est effectuée avec UPPAAL ou le TTool Model Checker, et la vérification de sécurité est effectuée avec ProVerif [13, 15, 16]. Après vérification, chaque pragma est marqué automatiquement "vérifié", "non vérifié", ou "ne peut pas être analysé" [209, 252]. Sur la base des résultats de la vérification, les modèles du système peuvent être affinés et des contre-mesures supplémentaires, comme des protocoles de sécurité exacts, des contrôles de plausibilité, etc ajouté. Une fois que le logiciel est suffisamment conçu et vérifié, notre outil peut générer automatiquement du code C pour le prototypage et les séquences de test.

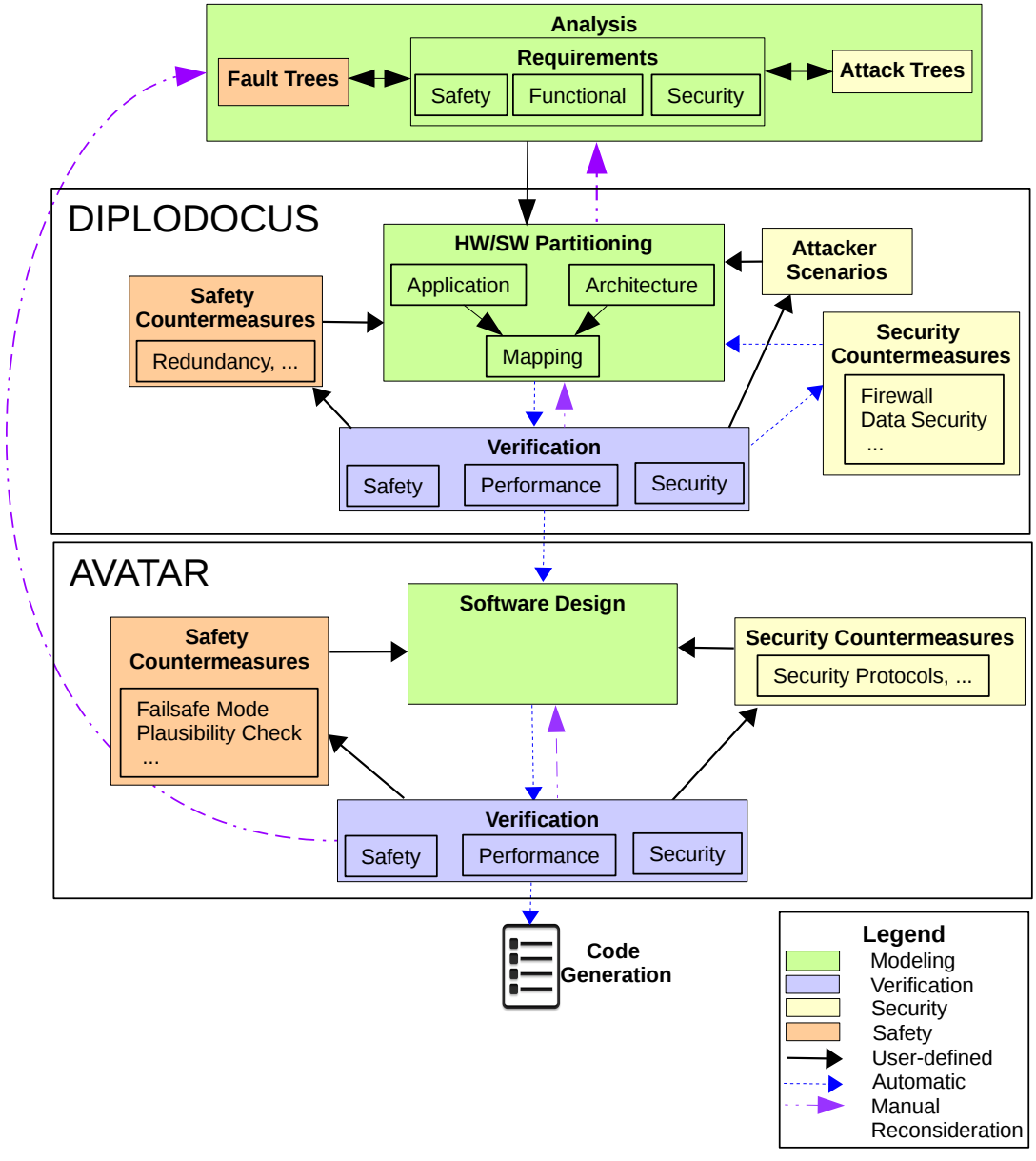


Figure 9-1: SysML-Sec Méthodologie pour la Conception de Systèmes Embarqués Sûrs et Sécurisés

9.4 Sécurité d'un Partitionnement Logiciel/Matériel

Le partitionnement matériel/logiciel, phase initiale importante dans le développement des systèmes embarqués, modélise la fonctionnalité abstraite et de haut niveau d'un système et distribue les fonctions du système entre les architectures matérielles candidates. Cette phase décide de la "meilleure" architecture en fonction de plusieurs critères, dont le coût et la performance du matériel. La spécification de l'emplacement d'exécution de chaque fonction dans l'architecture génère un modèle appelé "allocation".

Lors de la modélisation des mécanismes de sécurité après la sélection d'une allocation, les concepteurs peuvent déterminer que l'impact sur les performances dû à l'ajout de cryptage/décryptage peut rendre l'allocation sélectionnée non optimal, les forçant à refaire leur modèle et à sélectionner une autre allocation. Le partitionnement du matériel/logiciel peut également être modifié si les protocoles de sécurité sont choisis pour être déplacés afin d'être exécutés dans le matériel. Certaines des contre-mesures de sécurité qui pourraient être ajoutées, telles que les modules de sécurité matérielle et les pare-feu, sont également basées sur le matériel. Il peut donc être nécessaire de reconsidérer l'architecture.

Dans cette section, nous expliquons comment modéliser abstraitement la sécurité dans cette phase, puis comment vérifier formellement la sécurité du système. Nous discutons de la façon de modéliser à la fois la capacité de l'attaquant à cibler l'architecture et les efforts pour adapter les mécanismes de sécurité à un niveau élevé d'abstraction. Pour soutenir notre modélisation de la sécurité, nous faisons la distinction entre les emplacements architecturaux sécurisés et non sécurisés. Les bus de communication peuvent être modélisés en interne (en fonction des capacités supposées de l'attaquant) et donc protégés contre un attaquant externe. Au contraire, les bus externes peuvent être espionnés. Les fonctions allouées sur le même processeur peuvent considérer leur échange de messages comme sécurisé. L'architecture choisie affecte donc le besoin d'algorithmes de chiffrement et de stockage de matériel cryptographique. Les mécanismes de sécurité sont décrites grâce à un opérateur de comportement qui définit notamment un label pour aider à suivre les opérations de sécurité appliquées à certaines données.

9.4.1 Modèle d'Attaquant

Nous nous basons sur le même modèle Dolev-Yao Attacker que celui utilisé pour l'analyse de sécurité dans les modèles de conception de logiciels [209]. L'attaquant Dolev-Yao est un attaquant idéalisé qui a le contrôle total du réseau de communication public, car il peut lire, supprimer et envoyer tous les messages sur un canal public, et effectuer des opérations cryptographiques (cryptage, décryptage, hachage, etc.) et d'autres opérations sur les messages (splitting/joining) [55, 65, 85]. Il suppose une cryptographie parfaite, car il suppose que les messages ne peuvent pas être décryptés par la force brute, mais seulement si la clé est connue. Le modèle Dolev-Yao est dit utiliser la cryptographie 'black-box' car il ne permet pas de deviner les clés, et des modèles probabilistes ont été développés pour répondre à cette limitation [60, 317, 350].

9.4.2 Modèle de Vulnérabilités

Les hypothèses concernant les capacités de l'attaquant sont reflétées sur l'architecture. Sur la modélisation architecturale, les bus peuvent être spécifiés comme publics ou privés, correspondant au modèle Dolev-Yao, selon qu'ils sont accessibles ou non à un attaquant. Par exemple, les appareils communiquant sur un réseau WiFi seraient modélisés comme échangeant sur un bus public, tandis que le bus interne serait

modélisé comme privé. Les bus privés sont marqués *sécure* avec un bouclier vert, comme indiqué sur le diagramme d'architecture. Le bus public 'PublicBus' ne contient pas de bouclier vert. Les distinctions entre les types de bus supposent également les capacités de l'attaquant : si nous supposons qu'un attaquant n'a pas d'accès physique au système, alors nous pouvons décrire les bus internes comme étant privés, mais si un attaquant peut physiquement sonder le bus, alors il doit être indiqué comme étant public.

9.4.3 Scénarios d'attaque

Les scénarios d'attaque modélisent explicitement les interactions de l'attaquant avec un système. Ils sont basés sur les arbres d'attaque, mais contiennent plus de détails d'implémentation et peuvent être simulés en conjonction avec le modèle de système.

Un scénario d'attaquant consiste en une ou plusieurs tâches de l'attaquant, représentant des fonctions distinctes travaillant ensemble pour effectuer une attaque sur un système. Le comportement d'une tâche d'attaquant inclut toutes les actions possibles d'une tâche d'application normale (lecture/écriture de données sur un canal, opérations de contrôle, exécution de calculs, etc). En outre, lorsque les tâches normales ne peuvent lire et écrire que sur des canaux directement associés à eux-mêmes, les tâches de l'attaquant peuvent lire et écrire sur n'importe quel canal public. Un canal public est défini comme étant tout canal alloué à un chemin comprenant au moins un bus accessible à l'attaquant. De plus, les tâches de l'attaquant possèdent une capacité supplémentaire : 'Code Injection', qui remplace le comportement d'une tâche de l'application par un comportement déterminé par l'attaquant, modélisant la capacité de l'attaquant à changer le flux d'exécution d'une tâche qui peut inclure une modification du code.

Les scénarios d'attaque peuvent être simulés pour analyser leur effet sur le système. L'une des nouvelles propriétés de sécurité qui peuvent ainsi être analysées est la disponibilité. Pendant la simulation avec et sans l'attaquant, nous pouvons vérifier la charge des processeurs et le nombre de fausses commandes envoyées par rapport au nombre de commandes légitimes envoyées, pour voir si l'attaquant peut effectuer une attaque par déni de service.

9.4.4 Modèle de Contre-mesures

Lorsqu'un attaquant peut accéder à un bus public, et donc à toutes les communications qui le traversent, l'attaquant peut altérer les communications selon une méthode qui pourrait affecter négativement le comportement du système. Comme décrit précédemment, un attaquant ne devrait pas être capable d'envoyer des commandes et de prendre le contrôle d'une voiture voisine, ou de modifier les données de perception pour indiquer qu'il n'y a pas d'obstacles. La sécurisation des communications peut impliquer l'ajout d'un protocole de sécurité éprouvé. Alors que les protocoles de sécurité exacts sont modélisés dans les phases ultérieures de développement [209], dans cette phase, nous devons encore prendre en compte le temps d'exécution pour exécuter les protocoles de sécurité pour un partitionnement HW/SW précis, et trouver une méthode pour vérifier formellement que les données sont bien sécurisées. A ce niveau d'abstraction, nous ne sommes pas intéressés pendant le partitionnement par l'implémentation d'algorithmes de chiffrement: il suffit de considérer les paramètres qui affecteront le choix du partitionnement (satisfaction des propriétés de sécurité, temps d'exécution, taille des données).

