Cyber-security for embedded systems: methodologies, techniques and tools


Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (32nd cycle)

Cyber-security for embedded systems: methodologies, techniques and tools

Sebastiano Fabrizio Finocchiaro

* * * * * *

Supervisor
Prof. Gianpiero Cabodi

Doctoral Examination Committee:
Prof. Aurelien Francillon, Referee, EURECOM
Prof. Joao Marques Silva, Referee, University of Toulouse
Prof. Pietro Laface, Politecnico di Torino
Prof. Luciano Lavagno, Politecnico di Torino
Prof. Silvio Ranise, Fondazione Bruno Kessler

Politecnico di Torino
2020

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Sebastiano Fabrizio Finocchiaro

Turin, 2020

Summary

The last two decades have produced an enormous change in technical systems that revolutionized the way how people live their lives, organizations conduct business and even the notion of society itself. This tendency is far from ending soon, rather the future will be ubiquitously connected by computing environments that pervade our whole life. These environments (not merely hardware/software systems, but they are inherently designed for the cyberspace, and perhaps in the future will be integrated also in the human body) are now termed as Cyber-Physical Systems (CPSs). CPSs are made up by a heterogeneous set of subsystems: embedded computers, actuators, distributed sensors, physical processes are integrated together over a communication network. It has been anticipated that CPSs are likely to take over traditional computing systems thanks to adaptability, safety, robustness, efficiency, and reliability. The level of predominant presence of such systems will dictate the development of intuitive and intelligent user interfaces, some of them manually operated and others autonomously controlled. The interfaces will provide service delivering and communication effectively and efficiently, thus enabling sophisticated ambient intelligence systems. Today, CPSs are typically deployed in manufacturing, medical devices, civil infrastructure, smart grids, transportation, industrial environments and so on. As society employs more and more ubiquitous computing technologies, individuals, organisations and companies are increasingly depending on them. It is already challenging and it will be even more to orient oneself among the extensive amounts of services and information. On one hand, this trend will lead to an increase of sensitive and critical operations. On the other hand, its very ubiquity (almost imperative presence in every aspect of society) will entice people to expect a comforting nature from it, making people forget about security requirements yet especially crucial.

Nowadays, the use of CPSs in some strategic areas of our society, where human life is directly affected, is of vital importance: from medicine to traffic control to stock exchange markets and international finance. We are progressively relying on such systems, even depending in some cases, yet a question raises concerns: *Do we trust enough these systems to let them manage our lives?*. This question hides the growing need for new and more effective security approaches in embedded systems and it is the motivation of this work.

Historically, security research and development were born in the IT area. Traditional IT security methods use the notions of confidentiality, integrity and availability. They, too, can be applied to CPSs security. However, they do not cover the full spectrum of CPSs security, hence new challenges are open to be solved.

In particular, the effort has been devoted to verification techniques and methodologies. The main focus of this work is on formal verification of security properties that guarantee confidentiality, integrity and availability. Starting from state-of-the-art formal verification techniques (model checking, equivalence checking, etc.) we developed new approaches for security-critical systems. We defined a portfolio of Taint properties and provided experimental results for hardware systems that employ root of trust support and remote attestation. From the experience of applying Taint properties on real architectures we realised that this class of properties is able to capture some specific behaviors of a system, neglecting others. We therefore proposed a new class of properties called Path properties that are able to capture information flow behaviors. A portfolio of this properties has been presented and experimental results have been provided to support our claim. Eventually we thought that a combined approach would be useful, so we presented a verification approach to verify both Taint properties and Path properties in a standard model checker. Ground breaking attacks such as Spectre and Meltdown made us wonder whether it would be possible to verify hardware designs against this kind of attacks. We then proposed a novel verification approach towards out-of-order speculative execution microprocessors that are prone to side channel attacks like Spectre and Meltdown. Our verification methods rely on widely known abstraction and reduction techniques for large designs, but they have been paired with the application of a information flow verification approach.

Acknowledgements

I would like to acknowledge whoever supported me in any possible way throughout this journey. My gratitude firstly goes to my supervisor Gianpiero Cabodi. I would also like to acknowledge Prof. Paolo Camurati and Prof. Stefano Quer. My colleagues with whom I shared ups and downs: Francesco, Danilo, Denis, Marco, Paolo, Evelina, Federica, Federica, Alberto.
After all I would like to thank every person, friend or not, who smiled at me.