Pour modéliser abstraitement les protocoles de sécurité et ne prendre en compte que ces propriétés pertinentes, nous introduisons un opérateur appelé *Cryptographic Configurations*, qui comprend une balise à

ajouter aux communications pour indiquer la présence de sécurité et le temps de traitement pour l'exécution du chiffrement/déchiffrement sur ces données

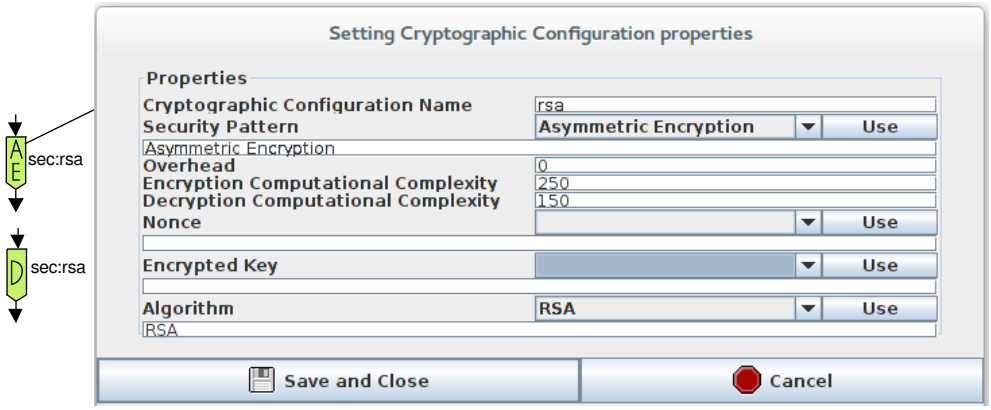


Figure 9-2: Spécification de la "Cryptographic Configuration" pour le Chiffrement et le Déchiffrement Asymétrique

Cryptographic Configurations sont des artefacts graphiques qui permettent au vérificateur de sécurité de suivre les éléments de chiffrement des données. Dans les diagrammes d'activité, ils apparaissent comme un pentagone à l'envers. marqués avec leur type, comme indiqué dans la Figure 9-2, où 'AE' représente le cryptage asymétrique et 'D' représente le décryptage. Les configurations cryptographiques peuvent être tapées comme suit: *Cryptage symétrique* et *Cryptage asymétrique*. Les modèles cryptent les données ainsi qu'une ou plusieurs clés spécifiques au modèle. Un *MAC* peut être ajouté aux messages pour l'authentifier et déterminer s'il a été modifié. *Hash* calcule un hachage des données (et le vérifie ensuite). *Nonces* peut être concaténée à des messages avant le cryptage pour vérifier l'authenticité. *Advanced* permet à l'utilisateur de indiquent leur propre schéma de cryptage, comme des combinaisons de données cryptographiques. opérations.

Figure 9-2 (à droite) montre la spécification d'une configuration cryptographique. Le concepteur peut choisir le type et ensuite la performance correspondante, ou sélectionner un algorithme pré-construit qui estimera automatiquement les paramètres de performance sur la base des résultats expérimentaux dans [73]. Ces paramètres estimés devraient être modifiés pour mieux refléter le CPU utilisé, mais ils peuvent toujours être utilisés pour comparer les performances de différents algorithmes. Par exemple, les opérations de chiffrement peuvent être caractérisées par une complexité de calcul de chiffrement et de déchiffrement (une mesure des cycles d'exécution relatifs selon le processeur) et des coût supplémentaires (bits supplémentaires ajoutés au message pendant le chiffrement). Ces paramètres nous permettent de modéliser l'impact des mécanismes de sécurité sur la performance lors de l'évaluation d'une allocation.

Les protocoles de sécurité peuvent ajouter des surcouts. Comme mentionné précédemment dans la liste des contre-mesures possibles, les modules de sécurité matérielle (HSM) sont un élément matériel spécialement conçu pour effectuer les opérations cryptographiques plus efficacement que les processeurs ordinaires. Nous les modélisont en tant qu'accélérateur matériel qui effectue le chiffrement et le déchiffrement en moins de cycles.

Les pare-feu peuvent être utilisés pour filtrer les communications. Ils peuvent être modélisés comme une tâche qui transmet ou supprime les communications en fonction des règles de pare-feu en vigueur. Les règles peuvent être modifiées dynamiquement via des signaux, pour représenter comment le pare-feu peut

avoir besoin de bloquer une communication qui a été détectée comme anormale. Si le pare-feu reçoit une communication qui devrait être bloquée, il ne transmet pas la communication au destinataire prévu.

9.4.5 Vérification Formelle

Ensuite, un processus de vérification de la sécurité devrait être utilisé pour confirmer que les contre-mesures ajoutées sont suffisantes. Les propriétés de sécurité que nous devons vérifier dans les systèmes embarqués sont: vérifier si des données importantes seront gardées secrètes pour un attaquant (Confidentialité), et si des données importantes ne peuvent pas être falsifiées par un attaquant (Authenticité). De plus, pour vérifier que les protocoles sont capables de s'exécuter correctement par exemple de déchiffrer le message reçu, il faut vérifier l'accessibilité des états. On utilise le langage de spécification formel **ProVerif**. Notre outil traduit automatiquement les modèles d'allocation en AVATAR, et ensuite en ProVerif [201, 209].

ProVerif est basé sur le modèle d'attaquant un Dolev-Yao, qui est un modèle d'attaquant dans lequel n'importe qui peut lire ou écrire sur n'importe quel bus public, créer de nouveaux messages ou appliquer des primitives connues. En d'autres termes, une fois que l'attaquant a intercepté un message, il se comporte comme une personne raisonnablement compétente avec une connaissance de la cryptographie de base, et peut récupérer le message décrypté s'il peut aussi intercepter une clé. L'attaquant peut acquérir des connaissances à chaque fois qu'un processus effectue une écriture sur un canal public, puis à partir de tous les calculs possibles. Par exemple, si l'attaquant connaît la clé, puis intercepte un message chiffré, l'attaquant connaîtra alors le message original. Cela correspond à nos hypothèses d'un attaquant actif sur le système, qui peut effectuer des opérations cryptographiques et tentera activement d'injecter des messages falsifiés ou de déchiffrer les messages récupérés.

Les résultats de sécurité retournés par le prouveur ProVerif indiquent si chaque requête est vérifiée vraie, vérifiée fausse, ou ne peut pas être vérifiée. Ces résultats sont ajoutés sur les diagrammes de composants DIPLODOCUS pour la commodité de l'utilisateur, et restent sur les diagrammes pour que l'utilisateur puisse s'y référer lorsqu'il reconsidère et modifie les modèles. Cette capacité a d'abord été ajoutée comme décrit [209].

Les annotations de sécurité, sous forme de cadenas gris, sont colorées pour indiquer si leur propriété de sécurité correspondante est satisfaite (vert) ou non satisfaite (rouge). S'il y a une erreur avec le prouveur et que les résultats ne peuvent être déterminés, le cadenas reste gris.

Pour les résultats de confidentialité, les données échangées sur un canal vulnérable à un attaquant sont indiquées par un cadenas rouge barré. Cependant, les données prouvées confidentielles sont indiquées par un cadenas vert. L'authenticité faible et l'authenticité forte sont indiquées par un cadenas séparé en deux demi-blocs colorés en vert ou rouge selon le résultat de vérification.

Une altération des communications internes par un attaquant pourrait empêcher le système de freinage de fonctionner correctement. Nous démontrons comment fonctionne la vérification ProVerif en vérifiant la sécurité de notre système de véhicule autonome, en mettant l'accent sur l'authenticité des communications entre les unités de navigation et de perception. Dans l'architecture actuelle, montrée dans la Figure 9-3, les unités Perception et Navigation communiquent à travers un bus non sécurisé. Comme prévu, les données de perception ne sont pas vérifiées authentiques.

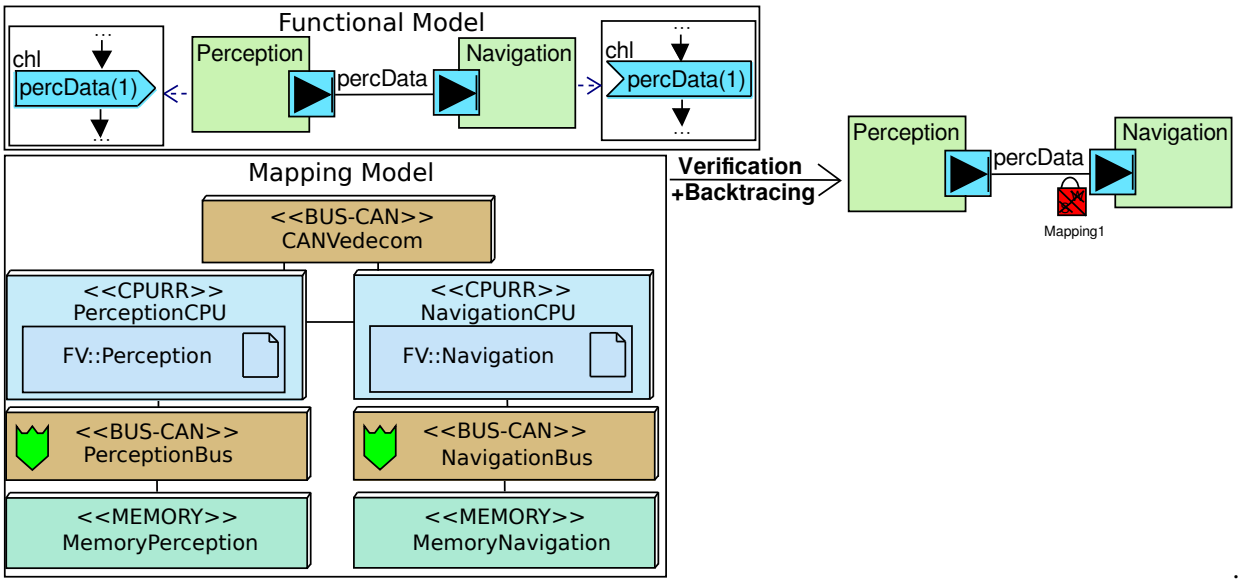


Figure 9-3: Résultats de vérification pour l'allocation par défaut

.

9.4.6 Génération Automatique de Contre-mesures

Comme indiqué précédemment, les ingénieurs logiciels ne sont souvent pas des experts en sécurité. Dans la phase de partitionnement HW/SW, nous offrons donc des capacités de génération automatique, y compris l'ajout de configurations cryptographiques avec les opérateurs de sécurité associés et les modules de sécurité matérielle. A chaque génération, notre outil peut ajouter des opérateurs de sécurité, ajouter un HSM à des tâches sélectionnées, ou mapper toutes les clés sur une mémoire accessible en toute sécurité.

Les représentations de sécurité sont ajoutées en fonction des propriétés de sécurité qui doivent être remplies pour chaque canal. Cette sécurité générée automatiquement ne vise qu'à fournir une estimation des propriétés de performance du système sécurisé, et les protocoles de sécurité exacts devraient être décrits plus en détail dans la phase de conception du logiciel. Dans certains cas, il existe plusieurs méthodes pour satisfaire une propriété de sécurité, et nous ne prenons qu'une seule de ces options pour simplifier l'algorithme. Par exemple, par défaut, tous les opérateurs de cryptage utilisent le cryptage symétrique, pour simplifier l'allocation des clés.

9.5 Évaluation des Performances

Les mesures de performance sont couramment utilisées pour évaluer la pertinence d'une architecture et d'une allocation, comme dans le cas d'une exploration automatique [226, 257]. La performance d'un système se caractérise par les temps de traitement d'une donnéee ou la charge des composants de l'architecture, comme la charge des bus et des processeurs.

De nombreux systèmes critique, tels que les voitures autonomes, interagissent en permanence avec l'environnement et les utilisateurs. Les nouvelles entrées du monde extérieur sont traitées par le système, qui effectue ensuite une réponse observable dans le monde réel. Le moment de où ces interventions sont produites peut

avoir un impact considérable sur la fonctionnalité et la sécurité.

Il est donc important d'examiner la performance d'un système : à la fois les latences et la charge des processeurs. Les caractéristiques de synchronisation d'un système peuvent également être affectées par l'utilisation, ou la charge, sur les processeurs ou les bus de communication [99]. Si un processeur est trop chargé, il peut retarder des tâches moins prioritaires.

9.5.1 Mesure des Temps de Latence

Une fois les exigences de synchronisation définies, les opérateurs concernés doivent être sélectionnés sur les diagrammes de modélisation. Chaque opérateur concerné est étiqueté comme un "point de contrôle de latence", affiché comme un drapeau bleu.

Les latences sont mesurées en faisant la moyenne des valeurs mesurées sur plusieurs simulations. Le moteur de simulation, basé en C+++, est décrit dans [181, 182]. Chaque composant architectural planifie ensuite les opérations par les tâches ou les communications qui y sont mappées, qui s'exécutent ensuite comme une transaction sur une période de temps donnée. Chaque simulation est stockée sous la forme d'un ensemble de transactions exécutées sur l'ensemble des composants architecturaux.

Les latences sont ensuite retracées pour être affichées pour chaque opérateur. Pour chaque mesure de latence, le nom de l'autre opérateur est affiché, ainsi que la valeur moyenne de la mesure de latence en secondes.

9.5.2 Analyse de Système Sûr et Sécurisé

Tout au long de cette thèse, nous avons démontré chacune des nouvelles capacités de notre approche pour améliorer la sécurité et la sûreté du Véhicule Autonome dans notre exemple. Nous avons expliqué certaines des contre-mesures possibles à ajouter, puis nous concluons en intégrant un ensemble préliminaire de mécanismes que nous considérons nécessaires pour éviter la plupart des dangers possibles.

Nous concluons en intégrant toutes les contre-mesures de sûreté et de sécurité pertinentes dans un modèle unique, puis en évaluant sa sûreté, sa sécurité et son rendement globaux. Ce modèle, dont la sûreté et la sécurité sont améliorées, devrait protéger contre bon nombre des défaillances et des attaques, telles que les limitations des capteurs [180, 216, 229, 295], les capteurs manipulés [255, 294, 346], et les commandes et données injectées par l'attaquant [67, 67, 223, 337].

Nous vérifions maintenant l'impact des ajouts sur les performances. Nous examinons à la fois la latence de freinage et la charge sur le(s) processeur(s) de perception et de navigation comme avant. Le tableau 9.2 compare les résultats de performance de notre modèle de base (sans contre-mesures de sécurité ou de sûreté) et notre nouveau modèle final sûr et sécurisé. Comme prévu, la latence de freinage a été considérablement augmentée, bien que les charges sur les processeurs aient diminué, probablement en raison de l'interruption des tâches de perception et du temps d'attente en attente d'une autre tâche pour envoyer ou recevoir des communications. A ce stade, le temps de réaction au freinage d'un obstacle est en moyenne inférieur aux 1,08 secondes requises, la latence de freinage maximale dépasse le temps de réaction admissible. Pour améliorer cette latence, nous pourrions envisager de modifier davantage l'allocation, par exemple en utilisant de meilleurs processeurs ou en améliorant le temps d'exécution des différents algorithmes. Dans ce cas, alors que cette première série de contre-mesures de sécurité et de sûreté devrait

Table 9.2: Résultats de Performance

Allocation	Latence de freinage (s)			Charge (%)			
	Max	Moyenne	Écart-type	Perc CPU	Perc CPU1	Perc CPU2	Nav CPU
Allocation Défaut	0.52	0.19	0.13	53			39
Allocation Sûre et Sécurisée	1.13	0.46	0.10	4	10	10	42
Allocation Sûre et Sécurisée (Performance Amélioré)	0.99	0.39	0.15	5	7	7	36

améliorer la sécurité de notre véhicule autonome, leur ajout a dégradé ses performances jusqu'à un point où le système ne pouvait plus être garanti sûr.

Par conséquent, nous devons modifier notre système pour qu'il réponde aux exigences de performance. Nous pourrions réduire la latence de freinage en utilisant différentes approches, telles l'utilisation de processeurs plus efficients, des protocoles de sécurité plus efficaces, des algorithmes optimisés, etc. Nous avons amélioré les algorithmes des tâches de Perception car ces sont les plus consommateurs en terme de complexité de calcul. Le tableau 9.2 présente nos résultats du point de vue des performances. Nous voyons que le système peut satisfaire aux exigences de temps si la complexité de calcul est réduite de 10 %. Cette hypothèse nécessite cependant d'être validée à nouveau après la phase de re-conception logiciel. Ce processus est conforme à notre méthodologie; les étapes de modélisation, de vérification et de révision seront répétées jusqu'à ce que toutes les exigences du système soient satisfaites.

9.6 Conclusion

Les systèmes embarqués connectés, tels que les futurs véhicules autonomes, devraient apporter de nombreuses commodités dans nos vies. Non seulement la connectivité Internet apporte du divertissement aux passagers lors d'un long voyage, mais elle nous donne accès à la richesse des connaissances disponibles : un emplacement exact avec des instructions de conduite complètes, l'emplacement des restaurants ou des stations-service pour les aires de repos, et les conditions de circulation tout autour de nous pour mieux nous aider à éviter les embouteillages. Les conducteurs n'ont plus besoin de lire une carte papier ou de trouver des détours à la main. Des dispositifs de sécurité supplémentaires, comme le freinage automatique ou l'aide au stationnement, peuvent également prévenir les collisions.

Cependant, en adoptant ces fonctions connectées et automatisées, nous abandonnons une partie du contrôle de notre véhicule et nous nous appuyons davantage sur le logiciel. Les accidents dans un véhicule complètement autonome seront dus à des erreurs du système au lieu d'erreurs du conducteur. Par conséquent, ces systèmes devraient être assurés de fonctionner correctement et en toute sécurité. Comme l'ont montré les accidents impliquant un pilote automatique, ces systèmes ne peuvent pas encore gérer toutes les conditions routières, et les capteurs et les algorithmes de traitement peuvent ne pas percevoir correctement l'environnement correctement [180, 300, 323]. La sûreté de ces systèmes devrait être assurée par un processus de vérification complet.

La sûreté des systèmes embarqués repose sur de multiples aspects. Premièrement, le système devrait être

protégé contre les pirates: il faut que les pirates ne puissent pas modifier le code du système, comme dans le hack [223] de Miller et Valasek, où des fausses ommandes sont injectiées dans le système. Par exemple, un pirate informatique qui peut prendre le contrôle du véhicule d'une autre personne pourrait provoquer un accident mortel. Un système sûr doit aussi tenir compte de ses limites et les compenser. Comme les caméras ne fonctionnent pas bien dans des conditions de faible éclairage, le système devrait s'appuyer principalement sur les autres capteurs la nuit. Enfin, la sûreté du système dépend de la performance ou du timing. Les composants matériels ne doivent pas être chargés à un point tel qu'ils ne peuvent pas exécuter des fonctions de manière fiable, et les événements critiques, tels que l'évitement d'obstacles, ne doivent pas être retardés de manière à ce qu'ils s'exécutent au-delà d'un délai strict.

9.6.1 Contributions

Cette thèse portait sur la conception sûre et sécurisée de tels systèmes en s'appuyant sur l'ingénierie pilotée par les modèles. La modélisation d'un système aide les concepteurs à considérer systématiquement tous les besoins du système et à détecter les problèmes dans les phases de modélisation avant l'achat de matériel ou le développement de code qui prend beaucoup de temps. Réparer les problèmes dès le début de ces phases permet d'éviter des retouches ou des correctifs coûteux après la production de masse et la distribution.

Nous avons commencé par un résumé des problèmes auxquels sont confrontés les véhicules autonomes et connectés, et nous avons discuté de la façon dont la modélisation et la vérification pourraient soutenir la conception de tels systèmes. Nous avons présenté les capacités d'autres méthodologies et d'outils, mais comme aucune ne répondait à tous nos besoins de conception, nous avons proposé une nouvelle méthodologie. Notre nouvelle méthodologie améliore la méthodologie SysML-Sec [13] pour prendre en compte la sécurité pendant la phase de partitionnement HW/SW, permettant de modéliser les capacités de l'attaquant pour cibler une architecture et des contre-mesures basées sur l'architecture, et d'améliorer l'analyse des performances en examinant les latences entre les événements critiques. Ces capacités améliorées de modélisation et de vérification aident à sélectionner une cartographie répondant aux exigences de sûreté, de sécurité et de performance, et fournissent une meilleure base pour la phase suivante de conception logicielle.

9.6.2 Perspectives

Le but ultime de cette thèse était de déterminer comment concevoir des systèmes sûrs et sécurisés. Nous avons fourni une méthodologie et un outil support qui devrait être facile à utiliser, mais qui ne peut toujours pas aborder tous les aspects de la conception. Bien que cette thèse ait proposé de nouveaux aspects pour la conception des systèmes embarqués, il existe des travaux futurs importants, au-delà de la modélisation et de la vérification. Une importante orientation de recherche future est d'étudier les aspects pratiques de la sécurité dans les systèmes embarqués, afin de déterminer si notre modélisation représente adéquatement les systèmes et leurs vulnérabilités potentielles.

Ainsi, nous avons proposé une contribution, comme sur les expériences pratiques pour voir si nos postulats et théories sont suffisamment réalistes, et les améliorations de notre outil pour une meilleure expérience utilisateur. On doit enrichir notre modèle d'attaquant, et prendre en compte les probabilities d'attaque. Idéalement, notre outil pourrait aussi prendre l'ensemble de toutes les exigences (sûreté, sécurité et performance), puis générer automatiquement un modèle entièrement sûr et sécurisé.

Bibliography

[1] Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model, Version 3.1, Revision 1 (CCMB-2006-09-001), Sep 2006. http://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R1.pdf.

[2] Pekka Abrahamsson. Speeding up embedded software development. *ITEA Innovation report*, 2007.

[3] Alan Grau. The internet of things needs firewalls too. http://www.electronicdesign.com/communications/internet-things-needs-firewalls-too, Mar 2013.

[4] Azadeh Alebrahim and Maritta Heisel. Applying performance patterns for requirements analysis. In *Proceedings of the 20th European Conference on Pattern Languages of Programs*, EuroPLoP '15, pages 35:1–35:15, New York, NY, USA, 2015. ACM.

[5] Yomna Ali, Sherif El-Kassas, and Mohy Mahmoud. A rigorous methodology for security architecture modeling and verification. In *Proceedings of the 42nd Hawaii International Conference on System Sciences*, volume 978-0-7695-3450-3/09. IEEE, 2009.

[6] Dave Altavilla. When the IOT fails: Nest recalls over 400k smoke detectors. https://www.forbes.com/sites/davealtavilla/2014/05/22/when-the-iot-fails-nest-recalls-over-400k-smoke-detectors/, May 2014.

[7] M. Althoff, O. Stursberg, and M. Buss. Model-based probabilistic collision detection in autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 10(2):299–310, Jun 2009.

[8] Matthias Althoff, Olaf Stursberg, and Martin Buss. Safety assessment of autonomous cars using verification techniques. In *American Control Conference, 2007. ACC'07*, pages 4154–4159. IEEE, 2007.

[9] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425, Jun 1990.

[10] ANSYS. Medini analyze. http://www.medini.eu/index.php/en/products/functional-safety, 2017.

[11] Axelle Apvrille. Geek usages for your Fitbit Flex tracker Hack.lu, Luxemburg, October 2015. Slides at framadrive.org/index.php/s/Wk6nxAKMpVTdQl4, October 2015.

[12] L. Apvrille, L. Li, and Y. Roudier. Model-driven engineering for designing safe and secure embedded systems. In *2016 Architecture-Centric Virtual Integration (ACVI)*, pages 4–7, April 2016.