Alla mia famiglia.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The last two decades have produced an enormous change in technical systems that revolutionized the way how people live their lives, organizations conduct business and even the notion of society itself. The term cyberspace describes this trend: the networked universe where part of our daily routines live. It is so extremely interconnected that even a small ripple on this sea is propagated widely through other parts of the cyberspace. This tendency is far from ending soon, rather the future will be ubiquitously connected by computing environments that pervade our whole life. The mutual dependency will be stronger than ever and data plays the most important role in it. These environments are not merely hardware/software systems, but they are inherently designed for the cyberspace. Indeed they are now termed as Cyber-Physical Systems (CPSs) [1].

CPSs are made up by a heterogeneous set of subsystems: embedded computers, actuators, distributed sensors, physical processes are integrated together over a communication network. It has been anticipated that CPSs are likely to take over traditional computing systems thanks to adaptability, safety, robustness, efficiency, and reliability [2].

CPSs are and will be capable of providing several degrees of computational power for different needs. CPSs are very helpful for our daily life but they all will be bounded to the most crucial aspect: data and the transmission of huge quantity of it. All this traffic is organized into different networking schemes, CPS tend not to be operate alone but in structured organizations. The schemes vary from dynamic ad-hoc peer-to-peer networks to the standard client-server architectures,

including a diversified heap of hybrids in between. The level of predominant presence of such systems will dictate the development of intuitive and intelligent user interfaces, some of them manually operated and others autonomously controlled. The interfaces will provide service delivering and communication effectively and efficiently, thus enabling sophisticated ambient intelligence systems.
Today, CPSs are typically deployed in manufacturing [3], medical devices, civil infrastructure [4], smart grids, transportation [5], industrial environments [6, 7] and so on. As society employs more and more ubiquitous computing technologies, individuals, organisations and companies are increasingly depending on them. It is already challenging and will be even more to orient oneself among the extensive amounts of services and information. On the one hand, this trend will lead to an increase of sensitive and critical operations. On the other hand, its very ubiquity (almost imperative presence in every aspect of society) will entice people to expect a comforting nature from it, making people forget about security requirements yet especially crucial [8, 9].

Nowadays, the use of CPSs in some strategic areas of our society, where human life is directly affected, is of vital importance: from medicine to traffic control to stock exchange markets and international finance. We are progressively relying on such systems, even depending in some cases, yet a question raises concerns:
"Do we have enough confidence in computer systems that we let them handle our most valuable goods, namely our life and money?"[10].
The term *confidence* translates to the notion of *dependability*, of which a useful definition has been provided by [11]:
"Dependability of a computing system is the ability to deliver service that can justifiably be trusted."

This definition underlines two aspects that affect the design, the implementation and our reasoning about system "correctness":

- Dependability does not explicitly preclude systems faults. Rather, it requires that the application service, i.e., the functionality required by the end user of the system, is delivered as specified.
- Delivery of service is only required to the extent covered by the specification. The term *justifiably* means that the delivery of service is correct up to the

specification. This point is crucial because otherwise the system developer/designer could never prove the correctness. Moreover, the service is only as good as the requirements: the specification document should reflect the user understanding on how the system should behave. As a consequence, the specification phase has become the most critical phase of the whole system development life cycle.

Although the definition of *Dependability* is ambiguous, some contributions have tried to formalize it. For instance, the authors in [11] identify four attributes to characterize the dependability:

- safety: absence of catastrophic consequences on the user and the environment
- security: preservation of confidentiality and integrity of information
- availability: readiness for correct service
- reliability: continuity of correct service

The above attributes are directly related to vulnerabilities: the primary cause of adversarial attacks or unexpected system behaviours. This demands an effort by design engineers to satisfy security requirements while preserving functional ones. As we will see these two needs are now generally enforced as a conjunct approach during the design steps. The concept of cybersecurity has been changing through time, reflecting the demands from the involved parties: users, companies, organizations, scientific community, etc. Traditionally, cybersecurity has been viewed as protection of resources:

- Confidentiality: prevention of unauthorized disclosure of information
- Integrity: modification or destruction of information

The concept now goes beyond that and embodies also the notion of availability: the whole system as well as its parts (data, software, services, etc.) must be usable whenever they are needed by the authorized users.

Moreover, cybersecurity carries also additional tasks: gather evidence of, prevent or detect illegal activities in cyberspace, from acts of terrorism [12] to articulated web-based investment frauds [13].