[13] L. Apvrille and Y. Roudier. SysML-Sec: A Model Driven Approach for Designing Safe and Secure Systems. In *3rd International Conference on Model-Driven Engineering and Software Development, Special session on Security and Privacy in Model Based Engineering*, France, February 2015. SCITEPRESS Digital Library.

[14] Ludovic Apvrille. If I secure my car, will it still brake? http://archive.hack.lu/2014/hacklu_cars.pdf, October 2014.

[15] Ludovic Apvrille. Webpage of TTool. In *http://ttool.telecom-paristech.fr/*, 2015.

[16] Ludovic Apvrille and Letitia W. Li. 9 - Safe and Secure Support for Public Safety Networks. In *Wireless Public Safety Networks 3*, pages 185 – 210. Elsevier, 2017.

[17] Ludovic Apvrille and Yves Roudier. SysML-Sec Attack Graphs: Compact Representations for Complex Attacks. In Sjouke Mauw, Barbara Kordy, and Sushil Jajodia, editors, *Graphical Models for Security*, pages 35–49, Cham, 2016. Springer International Publishing.

[18] Ashraf Armoush, Falk Salewski, and Stefan Kowalewski. Effective pattern representation for safety critical embedded systems. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 4, pages 91–97. IEEE, 2008.

[19] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *Design, Automation and Test in Europe*, pages 178–183 Vol. 1, March 2005.

[20] Warwick Ashford. Developers lack skills needed for secure devops, survey shows. http://www.computerweekly.com/news/450424614/Developers-lack-skills-needed-for-secure-DevOps-survey-shows, August 2017.

[21] Colin Atkinson and Thomas Kühne. Profiles in a strict metamodeling framework. *Science of Computer Programming*, 44(1):5–22, 2002.

[22] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[23] Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *Computer Security Foundations Symposium, 2008. CSF'08. IEEE 21st*, pages 195–209. IEEE, 2008.

[24] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36(4):45–52, April 2003.

[25] Michael Barr and Anthony Massa. *Programming embedded systems: with C and GNU development tools*. " O'Reilly Media, Inc.", 2006.

[26] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. 1987.

[27] Nicholas Becker. Safety of the intended functionality for ADAS. http://files.hanser-tagungen.de/docs/20140905155628_SafetyoftheintendedFunctionalityfor20ADAS.pdf.

[28] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[29] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, pages 87–124. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, 2004.

[30] Better buys. Estimating password-cracking times. https://www.betterbuys.com/estimating-password-cracking-times/.

[31] John Birch. Safety argument framework for vehicle autonomy. http://safety.addalot.se/upload/2017/2-6-2%20JohnBirch.pdf, May 2017.

[32] Norbert Bißmeyer, Joël Njeukam, Jonathan Petit, and Kpatcha M Bayarou. Central misbehavior evaluation for VANETS based on mobility data plausibility. In *Proceedings of the ninth ACM international workshop on Vehicular inter-networking, systems, and applications*, pages 73–82. ACM, 2012.

[33] Bob Blakeley. *Introduction to Security Design Patterns*. Open Group, 2004.

[34] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, July 2009.

[35] B. Blanchet. Proverif automatic cryptographic protocol verifier user manual. Technical report, CNRS, Département d'Informatique École Normale Supérieure, Paris, July 2010.

[36] Matt Blaze, Whitfield Diffie, Ronald L Rivest, Bruce Schneier, and Tsutomu Shimomura. Minimal key lengths for symmetric ciphers to provide adequate commercial security. a report by an ad hoc group of cryptographers and computer scientists. Technical report, Information Assurance Technology Analysis Center Falls Church VA, 1996.

[37] Hans Blom, De-Jiu Chen, Henrik Kaijser, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Ramin Tavakoli Kolagari, and Sara Tucci. EAST-ADL: An Architecture Description Language for Automotive Software-intensive Systems in the Light of Recent use and Research. *International Journal of System Dynamics Applications (IJSDA)*, 5(3):1–20, 2016.

[38] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[39] Mathilde Boisson, Emmanuel Arbaretier, and Annie Bracquemond. Architecture sûre du véhicule autonome sans chauffeur. In *Congrès Lambda Mu 20*, 2016.

[40] Connor Bolton, Sara Rampazzi, Chaohao Li, Andrew Kwong, Wenyuan Xu, and Kevin Fu. Blue Note: How intentional acoustic interference damages availability and integrity in hard disk drives and operating systems. In *Proceedings of the 39th Annual IEEE Symposium on Security and Privacy*, May 2018.

[41] Jonathan Peter Bowen. *Formal specification and documentation using Z: A case study approach*, volume 66. International Thomson Computer Press London, 1996.

[42] Lisane Brisolara, Leandro Becker, Luigi Carro, Flávio Wagner, Carlos E Pereira, and Ricardo Reis. Comparing high-level modeling approaches for embedded system design. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 986–989. ACM, 2005.

[43] Great Britain, Air Accidents Investigation Branch, and G Britain. *Report on the Accident to Boeing 737-400 G-OBME Near Kegworth, Leicestershire, on 8 January 1989*. HM Stationery Office, 1990.

[44] Rodney Brooks. The big problem with self-driving cars is people. https://spectrum.ieee.org/transportation/self-driving/the-big-problem-with-selfdriving-cars-is-people, July 2017.

[45] Rodney Brooks. Symantec anomaly detection for automotive. https://www.symantec.com/products/anomaly-detection-for-automotive, 2018.

[46] RR Brooks, S Sander, Juan Deng, and Joachim Taiber. Automobile security concerns. *IEEE Vehicular Technology Magazine*, 4(2), 2009.

[47] Frederick P Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.

[48] Julien Brunel and David Chemouil. Safety and security assessment of behavioral properties using alloy. In Floor Koornneef and Coen van Gulijk, editors, *Computer Safety, Reliability, and Security*, pages 251–263, Cham, 2015. Springer International Publishing.

[49] Julien Brunel, David Chemouil, Laurent Rioux, Mohamed Bakkali, and Frédérique Vallée. A viewpoint-based approach for formal safety & security assessment of system architectures. In *11th Workshop on Model-Driven Engineering, Verification and Validation*, volume 1235, pages 39–48, 2014.

[50] Simon Burton, Jürgen Likkei, Priyamvadha Vembar, and Marko Wolf. Automotive functional safety = safety + security. In *Proceedings of the First International Conference on Security of Internet of Things*, SecurIT '12, pages 150–159, New York, NY, USA, 2012. ACM.

[51] Giorgio Calandriello, Panos Papadimitratos, Jean-Pierre Hubaux, and Antonio Lioy. Efficient and Robust Pseudonymous Authentication in VANET. In *Proceedings of the Fourth ACM International Workshop on Vehicular Ad Hoc Networks*, VANET '07, pages 19–28, New York, NY, USA, 2007. ACM.

[52] Carl R. Nave. Stopping Distance for Auto. http://hyperphysics.phy-astr.gsu.edu/hbase/crstp.html.

[53] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security Symposium*, pages 385–399, 2014.

[54] Mumin Cebe, Enes Erdin, Kemal Akkaya, Hidayet Aksu, and Selcuk Uluagac. Block4forensic: An integrated lightweight blockchain framework for forensics applications of connected vehicles. *arXiv preprint arXiv:1802.00561*, 2018.

[55] Iliano Cervesato. The Dolev-Yao intruder is the most powerful attacker. In *16th Annual Symposium on Logic in Computer Science—LICS*, volume 1, 2001.

[56] Jed Kao-Tung Chang, Chen Liu, and Jean-Luc Gaudiot. Hardware acceleration for cryptography algorithms by hotspot detection. In *International Conference on Grid and Pervasive Computing*, pages 472–481. Springer, 2013.

[57] Benjie Chen and Robert Morris. Certifying program execution with secure processors. In *HotOS*, pages 133–138, 2003.

[58] D. Chen, R. Johansson, H. Lönn, H. Blom, M. Walker, Y. Papadopoulos, S. Torchiaro, F. Tagliabo, and A. Sandberg. Integrated safety and architecture modeling for automotive embedded systems. *e & i Elektrotechnik und Informationstechnik*, 128(6):196–202, Jun 2011.

[59] Sébastien Christiaens, Juergen Ogrzewalla, and Stefan Pischinger. Functional safety for hybrid and electric vehicles. Technical report, SAE Technical Paper, 2012.

[60] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science*, 59(3):238–251, 2002.

[61] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[62] Pat L Clemens. Fault tree analysis. *JE Jacobs Severdurup*, 2002.

[63] Simon Cogliani, Diana-Ştefania Maimuţ, David Naccache, Rodrigo Portella do Canto, Reza Reyhanitabar, Serge Vaudenay, and Damian Vizár. OMD: a compression function mode of operation for authenticated encryption. In *International Workshop on Selected Areas in Cryptography*, pages 112–128. Springer, 2014.

[64] Matjaž Colnarič, Domen Verber, and Wolfgang A Halang. Real-time characteristics and safety of embedded systems. *Distributed Embedded Control Systems: Improving Dependability with Coherent Design*, pages 3–28, 2008.

[65] Hubert Comon and Vitaly Shmatikov. Is it possible to decide whether a cryptographic protocol is secure or not? *Journal of Telecommunications and Information Technology*, pages 5–15, 2002.

[66] Jamie Condliffe. The reason we won't have autonomous cars any time soon. https://gizmodo.com/how-to-teach-an-autonomous-car-to-drive-1694725874, May 2015.

[67] Lucian Constantin. Researchers hack Tesla Model S with remote attack. http://www.pcworld.com/article/ 3121999/security/researchers-demonstrate-remote-attack-against-tesla-model-s.html, September 2016.

[68] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, New York, NY, USA, 2015. ACM.

[69] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[70] Pascal Cotret, Guy Gogniat, and Martha Johanna Sepúlveda Flórez. Protection of heterogeneous architectures on FPGAs: An approach based on hardware firewalls. *Microprocessors and Microsystems*, 42:127 – 141, 2016.

[71] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with stackguard. In *Linux Expo*, 1999.

[72] Scott A Crosby and Dan S Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium*, pages 29–44, 2003.

[73] Crypto++. Crypto++ 6.0.0 benchmarks. https://www.cryptopp.com/benchmarks.html, December 2017.

[74] Crypto++. Advanced encryption standard. https://www.cryptopp.com/wiki/ Advanced_Encryption_Standard, Apr 2018.

[75] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. *11 The EAST-ADL Architecture Description Language for Automotive Embedded Software*, pages 297–307. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[76] Dark Reading Staff. Most companies suffer reputation damage after security incidents. https://www.darkreading.com/cloud/most-companies-suffer-reputation-damage-after-security-incidents/d/d-id/1330869?, January 2018.

[77] Tom Davenport and John Lucker. Running on data: Activity trackers and the internet of things. https://dupress.deloitte.com/dup-us-en/deloitte-review/issue-16/internet-of-things-wearable-technology.html, January 2015.

[78] Jerald Dawkins and John Hale. A systematic approach to multi-stage network attack analysis. In *Information Assurance Workshop, 2004. Proceedings. Second IEEE International*, pages 48–56. IEEE, 2004.

[79] R. Debouk, B. Czerny, J. d'Ambrosio, and J.J. Joyce. Safety strategy for autonomous systems. In *International Systems Safety Conference. System Safety Society*, volume 3, 2011.

[80] Lenny Delligatti. *SysML distilled: A brief guide to the systems modeling language*. Addison-Wesley, 2013.

[81] Steve Denning. Agile: The world's most popular innovation engine. https://www.forbes.com/sites/stevedenning/2015/07/23/the-worlds-most-popular-innovation-engine/, July 2015.

[82] Mamadou H Diallo, Jose Romero-Mariona, Susan Elliott Sim, Thomas A Alspaugh, and Debra J Richardson. A comparative evaluation of three approaches to specifying security requirements. In *12th Working Conference on Requirements Engineering: Foundation for Software Quality, Luxembourg*, 2006.

[83] Steven X Ding. *Model-based fault diagnosis techniques: design schemes, algorithms, and tools*. Springer Science & Business Media, 2008.

[84] Defense System Software Development. Standard, 1983.

[85] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE trans. on Information Theory*, 29:198–208, 1983.

[86] Derrick Dominic, Sumeet Chhawri, Ryan M. Eustice, Di Ma, and André Weimerskirch. Risk assessment for cooperative automated driving. In *Proceedings of the 2Nd ACM Workshop on Cyber-Physical Systems Security and Privacy*, CPS-SPC '16, pages 47–58, New York, NY, USA, 2016. ACM.

[87] Bruce Powel Douglass. *Real-time design patterns: robust scalable architecture for real-time systems*, volume 1. Addison-Wesley Professional, 2003.

[88] Terry Dunlap. The 5 worst examples of IoT hacking and vulnerabilities in recorded history. https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities/, May 2017.

[89] M. Eby, J. Werner, G. Karsai, and A. Ledeczi. Integrating security modeling into embedded system design. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, pages 221–228, March 2007.

[90] Economist. The long, winding road for driverless cars. https://www.economist.com/news/science-and-technology/21722628-forget-hype-about-autonomous-vehicles-being-around-cornerreal-driverless-cars-will, May 2017.

[91] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, Mar 1997.

[92] Fabian Eisele. Introducing Hardware Security Modules to Embedded Systems. https://vector.com/portal/medien/cmc/events/Vector_EMOB_2017_Phanuel_Hieber.pdf, 2017.

[93] M. Ekstedt, P. Johnson, R. Lagerström, D. Gorton, J. Nydrén, and K. Shahzad. Securi CAD by Foreseeti: A CAD Tool for Enterprise Cyber Security Management. In *2015 IEEE 19th International Enterprise Distributed Object Computing Workshop*, pages 152–155, Sept 2015.

[94] Andrea Enrici, Letitia Li, Ludovic Apvrille, and Dominique Blouin. A Tutorial on TTool/DIPLODOCUS: an Open-source Toolkit for the Design of Data-flow Embedded Systems. https://ttool.telecom-paristech.fr/docs/Tutorial.pdf, 2018.

[95] Cagkan Erbas, Selin Cerav-Erbas, and Andy D. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation*, 10(3):358–374, 2006.

[96] Ericsson. Ericsson mobility report: On the pulse of the netwroked society. https://www.ericsson.com/assets/local/mobility-report/documents/2016/Ericsson-mobility-report-june-2016.pdf, June 2016.

[97] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, New York, NY, USA, 2015. ACM.

[98] S. Evans and J. Wallner. Risk-based security engineering through the eyes of the adversary. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pages 158–165, June 2005.

[99] Event Helix. Issues in Real-time System Design. https://www.eventhelix.com/ RealtimeMantra IssuesInRealtimeSystemDesign.htm, 2017.

[100] EVITA. E-safety Vehicle InTrusion protected Applications. http://www.evita-project.org/.

[101] Daniel J. Fagnant and Kara Kockelman. Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice*, 77:167 – 181, 2015.

[102] T. Farkas, E. Meiseki, C. Neumann, K. Okano, A. Hinnerichs, and S. Kamiya. Integration of UML with Simulink into embedded software engineering. In *2009 ICCAS-SICE*, pages 474–479, Aug 2009.

[103] Marouane Fazouane, Henning Kopp, Rens W van der Heijden, Daniel Le Métayer, and Frank Kargl. Formal verification of privacy properties in electric vehicle charging. In *International Symposium on Engineering Secure Software and Systems*, pages 17–33. Springer, 2015.

[104] Peter H. Feiler, Bruce A. Lewis, Steve Vestal, and Edward Colbert. An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In Pierre Dissaux, Mamoun Filali-Amine, Pierre Michel, and François Vernadat, editors, *IFIP-WADL*, volume 176 of *IFIP*, pages 3–15. Springer, 2004.

[105] David C. Feldmeier and Philip R. Karn. Unix password security - ten years later. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 44–63, New York, NY, 1990. Springer New York.

[106] Ana M Fernández-Sáez, Michel RV Chaudron, and Marcela Genero. Exploring Costs and Benefits of Using UML on Maintenance: Preliminary Findings of a Case Study in a Large IT Department. In *EESSMOD@ MoDELS*, pages 33–42, 2013.

[107] Andreas Fiessler, Sven Hager, Björn Scheuermann, and Andrew W Moore. HyPaFilter-A versatile hybrid FPGA packet filter. In *Architectures for Networking and Communications Systems (ANCS), 2016 ACM/IEEE Symposium on*, pages 25–36. IEEE, 2016.

[108] Jason Fossen. How long to crack a password spreadsheet. https://cyber-defense.sans.org/blog/2009/06/12/how-long-to-crack-a-password-spreadsheet, June 2009.

[109] Ian D Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. Fast and vulnerable: A story of telematic failures. In *WOOT*, 2015.

[110] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[111] Bernhard Friedrich. The effect of autonomous vehicles on traffic. In *Autonomous Driving*, pages 317–334. Springer, 2016.

[112] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. An introduction to UML profiles. *UML and Model Engineering*, 2, 2004.

[113] Daniel D Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media, 2009.

[114] Wilbert O Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. John Wiley & Sons, 2007.

[115] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embedded Comput. Syst*, 10(4):39, 2011.

[116] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 295–306, Feb 2003.

[117] Mengmeng Ge, Jin B Hong, Walter Guttmann, and Dong Seong Kim. A framework for automating security analysis of the internet of things. *Journal of Network and Computer Applications*, 83:12–27, 2017.

[118] Daniela Genius, Letitia W. Li, and Ludovic Apvrille. Model-Driven Performance Evaluation and Formal Verification for Multi-level Embedded System Design. In *Conference on Model-Driven Engineering and Software Development (Modelsward'2017)*, Porto, Portugal, February 2017.

[119] Daniela Genius, Letitia W Li, Ludovic Apvrille, and Tullio Tanzi. Multi-level latency evaluation with an mde approach. In *6th International Conference on Model-Driven Rngineering and Software Development (MODELSWARD 2018)*, 2018.

[120] Patrick George. Have new technologies made cars less safe? https://auto.howstuffworks.com/car-driving-safety/safety-regulatory-devices/new-technologies-cars-less-safe.htm, March 2010.

[121] Jack M. Germain. Can software kill you? https://www.technewsworld.com/story/33398.html, April 2004.

[122] Dave Gershgorn. Instead of hacking self-driving cars, researchers are trying to hack the world they see. https://qz.com/1031233/instead-of-hacking-self-driving-cars-researchers-are-trying-to-hack-the-world-they-see/, July 2017.

[123] Robert L. Glass. Real-time: The "Lost World" of software debugging and testing. *Commun. ACM*, 23(5):264–271, May 1980.

[124] F. S. Gonçalves, D. Pereira, E. Tovar, and L. B. Becker. Formal Verification of AADL Models Using UPPAAL. In *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 117–124, Nov 2017.

[125] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. *NDSS (Feb. 2017)*, 2017.

[126] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation framework for executing component and connector models of self-driving vehicles. In *20th International Conference on Model Driven Engineering Languages and Systems MODELS 2017*, pages 109–115, 2017.

[127] Andy Greenberg. Car hack technique uses dealerships to spread malware. https://www.wired.com/2015/10/car-hacking-tool-turns-repair-shops-malware-brothels/, October 2015.

[128] Andy Greenberg. A deep flaw in your car lets hackers shut down safety features. https://www.wired.com/story/car-hack-shut-down-safety-features/, August 2017.

[129] B. Greene. Agile methods applied to embedded firmware development. In *Agile Development Conference*, pages 71–77, Jun 2004.

[130] Hans Grönninger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased modeling. *arXiv preprint arXiv:1409.6623*, 2014.

[131] Shay Gueron, Simon Johnson, and Jesse Walker. Sha-512/256. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 354–358. IEEE, 2011.

[132] Darko Gvozdanović, Saša Dešić, and Darko Huljenić. UML Supported Software Design. In *International Conference on Software, Telecommunications and Computer Networks SoftCOM 2001*, 2001.

[133] Axel Habermaier, Johannes Leupolz, and Wolfgang Reif. Unified simulation, visualization, and formal analysis of safety-critical systems with s. In Maurice H. ter Beek, Stefania Gnesi, and Alexander Knapp, editors, *Critical Systems: Formal Methods and Automated Verification*, pages 150–167, Cham, 2016. Springer International Publishing.

[134] Kristen Hall-Geisler. Even your connected car will need antivirus software. https://techcrunch.com/2016/05/02/even-your-connected-car-will-need-antivirus-software/, May 2016.

[135] Brahim Hamid, Jacob Geisel, Adel Ziani, Jean-Michel Bruel, and Jon Perez. Model-driven engineering for trusted embedded systems based on security and dependability patterns. In *International SDL Forum*, pages 72–90. Springer, 2013.

[136] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B Bobba. A design-space exploration for allocating security tasks in multicore real-time systems. *arXiv preprint arXiv:1711.04808*, 2017.

[137] Bill Haskins, Jonette Stecklein, Brandon Dick, Gregory Moroney, Randy Lovell, and James Dabney. 8.4.2 Error Cost Escalation Through the Project Life Cycle. In *INCOSE International Symposium*, volume 14, pages 1723–1737. Wiley Online Library, 2004.

[138] Denis Hatebur, Maritta Heisel, and Holger Schmidt. Security engineering using problem frames. In Günter Müller, editor, *Emerging Trends in Information and Communication Security*, pages 238–253, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[139] Klaus Havelund, Mike Lowry, SeungJoon Park, Charles Pecheur, John Penix, Willem Visser, Jonathan White, et al. Formal analysis of the remote agent before and after flight. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, volume 134, 2000.

[140] J. Heffley and P. Meunier. Can source code auditing software identify common vulnerabilities and be used to evaluate software security? In *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*, pages 10 pp.–, Jan 2004.

[141] Joerg Henkel and Rolf Ernst. The interplay of run-time estimation and granularity in hw/sw partitioning. In *Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, CODES '96, pages 52–, Washington, DC, USA, 1996. IEEE Computer Society.

[142] Olaf Henniger, Alastair Ruddle, Hervé Seudié, Benjamin Weyl, Marko Wolf, and Thomas Wollinger. Securing Vehicular On-Board IT Systems: The EVITA Project. In *VDI/VW Automotive Security Conference*, 2009.

[143] T. A. Henzinger and J. Sifakis. The discipline of embedded systems design. *Computer*, 40(10):32–40, Oct 2007.

[144] Thomas A Henzinger. Two challenges in embedded systems design: predictability and robustness. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 366(1881):3727–3736, 2008.

[145] Thomas A Henzinger and Joseph Sifakis. The embedded systems design challenge. In *International Symposium on Formal Methods*, pages 1–15. Springer, 2006.

[146] Nat Hillary. Measuring performance for real-time systems. *Freescale Semiconductor, November*, 2005.

[147] H. Holm, K. Shahzad, M. Buschle, and M. Ekstedt. P^2 cysemol: Predictive, probabilistic cyber security modeling language. *IEEE Transactions on Dependable and Secure Computing*, 12(6):626–639, Nov 2015.

[148] Jin B Hong and Dong Seong Kim. Assessing the effectiveness of moving target defenses using security models. *IEEE Transactions on Dependable and Secure Computing*, 13(2):163–177, 2016.

[149] Todd E. Humphreys, Brent M. Ledvina, Mark L. Psiaki, Brady W. O'Hanlon, and Paul M. Kintner Jr. Assessing the spoofing threat: Development of a portable gps civilian spoofer. In *Proceedings of the ION GNSS international technical meeting of the satellite division*, volume 55, page 56, 2008.