Cybersecurity is a very complex task: it is expensive, both in terms of money and time, it has to be comprehensive, it requires certain knowledge, it has to be coordinated with other efforts. Therefore, it is often misinterpreted or incorrectly implemented, making room for threats, vulnerabilities and attacks. For example, industrial control systems are prone to cyberattacks with possible catastrophic consequences, as showed by the Stuxnet worm [14]. In 2015 a group of researchers demonstrated that an attack to control an entire vehicle is already viable [15], while Meltdown and Spectre attacks [16, 17] showed that microarchitectural vulnerabilities can be exploited to retrieve sensitive data from almost the entire user and server computer market.

The literature is packed with surveys and assessments on cybersecurity or state of cybercrime [18, 19].

Securing a computing system is a complex task, there are a plethora of different approaches and methods, but they can be grouped in four general categories:

- the use of procedures to achieve a security requirement
- the integration of the system with additional functions or mechanisms
- the use of assurance techniques
- the use of intrusion detection systems.

Some security requirements prescribe that the user must follow a very specific procedure, i.e., an appropriate behavior when using the system. "*The password must be at least 8 characters long*" is an example of this approach. Enhancing the system security can also be achieved by adding functions or mechanisms (either hardware or software). Examples are hardware-accelerated security chips, access control, authentication mechanisms, and inference control. This solution is expensive though, it requires money for the hardware and time for software development. Combining this approach with some assurance techniques is a possible trade off. Assurance techniques are rigorous methods (either formal or non-formal) to increase security confidence towards the system. For example, formal functional verification is able to prove the system correctness under the system model and its specification.

Nonetheless, no matter how engineers implement security requirements in their systems, it is now generally accepted that cybersecurity can be obtained only

through systematic development, it can not be achieved through disorganized and improvised methods [20].

Historically, security research and development was born in the IT area. Traditional IT security methods use the notions of confidentiality, integrity and availability. They, too, can be applied to CPSs security. However, they do not cover the full spectrum of CPSs security, hence new challenges are open to be solved [21].

Although standard notions of security are quite effective on the software side, they tend to provide little to nothing solutions in the hardware side of CPSs. As a consequence, information security approaches fail to discover unwanted behaviors under attacks that take advantage of hardware vulnerabilities.

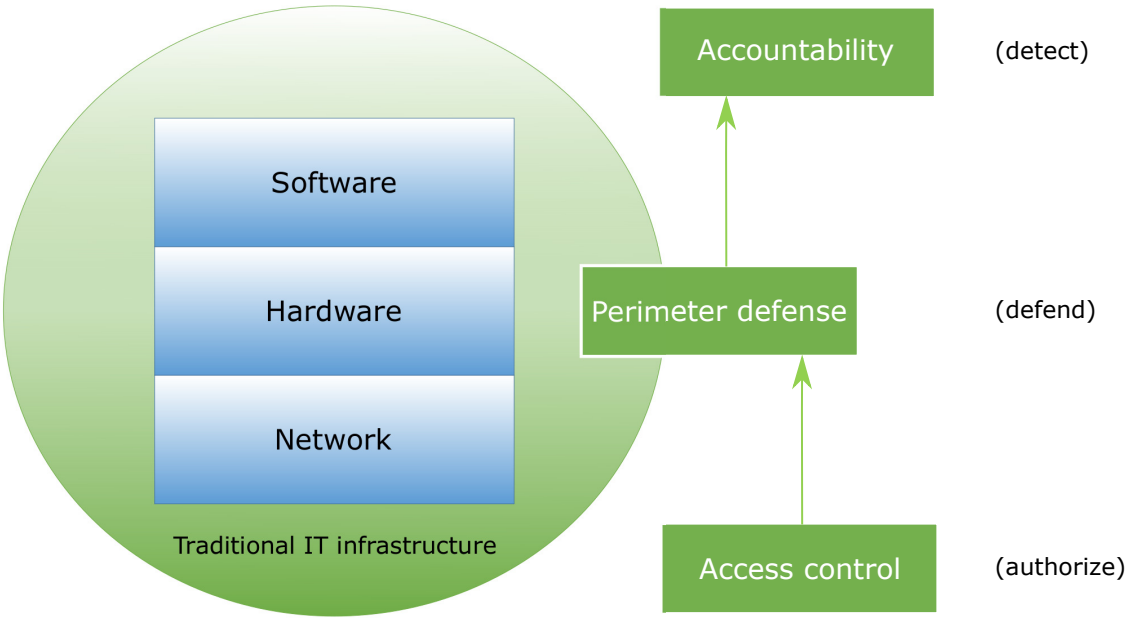

Figure 1.1: Vulnerable components in traditional IT infrastructure and defense strategies

In the IT world the security is mostly depicted as in Figure 1.1. The countermeasures are: accountability, perimeter defense, access control mechanisms. They serve to, respectively, authorise accesses, defend the secure area and detect unwanted behaviors. The scheme is well established and indeed works on most the IT architectures, but emerging threats in the CPS world compromise the trustworthiness of it. The biggest disadvantage of this approach is that, while it is efficient against attack from the *outside*, is useless against *inside* attacks, i.e., attacks that

exploit system faults or design flaws.