[150] Troy Hunt. Controlling vehicle features of Nissan LEAFs across the globe via vulnerable APIs. https://www.troyhunt.com/controlling-vehicle-features-of-nissan, December 2016.

[151] David D Hwang, Patrick Schaumont, Kris Tiri, and Ingrid Verbauwhede. Securing embedded systems. *IEEE Security & Privacy*, 4(2):40–49, 2006.

[152] Icon Labs. Floodgate IoT security toolkit. http://www.iconlabs.com/prod/products/floodgate-iot-security-toolkit, 2018.

[153] ICS-CERT. Hospira Lifecare PCA Infusion System Vulnerabilities, Advisory (icsa-15-125-01b). https://ics-cert.us-cert.gov/advisories/ICSA-15-125-01B, June 2015.

[154] Infineon Technologies AG. Aurix security hardware. https://www.infineon.com/cms/en/ product/microcontroller/ 32-bit-tricore-microcontroller/ aurix-safety-joins-performance/ aurix-security-solutions/aurix-security-hardware/, 2018.

[155] Terrance R Ingoldsby. Understanding risks through attack tree analysis. *Computer Security Journal*, 20(2):33–59, 2004.

[156] Institut Mines-Telecom. No autonomous cars without cybersecurity. https://blogrecherche.wp.imt.fr/en/2017/12/12/autonomous-cars-cybersecurity/, December 2017.

[157] National Instruments. Best practices for embedded software testing of safety compliant systems. http://www.ni.com/white-paper/13671/en/, January 2016.

[158] Pedro Isaias and Tomayess Issa. *Information System Development Life Cycle Models*, pages 21–40. Springer New York, New York, NY, 2015.

[159] Rob Miller Ishtiaq Rouf, Hossen Mustafa, Sangho Oh Travis Taylor, Wenyuan Xu, Marco Gruteser, Wade Trappe, and Ivan Seskar. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *19th USENIX Security Symposium, Washington DC*, pages 11–13, 2010.

[160] Road vehicles - Functional safety. Standard, 2011.

[161] J. A. Cook. Automatic code generation. https://www.ethz.ch/content/dam/ethz/special-interest/mavt/dynamic-systems-n-control/idsc-dam/Lectures/Embedded-Control-Systems/OtherNotes/Automatic_Code_Generation.pdf, Mar 2008.

[162] Chafic Jaber. *High-level SOC Modeling and Performance Estimation: Application To A Multi-core Implementation Of LTE EnodeB Physical Layer*. PhD thesis, Telecom Paristech, September 2011.

[163] Chafic Jaber, Andreas Kanstein, Ludovic Apvrille, Amer Baghdadi, Patricia Le Moenner, and Renaud Pacalet. High-Level System Modeling for Rapid HW/SW Architecture Exploration. In *Proc. of the 20th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP'2009)*, Jun 2009.

[164] Michael Jackson. Problem frames-analyzing and structuring software development problems. 2001. *New York, Oxoford: Addison-Wesley*, 390.

[165] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. ACM.

[166] K. Jiang, P. Eles, and Z. Peng. Co-design techniques for distributed real-time embedded systems with communication security constraints. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 947–952, March 2012.

[167] Anjali Joshi and Mats PE Heimdahl. Model-based safety analysis of Simulink models using SCADE design verifier. In *International Conference on Computer Safety, Reliability, and Security*, pages 122–135. Springer, 2005.

[168] Jan Jürjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 412–425, London, UK, UK, 2002. Springer-Verlag.

[169] Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[170] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. UML-based Multiprocessor SoC Design Framework. *ACM Trans. Embed. Comput. Syst.*, 5(2):281–320, May 2006.

[171] Frank Kargl and Norbert Bissmeyer. Y4&5 dissemination report. Technical Report Deliverable 6.4, PRESERVE Project, Jun 2015.

[172] Dawn Kawamoto. IoT security incidents rampant and costly. https://www.darkreading.com/vulnerabilities—threats/iot-security-incidents-rampant-and-costly/d/d-id/1329367, July 2017.

[173] Matt Kelly. With the rise of autonomous vehicles, hackers pose a serious new threat. https://www.news.virginia.edu/content/rise-autonomous-vehicles-hackers-pose-serious-new-threat, Apr 2017.

[174] S Kelly and S Frankel. Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec. Technical report, 2007.

[175] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, Dec 2000.

[176] Arjun Kharpal. After Samsung's Note 7, here's 10 of the biggest tech recalls ever. https://www.cnbc.com/2016/09/13/after-samsung-note-7-recall-10-biggest-tech-recalls-ever.html, September 2015.

[177] B. Kienhuis, E.F. Deprettere, P. van der Wolf, and K.A. Vissers. A Methodology to Design Programmable Embedded Systems: The Y-Chart Approach. In *Embedded Processor Design Challenges*, pages 18–37. Springer, 2002.

[178] Trevor A Kletz. *HAZOP and HAZAN: identifying and assessing process industry hazards*. IChemE, 1999.

[179] J. Kloos, T. Hussain, and R. Eschbach. Risk-based testing of safety-critical embedded systems driven by fault tree analysis. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 26–33, March 2011.

[180] Will Knight. Tesla Crash Will Shape the Future of Automated Cars. https://www.technologyreview.com/s/601829/tesla-crash-will-shape-the-future-of-automated-cars/, July 2016.

[181] D. Knorreck. *UML-based Design Space Exploration, Fast Simulation and Static Analysis*. PhD thesis, Telecom ParisTech, EDITE, October 2011.

[182] Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. Fast simulation techniques for design space exploration. In *Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, pages 308–327. Springer Berlin Heidelberg, 2009.

[183] Woo-Hyun Ko, B. Satchidanandan, and P. R. Kumar. Theory and implementation of dynamic watermarking for cybersecurity of advanced transportation systems. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 416–420, Oct 2016.

[184] Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. Security as a new dimension in embedded system design. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 753–760, New York, NY, USA, 2004. ACM. Moderator-Ravi, Srivaths.

[185] S. Konrad, B. H. C. Cheng, and L. A. Campbell. Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30(12):970–992, Dec 2004.

[186] Phil Koopman. Secrecy vs. integrity and why encryption might be the wrong choice. https://betterembsw.blogspot.fr/2013/10/secrecy-vs-integrity-and-why-encryption.html, October 2013.

[187] Barbara Kordy, Piotr Kordy, Sjouke Mauw, and Patrick Schweitzer. ADTool: Security Analysis with Attack-Defense Trees. In *Quantitative Evaluation of Systems*, volume 8054 of *Lecture Notes in Computer Science*, pages 173–176. Springer Berlin Heidelberg, 2013.

[188] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society.

[189] Bill Krause. Use processor redundancy for maximum reliability. https://www.eetimes.com/document.asp?doc_id=1277540, February 2002.

[190] Marina Krotofil, Jason Larsen, and Dieter Gollmann. The process matters: Ensuring data veracity in cyber-physical systems. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 133–144, New York, NY, USA, 2015. ACM.

[191] J. H. Lala and R. E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82(1):25–40, Jan 1994.

[192] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues. Ocarina: An environment for AADL models analysis and automatic code generation for high integrity applications. In *International Conference on Reliable Software Technologies*, pages 237–250. Springer, 2009.

[193] Edward A Lee. Cyber physical systems: Design challenges. In *Object oriented real-time distributed computing (isorc), 2008 11th IEEE international symposium on*, pages 363–369. IEEE, 2008.

[194] Tim Leinmüller, Levente Buttyan, Jean-Pierre Hubaux, Frank Kargl, Rainer Kroh, Panagiotis Papadimitratos, Maxim Raya, and Elmar Schoch. Sevecom-secure vehicle communication. In *IST Mobile and Wireless Communication Summit*, number LCA-POSTER-2008-005, 2006.

[195] Laurens Lemaire, Jorn Lapon, Bart De Decker, and Vincent Naessens. A SysML extension for security analysis of industrial control systems. In *2nd International Symposium for ICS & SCADA Cyber Security Research 2014*, pages 1–9. BCS Learning & Development Ltd., 2014.

[196] Arjen K Lenstra and Eric R Verheul. Selecting cryptographic key sizes. *Journal of cryptology*, 14(4):255–293, 2001.

[197] Nancy G. Leveson. Software Safety in Embedded Computer Systems. *Commun. ACM*, 34(2):34–46, February 1991.

[198] Nancy G Leveson and Clark S Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[199] Doron Levin. Here are some of worst car scandals in history. http://fortune.com/2015/09/26/auto-industry-scandals/, September 2015.

[200] Letitia W. Li, Ludovic Apvrille, and Annie Bracquemond. Design and Verification of Secure Autonomous Vehicles. In *Intelligent Transportation Systems 2017)*, Strausbourg, France, June 2017.

[201] Letitia W. Li, Florian Lugou, and Ludovic Apvrille. Security-Aware Modeling and Analysis for HW/SW Partitioning. In *Conference on Model-Driven Engineering and Software Development (Modelsward'2017)*, Porto, Portugal, February 2017.

[202] Letitia W. Li, Florian Lugou, and Ludovic Apvrille. Security modeling for embedded system design. In *International Workshop on Graphical Models for Security*, pages 99–106. Springer, 2017.

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

[204] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[205] Chung-Wei Lin, Bowen Zheng, Qi Zhu, and Alberto Sangiovanni-Vincentelli. Security-Aware Design Methodology and Optimization for Automotive Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 21(1):18, 2015.

[206] Torsten Lodderstedt, David A. Basin, and Jurgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML'02, pages 426–441, London, UK, UK, 2002. Springer-Verlag.

[207] G. Lu, D. Zeng, and B. Tang. Anti-jamming filtering for drfm repeat jammer based on stretch processing. In *2010 2nd International Conference on Signal Processing Systems*, volume 1, pages V1–78–V1–82, July 2010.

[208] Florian Lugou. *Environments for Analyzing the Security of Smart Objects*. PhD thesis, Telecom Paristech, February 2018.

[209] Florian Lugou, Letitia W. Li, Ludovic Apvrille, and Rabea Ameur-Boulifa. SysML Models and Model Transformation for Security. In *Conference on Model-Driven Engineering and Software Development (Modelsward'2016)*, Rome, Italy, February 2016.

[210] Georg Macher, Andrea Höller, Harald Sporer, Eric Armengaud, and Christian Kreiner. A combined safety-hazards and security-threat analysis method for automotive systems. In Floor Koornneef and Coen van Gulijk, editors, *Computer Safety, Reliability, and Security*, pages 237–250, Cham, 2015. Springer International Publishing.

[211] Bharat B. Madan, Katerina Goševa-Popstojanova, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. A method for modeling and quantifying the security attributes of intrusion tolerant systems. *Performance Evaluation*, 56(1):167 – 186, 2004. Dependable Systems and Networks - Performance and Dependability Symposium (DSN-PDS) 2002: Selected Papers.

[212] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 147–152, New York, NY, USA, 1997. ACM.

[213] Shahar Maoz, Ferdinand Mehlan, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. OCL framework to verify extra-functional properties in component and connector models.

In *20th International Conference on Model Driven Engineering Languages and Systems MODELS 2017*, pages 24–30, 2017.

[214] Eduard Marin, Dave Singelée, Flavio D Garcia, Tom Chothia, Rik Willems, and Bart Preneel. On the (in) security of the latest generation implantable cardiac defibrillators and how to secure them. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 226–236. ACM, 2016.

[215] Raluca Marinescu, Henrik Kaijser, Marius Mikučionis, Cristina Seceleanu, Henrik Lönn, and Alexandre David. Analyzing industrial architectural models by simulation and model-checking. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 189–205. Springer, 2014.

[216] John Markoff. A guide to challenges facing self-driving car technologists. https://www.nytimes.com/2017/06/07/technology/autonomous-car-technology-challenges.html, June 2017.

[217] Peter Marwedel. Embedded and cyber-physical systems in a nutshell. *DAC. COM Knowledge Center Article*, 20(10), 2010.

[218] Peter Marwedel and Gert Goossens. *Code generation for embedded processors*, volume 317. Springer Science & Business Media, 2013.

[219] D. Maynor. SCADA Security and Terrorism: We're Not Crying Wolf! In *Invited presentation at BlackHat BH 2006. Presentation available at: https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Maynor-Graham-up.pdf*, USA, 2006.

[220] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[221] Microsoft. The STRIDE Threat Model. https://msdn.microsoft.com/en-us/library/ee823878%28v=cs.20%29.aspx, 2005.

[222] Microsoft. What is the Security Development Lifecycle. http://www.microsoft.com/en-us/sdl/default.aspx, 2018.

[223] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

[224] Paolo Modesti. Anbx: Automatic generation and verification of security protocols implementations. In *International Symposium on Foundations and Practice of Security*, pages 156–173. Springer, 2015.

[225] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications. In *2009 International Conference on Availability, Reliability and Security*, pages 841–846, March 2009.

[226] Sumit Mohanty, Viktor K Prasanna, Sandeep Neema, and J Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *ACM SIGPLAN Notices*, 37(7):18–27, 2002.

[227] T. G. Moreira, M. A. Wehrmeister, C. E. Pereira, J. F. Pétin, and E. Levrat. Automatic code generation for embedded systems: From UML specifications to VHDL code. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 1085–1090, July 2010.

[228] Martin Moser. The PRESERVE V2X Security Subsystem. https://www.preserve-project.eu/sites/preserve-project.eu/files/preserve-ws-03-vss.pdf, June 2015.

[229] Danielle Muoio. 6 scenarios self-driving cars still can't handle. http://www.businessinsider.fr/us/autonomous-car-limitations-2016-8, August 2016.

[230] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, December 2002.

[231] Andrey Naumenko and Alain Wegmann. A metamodel for the Unified Modeling Language: critical analysis and solution. Technical report, 2002.

[232] NBC. Hyundai Recalls 41K SUVs Over Software Glitch. https://www.nbcwashington.com/news/national-international/Hyundai-Recalls-SUVs-Software-Flaw-May-Stop-Acceleration-393663311.html, May 2016.

[233] Lily Newman. The Botnet That Broke the Internet Isn't Going Away. https://www.wired.com/2016/12/botnet-broke-internet-isnt-going-away/, December 2016.

[234] Sen Nie, Ling Liu, and Yuefeng Du. Free-Fall: Hacking Tesla from Wireless to CAN Bus. https://www.blackhat.com/docs/us-17/thursday/us-17-Nie-Free-Fall-Hacking-Tesla-From-Wireless-To-CAN-Bus-wp.pdf, July 2017.

[235] No Magic. Choosing the right modeling tool. https://www.nomagic.com/getting-started/choosing-the-right-modeling-tool, 2018.

[236] Tammy Noergaard. *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Newnes, 2012.

[237] Bashar Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–119, 2001.

[238] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46. ACM, 2000.

[239] P. Nuzzo, J. B. Finn, A. Iannopollo, and A. L. Sangiovanni-Vincentelli. Contract-based design of control protocols for safety-critical cyber-physical systems. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2014.

[240] Robert Oates, Fran Thom, and Graham Herries. Security-aware, model-based systems engineering with sysml. In *Proceedings of the 1st International Symposium on ICS & SCADA Cyber Security Research*, pages 78–87. BCS, 2013.

[241] Marcus Obst, Laurens Hobert, and Pierre Reisdorf. Multi-sensor data fusion for checking plausibility of V2V communications by vision-based multiple-object tracking. In *Vehicular Networking Conference (VNC), 2014 IEEE*, pages 143–150. IEEE, 2014.

[242] Object Management Group OMG. SysML. In *http://www.sysml.org/*, 2011.

[243] Jonathan S Ostroff. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, 18(1):33–60, 1992.

[244] OVERSEE. Open Vehicular Secure Platform. https://www.oversee-project.com/, 2010.

[245] Miroslav Pajic, James Weimer, Nicola Bezzo, Oleg Sokolsky, George J Pappas, and Insup Lee. Design and implementation of attack-resilient cyberphysical systems: With a focus on attack-resilient state estimators. *IEEE Control Systems*, 37(2):66–81, 2017.

[246] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano Zanero. A stealth, selective, link-layer denial-of-service attack against automotive networks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 185–206, Cham, 2017. Springer International Publishing.

[247] Panagiotis Papadimitratos, Levente Buttyan, Tamás Holczer, Elmar Schoch, Julien Freudiger, Maxim Raya, Zhendong Ma, Frank Kargl, Antonio Kung, and Jean-Pierre Hubaux. Secure vehicular communication systems: design and architecture. *IEEE Communications Magazine*, 46(11), 2008.

[248] Panagiotis Papadimitratos, Levente Buttyan, Jean-Pierre Hubaux, Frank Kargl, Antonio Kung, and Maxim Raya. Architecture for secure and private vehicular communications. In *Telecommunications, 2007. ITST'07. 7th International Conference on ITS*, pages 1–6. IEEE, 2007.

[249] Corina S. Păsăreanu and Willem Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, pages 164–181, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[250] Paul Ohmart. How Much Data Can You Encrypt With RSA Keys? https://info.townsendsecurity.com/bid/29195/how-much-data-can-you-encrypt-with-rsa-keys, Apr 2011.

[251] PAX Team. NOEXEC. http://pax.grsecurity.net/docs/noexec.txt, May 2003.

[252] G. Pedroza, L. Apvrille, and D. Knorreck. AVATAR: A SysML Environment for the Formal Verification of Safety and Security Properties. In *2011 11th Annual International Conference on New Technologies of Distributed Systems*, pages 1–10, May 2011.

[253] Mert D. Pesé, Karsten Schmidt, and Harald Zweck. Hardware/Software Co-Design of an Automotive Embedded Firewall. https://web.eecs.umich.edu/ mpese/papers/2017-01-1659.pdf, 2017.

[254] J. Petit and S. E. Shladover. Potential cyberattacks on automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):546–556, April 2015.

[255] Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. Remote attacks on automated vehicles sensors: Experiments on camera and lidar. *Black Hat Europe*, 11:2015, 2015.

[256] Ludovic Piètre-Cambacédès and Marc Bouissou. Modeling safety and security interdependencies with BDMP (Boolean logic Driven Markov Processes). In *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, pages 2852–2861. IEEE, 2010.

[257] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, Feb 2006.

[258] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, 2001.

[259] Polarsys. ARCADIA/CAPELLA (webpage). In *https://www.polarsys.org/capella/arcadia.html*, 2008.

[260] Ponemon Institute LLC. The impact of data breaches on reputation & share value. https://www.centrify.com/media/4737054/ponemon_data_breach_impact_study.pdf, May 2017.

[261] Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In Rajeev Alur and George J. Pappas, editors, *Hybrid Systems: Computation and Control*, pages 477–492, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[262] Nikolaos Priggouris, Adeline Silva, Markus Shawky, Magnus Persson, Vincent Ibanez, Joseph Machrouh, Nicola Meledo, Philippe Baufreton, and Jason Mansell Rementeria. The system design life cycle. In *CESAR-Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*, pages 15–67. Springer, 2013.

[263] pymnts.com. Can A Connected Refrigerator Anchor The IoT Household? https://www.pymnts.com/intelligence-of-things/2017/can-a-connected-refrigerator-anchor-the-iot-household/, May 2017.

[264] Christian Raspotnig, Peter Karpati, and Andreas L Opdahl. Combined Assessment of Software Safety and Security Requirements: An Industrial Evaluation of the CHASSIS Method. *Journal of Cases on Information Technology (JCIT)*, 20(1):46–69, 2018.

[265] Christian Raspotnig, Vikash Katta, Peter Karpati, and Andreas L Opdahl. Enhancing CHASSIS: a method for combining safety and security. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 766–773. IEEE, 2013.

[266] D. R. Raymond and S. F. Midkiff. Denial-of-service in wireless sensor networks: Attacks and defenses. *IEEE Pervasive Computing*, 7(1):74–81, Jan 2008.

[267] L Ricci and L McGinness. Embedded system security. white paper, 2004.

[268] Nils Rodday. Hacking a Professional Drone. Slides at www.blackhat.com/docs/asia-16/materials/asia-16-Rodday-Hacking-A-Professional-Drone.pdf, March 2016.

[269] Jussi Ronkainen and Pekka Abrahamsson. Software development under stringent hardware constraints: Do agile methods have a chance? In *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 73–79. Springer, 2003.

[270] Paul Rook. Controlling software projects. *Software Engineering Journal*, 1(1):7–16, 1986.

[271] Rafael Rosales, Michael Glass, Jürgen Teich, Bo Wang, Yang Xu, and Ralph Hasholzner. MAESTRO— Holistic Actor-Oriented Modeling of Nonfunctional Properties and Firmware Behavior for MPSoCs. *ACM Trans. Des. Autom. Electron. Syst.*, 19(3):23:1–23:26, June 2014.

[272] Brian Ross, Cindy Galli, Stephanie Zimmermann, Cho Park, and Pete Madden. BMW recalls 1 million vehicles for fire risk. http://abcnews.go.com/US/bmw-recalls-million-vehicles-fire-risk/story?id=50922136, November 2017.

[273] Winston W Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society Press, 1987.

[274] A. Ruddle, D. Ward, B. Weyl, S. Idrees, Y. Roudier, M. Friedewald, T. Leimbach, A. Fuchs, S. Gürgens, O. Henniger, R. Rieke, M. Ritscher, H. Broberg, L. Apvrille, R. Pacalet, and G. Pedroza. Security requirements for automotive on-board networks based on dark-side scenarios. Technical Report Deliverable D2.3, EVITA Project, 2009.

[275] Jose Fran Ruiz, Rajesh Harjani, Antonio Mana, Vasily Desnitsky, Igor Kotenko, and Andrey Chechulin. A methodology for the analysis and modeling of security threats and attacks for systems of embedded components. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 261–268. IEEE, 2012.

[276] Sara Sadvandi, Nicolas Chapon, and Ludovic Piètre-Cambacédès. Safety and Security Interdependencies in Complex Systems and SoS: Challenges and Perspectives. In Omar Hammami, Daniel Krob, and Jean-Luc Voirin, editors, *Complex Systems Design & Management*, pages 229–241, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[277] Safety.com. The top 40 best wearable tech products for kids and families. https://www.safety.com/best-wearables, July 2017.

[278] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design Test of Computers*, 18(6):23–33, Nov 2001.

[279] Tripti Saxena and Gabor Karsai. *MDE-Based Approach for Generalizing Design Space Exploration*, pages 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[280] Eric Schlaepfer. Comparison of internal and external watchdog timers. https://www.maximintegrated.com/en/app-notes/index.mvp/id/4229, June 2008.

[281] Christian Schleiffer, Marko Wolf, André Weimerskirch, and Lars Wolleschensky. Secure key management-a key feature for modern vehicle electronics. Technical report, SAE Technical Paper, 2013.

[282] Bastian Schlich and Stefan Kowalewski. Model checking c source code for embedded systems. *International Journal on Software Tools for Technology Transfer*, 11(3):187–202, Jul 2009.

[283] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[284] Holger Schmidt. Pattern-based confidentiality-preserving refinement. In *International Symposium on Engineering Secure Software and Systems*, pages 43–59. Springer, 2009.

[285] Christoph Schmittner, Thomas Gruber, Peter Puschner, and Erwin Schoitsch. Security Application of Failure Mode and Effect Analysis (FMEA). In Andrea Bondavalli and Felicita Di Giandomenico, editors, *Computer Safety, Reliability, and Security*, pages 310–325, Cham, 2014. Springer International Publishing.