And this is the point where formal methods come into play. Recall the four approaches to enhance the security of a system: one of them is to subject the system to rigorous assurance techniques. Among these techniques there are formal methods. None of these methods can provide a full assurance for secure systems, but they increase the security confidence more than other approaches.

The literature offers contributions on formal approaches over these systems: they range from the diverse family of formal languages called Process Calculi [22, 23], Timed Automata [24], Game Theory [25], Theorem proving [26]. Among other formal methods, Model checking is a formal technique used in the verification steps of the design lifecycle. To the best of our knowledge few results have been proposed to model check hardware/software designs: either they target specific systems (e.g., stochastic systems [27]) or require great manual effort in order to translate the original HDL-based model into a specialized algebra or calculus (e.g., Security Process Algebra [28], PROMELA [29]).

Chapter 2

Background

2.1 Security features

Security is a well established practice in Software Development Life Cycle. As mentioned in the previous Chapter, hardware development life cycle approaches often lack of support for security. As we intend to deal with security mainly with both hardware and software, we must first provide a general scheme (or threat model) to identify security features and threats. We will refer to this common lexicon throughout this work. Among other methodologies, STRIDE [30] is one of most common in the software area. STRIDE identifies six security features. Following STRIDE we cover every aspect of security with each category.

2.1.1 Spoofing

Spoofing is declaring a false identity (e.g., process spoofing) or pretending to be someone else (e.g., user spoofing).

Authenticity is what this category deals with.

Examples:

- In computer networks, man-in-the-middle attacks are often carried out with IP spoofing.
- A phishing attack attempts to gain credentials from the users.

- Brute-forcing username/password credentials is one example of identity spoofing.

Typical countermeasures are proper authentication schemes.

2.1.2 Tampering

Tampering is about the malicious modification of data/processes. The modification may happen during the transmission of data, during the storage of data, or on processes.

Integrity is what this category deals with.

Examples:

- Injection attacks on the system processes, e.g., SQL injection.
- An attacker modifies data on the main memory/on secondary memory.
- An attacker alters data in transit by bit-flipping attacks.

Typical countermeasures are:

- Data protection mechanisms
- Validation of inputs and encoding of outputs.
- Minimize dependence to 3rd party libraries/dependencies.
- Integration with static code analysis tools.

2.1.3 Repudiation

Repudiation is about denying that an event has occurred.

Non-repudiation is what this category deals with.

Examples:

- Attackers usually hide their track by erasing log files.
- A destructive action (e.g., deleting an entire database) cannot be associated with a particular person.

Typical countermeasures are audit logging systems.

2.1.4 Information Disclosure

Information Disclosure is about data leaks. The leak may happen during the transmission of data, during the storage of data, or on processes.

Confidentiality is what this category deals with.

Examples:

- An attacker reads sensitive data in a database.
- An attacker retrieves the encryption key.
- An attacker manages to gain confidential information on the system.
- An attacker is able to sniff in transit data over a communication channel.

Typical countermeasures are proper encryption schemes.

2.1.5 Denial of Service

Denial of Service is about making a network resource or a service unavailable to its intended users.

Availability is what this category deals with.

Examples:

- The storage (i.e., disk, drive) is maliciously filled with data.
- An attacker floods the network (e.g., SYN flood attack).
- The encryption routine is continuously called thus preventing normal users to do so.

Prevention is generally not a feasible task for a plethora of reasons. Mitigation is the best tradeoff, for example by monitoring systems with Intrusion Detection Systems (IDS).

2.1.6 Elevation of Privileges

Elevation of Privileges, or Privilege escalation, is about exploiting a misconfiguration, a design flaw or a bug to gain access above one's clearances.

Authorization is what this category deals with.
Example:

- Attackers with limited permissions manage to elevate their clearances by crafting a special request or running a particular sequence of code (e.g., Meltdown [16] and Spectre [17] attacks).
- Elevation to root-level privileges by means of Buffer Overflow or Return Oriented attacks.

Typical countermeasures are:

- Proper access control.
- Security code analysis.
- Security by design.