[286] Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.

[287] Erwin Schoitsch. Design for safety and security of complex embedded systems: A unified approach. In *Proceedings of the NATO Advanced Research Workshop on Cyberspace Security and Defense: Research Issues*, pages 161–174. Springer, 2005.

[288] H. Schweppe, Y. Roudier, B. Weyl, L. Apvrille, and D. Scheuermann. Car2X Communication: Securing the Last Meter - A Cost-Effective Approach for Ensuring Trust in Car2X Applications Using In-Vehicle Symmetric Cryptography. In *2011 IEEE Vehicular Technology Conference (VTC Fall)*, pages 1–5, Sept 2011.

[289] Hendrik Schweppe. *Security and privacy in automotive on-board networks*. PhD thesis, Télécom ParisTech, 2012.

[290] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, Sept 2003.

[291] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, Sept 2003.

[292] H. Seudié, J. Shokrollahi, B. Weyl, A. Keil, M. Wolf, F. Zweers, T. Gendrullis, M. S. Idrees, Y. Roudier, H. Schweppe, H. Platzdasch, R. El Khayari, O. Henniger, D. Scheuermann, L. Apvrille, and G. Pedroza. Secure on-board architecture specification. Technical Report Deliverable D3.2, EVITA Project, 2010.

[293] Hervé Seudié. Vehicular on-board security: Evita project. https://www.evita-project.org/Publications/Seu09.pdf, November 2009.

[294] Hocheol Shin, Dohyun Kim, Yujin Kwon, and Yongdae Kim. Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 445–467, Cham, 2017. Springer International Publishing.

[295] Tom Simonite. Self-driving cars' spinning-laser problem. https://www.technologyreview.com/s/603885/autonomous-cars-lidar-sensors/, March 2017.

[296] Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, Jan 2005.

[297] SocLib consortium. The SoCLib project: An integrated system-on-chip modelling and simulation platform, 2003. www.soclib.fr.

[298] Jason Sparapani. Driverless cars not a 'solved problem,' says MIT professor. https://searchcio.techtarget.com/blog/TotalCIO/Driverless-cars-not-a-solved-problem-says-MIT-professor, February 2016.

[299] Max Steiner and Peter Liggesmeyer. Combination of safety and security analysis - finding security problems that threaten the safety of a system. In *SAFECOMP 2013-Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security*, 2013.

[300] Jack Stewart. Tesla's Autopilot Was Involved in Another Deadly Car Crash. https://www.wired.com/story/tesla-autopilot-self-driving-crash-california/, March 2018.

[301] Luke Swartz. Overwhelmed by technology: How did user interface failures on board the USS Vincennes lead to 290 dead. *Erişim tarihi*, 25, 2001.

[302] Synopsys. Architecture Risk Analysis, 2018. https://www.synopsys.com/software-integrity/software-security-services/software-architecture-design/risk-analysis.html.

[303] Synopsys. Scalable SoC Verification, 2018. https://www.synopsys.com/verification.html.

[304] Synopsys. Security Control Design Analysis (SCDA), 2018. https://www.synopsys.com/software-integrity/software-security-services/software-architecture-design/security-control-design-analysis.html.

[305] Synopsys. SSDLC 101: What is the secure software development life cycle?, 2018. https://www.synopsys.com/blogs/software-security/secure-sdlc/.

[306] Synopsys Editorial Team. Infographic: A lack of software security training puts companies at risk. https://www.synopsys.com/blogs/software-security/software-security-training-resources-infographic/, January 2018.

[307] Tivadar Szemethy and Gabor Karsai. Platform modeling and model transformations for analysis. *Journal of Universal Computer Science*, 10(10):1383–1407, 2004.

[308] A. Taylor, N. Japkowicz, and S. Leblanc. Frequency-based anomaly detection for the automotive can bus. In *2015 World Congress on Industrial Control Systems Security (WCICSS)*, pages 45–49, Dec 2015.

[309] A. Taylor, S. P. Leblanc, and N. Japkowicz. Probing the limits of anomaly detectors for automobiles with a cyber attack framework. *IEEE Intelligent Systems*, PP(99):1–1, 2018.

[310] P. Tessier, S. Gerard, C. Mraidha, F. Terrier, and J. M. Geib. A component-based methodology for embedded system prototyping. In *14th IEEE International Workshop on Rapid Systems Prototyping, 2003. Proceedings.*, pages 9–15, June 2003.

[311] Lothar Thiele and W Ernesto. Performance analysis of distributed embedded systems. In *In International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS*. Citeseer, 2011.

[312] V. L. L. Thing and J. Wu. Autonomous vehicle security: A taxonomy of attacks and defences. In *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 164–170, Dec 2016.

[313] Donald E Thomas, Jay K Adams, and Herman Schmit. A model and methodology for hardware-software codesign. *IEEE Design & test of computers*, 10(3):6–15, 1993.

[314] Cadie Thompson. As healthcare costs rise and patients demand better care, hospitals turn to new technologies. http://www.businessinsider.fr/us/how-hospitals-are-using-iot-2016-10/, October 2016.

[315] Syrine Tlili, XiaoChun Yang, Rachid Hadjidj, and Mourad Debbabi. Verification of CERT Secure Coding Rules: Case Studies. In Robert Meersman, Tharam Dillon, and Pilar Herrero, editors, *On*

the Move to Meaningful Internet Systems: OTM 2009, pages 913–930, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[316] Juha-Pekka Tolvanen, Metacase Janne Luoma, and De-Jiu Chen. Reaping the benefits of architectural modeling in embedded design. https://www.embedded.com/design/prototyping-and-development/4437065/Reaping-the-benefits-of-architectural-modeling-in-embedded-design, November 2014.

[317] Angelo Troina, Alessandro Aldini, and Roberto Gorrieri. A probabilistic formulation of imperfect cryptography. In *Proc. of 1st Int. Workshop on Issues in Security and Petri Nets, WISP*, volume 3. Citeseer, 2003.

[318] Trusted Computing Group. Trusted Platform Module (TPM) Summary, April 2008.

[319] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security & Privacy*, 3(6):81–84, 2005.

[320] UAVCAN development team. Hardware design recommendations. http://uavcan.org/Specification/8._Hardware_design_recommendations/#devices-with-different-number-of-redundant-interfaces, March 2018.

[321] David I. Urbina, Jairo A. Giraldo, Alvaro A. Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. Limiting the impact of stealthy attacks on industrial control systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1092–1105, New York, NY, USA, 2016. ACM.

[322] Anton V Uzunov, Eduardo B Fernandez, and Katrina Falkner. ASE: a Comprehensive Pattern-Driven Security Methodology for Distributed Systems. *Computer Standards & Interfaces*, 41:112–137, 2015.

[323] Lisa Vaas. Uber car software detected woman before fatal crash but failed to stop. https://nakedsecurity.sophos.com/2018/05/09/uber-car-software-detected-woman-before-fatal-crash-but-failed-to-stop/, May 2018.

[324] Michael Vai, David J. Whelihan, Benjamin R. Nahill, Daniil M. Utin, Sean R. O'Melia, and Roger I. Khazan. Secure embedded systems. Technical report, MIT Lincoln Laboratory Lexington United States, 2016.

[325] Axel van Lamsweerde. Elaborating Security Requirements by Construction of Intentional Anti-Models. *In Proc. of the 26th International Conference on Software Engineering , ICSE '04*, pages 148–157, 2004.

[326] Axel Van Lamsweerde, Simon Brohez, Renaud De Landtsheer, David Janssens, et al. From system goals to intruder anti-goals: attack generation and resolution for security requirements engineering. *Proc. of RHAS*, 3:49–56, 2003.

[327] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Software & Systems Modeling*, 3(2):85–113, May 2004.

[328] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205 – 227, 2002. Special Issue on Applications of Graph Transformations (GRATRA 2000).

[329] Maria Vasilevskaya and Simin Nadjm-Tehrani. *Quantifying Risks to Data Assets Using Formal Metrics in Embedded System Design*, pages 347–361. Springer International Publishing, Cham, 2015.

[330] William E Vesely, Francine F Goldberg, Norman H Roberts, and David F Haasl. Fault tree handbook. Technical report, Nuclear Regulatory Commission Washington DC, 1981.

[331] Jorgiano Vidal, Florent de Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *Design, Automation and Test in Europe*, pages 226–231, Dresden, Germany, April 2009.

[332] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, Apr 2003.

[333] Bill Vlasic and Nick Bunkley. Toyota will fix or replace 4 million gas pedals. http://www.nytimes.com/2009/11/26/business/26toyota.html, November 2009.

[334] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):339–390, Sep 2013.

[335] Jack Wallen. Five nightmarish attacks that show the risks of IoT security. http://www.zdnet.com/article/5-nightmarish-attacks-that-show-the-risks-of-iot-security/, June 2017.

[336] C. Wang, Z. Zhao, L. Gong, L. Zhu, Z. Liu, and X. Cheng. A Distributed Anomaly Detection System for In-Vehicle Network using HTM. *IEEE Access*, PP(99):1–1, 2018.

[337] Elizabeth Weise. Chinese group hacks a Tesla for the second year in a row. https://www.usatoday.com/story/tech/2017/07/28/chinese-group-hacks-tesla-second-year-row/518430001/, July 2017.

[338] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. E. Heimdahl, and S. Rayadurgam. Your "what" is my "how": Iteration and hierarchy in system design. *IEEE Software*, 30(2):54–60, March 2013.

[339] Michael E Whitman and Herbert J Mattord. *Principles of information security*. Cengage Learning, 2011.

[340] Anton Wijs and Luc Engelen. Refiner: Towards formal verification of model transformations. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, pages 258–263, Cham, 2014. Springer International Publishing.

[341] James Williams, Athanasios Zolotas, Nicholas Matragkas, Louis M Rose, Dimitios S Kolovos, Richard F Paige, and Fiona Polack. What do metamodels really look like? 1078, 01 2013.

[342] Wired. How connectivity is driving the future of the car. https://www.wired.com/brandlab/2016/02/how-connectivity-is-driving-the-future-of-the-car/, February 2016.

[343] Ben Wojdyla. The top automotive engineering failures: The ford pinto fuel tanks. https://www.popularmechanics.com/cars/a6700/top-automotive-engineering-failures-ford-pinto-fuel-tanks/, May 2011.

[344] Marko Wolf and Timo Gendrullis. Design, implementation, and evaluation of a vehicular hardware security module. In *International Conference on Information Security and Cryptology*, pages 302–318. Springer, 2011.

[345] Samuel Woo, Hyo Jin Jo, and Dong Hoon Lee. A practical wireless attack on the connected car and security protocol for in-vehicle CAN. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):993–1006, 2015.

[346] Chen Yan, Wenyuan Xu, and Jianhao Liu. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. *DEF CON*, 24, 2016.

[347] William Young and Nancy G. Leveson. An Integrated Approach to Safety and Security Based on Systems Theory. *Commun. ACM*, 57(2):31–35, February 2014.

[348] T. Zhang, H. Antunes, and S. Aggarwal. Defending connected vehicles against malware: Challenges and a solution framework. *IEEE Internet of Things Journal*, 1(1):10–21, Feb 2014.

[349] YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005.

[350] Roberto Zunino and Pierpaolo Degano. A note on the perfect encryption assumption in a process calculus. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, pages 514–528, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[351] Wei Zuo, Louis-Noel Pouchet, Andrey Ayupov, Taemin Kim, Chung-Wei Lin, Shinichi Shiraishi, and Deming Chen. Accurate High-level Modeling and Automated Hardware/Software Co-design for Effective SoC Design Space Exploration. In *Proc. 54th Annual Design Automation Conference 2017*, DAC '17, pages 78:1–78:6, New York, NY, USA, 2017. ACM.