All the features above are briefly summarized with their corresponding threat in Table 2.1.

Security property	**Security threat**
Authentication	Spoofing
Spoofing	Tampering
Non-repudiation	Repudiation
Confidentiality	Information Disclosure
Availability	Denial of Service
Authorization	Elevation of Privilege

Table 2.1: Security concerns with corresponding security threats suggested by the Security Development Lifecycle

2.2 Formal methods

Formal methods used in computing systems use mathematical techniques to provide frameworks within which we can describe, develop and verify hardware and

software systems. Unlike other approaches, formal methods adopt mathematical proof and rigorous reasoning to ensure "correct" behavior. As systems are becoming larger and more complex, safety and security turn up to be a critical issue, therefore formal methods provide another level of assurance.

Traditionally the verification of "correct" behavior has been carried out by extensive testing and simulation, but these two techniques are limited to finite conclusions. It has been demonstrated that testing is able to provide guarantees only under the conditions where the system does not fail, that is they cannot provide any guarantee about situations outside the test scenario. In contrast, formal methods offer the mathematical guarantee that a certain property is always true. Moreover, conventional simulation approaches yield only partial results quickly but they hardly reach full coverage. Formal verification, on the other hand, is capable to attain completeness, at least in principle [31, 32].

Generally, formal design is divided into a three step process:

Formal Specification In this phase, the designer, using a modeling language, methodically describes the system. A modeling language is a predefined grammar intended to model complex systems by means of primitive types and constructs. This step is analogous to some software engineering techniques developed to translate natural language-defined specifications into formal/-functional requirements. Both approaches help designers to clarify and settle their goals, problems, and solutions.

Verification As previously mentioned, formal methods are distinguished from other specification techniques by their heavy emphasis on provability and correctness. Since formal methods take advantage of rigorously defined logics, the engineer is therefore building a set of properties or theorems about the system. If the property is proven to be true, the corresponding requirement becomes always true, no matter the conditions. The complexity of proving a theorem has driven the verification community to developing automated theorem proving tools.

Implementation The last step is the implementation: translating the formal model into code. This *should* be the last step, but very often designers are forced, by time constraints, to implement the system and only after that

perform the verification.

2.2.1 Benefits Of Formal Models

As already mentioned, formal methods offer the great insurance of provability.

- **Precision**: In contrast to other design approaches, formal methods require very rigorously defined goals and techniques. One of the greatest advantages of formal methods is the elimination of ambiguity. This is particularly relevant for safety critical and security critical systems, where ambiguity can be extremely dangerous.

- **Discipline**: Formal methods require rigor, this forces formal engineers to think thoroughly while designing the system. Specifically, a formal proof of correctness demands a formal specification of requirements. This approach has been demonstrated to be more effective in identifying faulty reasoning than in traditional design [33].

2.2.2 Weaknesses Of Formal Methods

- **Expense**: Formal methods are usually more expensive than traditional approaches. Nonetheless, it is debatable how much this "*more expensive*" is for formal methods. Generally speaking, formal approaches are very costly at the beginning of the project, while becoming cheaper in time. This trend is exactly the opposite for standard development techniques. The expense varies from project to project, so the question arises: *Is this large initial cost worth the effort?*. Or in other words: *Are the benefits of this formal approach greater than its disadvantages?*.

- **Usability**: Modeling languages usually have the great strength to model anything and formal methods are extremely descriptive. Unfortunately, the same qualities require engineers to undergo through hard training and detailed skill acquisition. Moreover, formal methods generally have a steep learning curve.

- **Scalability**: Although there have been great efforts to improve scalability in formal methods, large designs are not quite feasible for most verification tools out there. The bright side is that the research in this field is still very active.

2.3 Formal Specification

A specification is a description of a system and of its desired properties. It is often used as higher level documentation alongside the implementation's documentation. Formal specification, in particular, describes the intended behavior of the system using a formal language, i.e., with mathematically-defined semantics and syntax. The great advantage of such language is that it provides an unambiguous method of description. This process, as already mentioned, helps uncover ambiguity and incompleteness.
The specification can be represented by two broad classes of formalisms:

- Logics: propositional, modal (temporal), first-order predicate, higher-order, etc.

- Automata/language theory: omega automata, finite state, etc.

They tend to have different pros and cons, making each one of those most suitable for a specific use case.

The desired behaviors are called *properties*. They can describe functional, safety or security correctness. Properties are generally divided in two classes:

- Safety property: it asserts that nothing bad happens, e.g., an error condition, or a deadlock state, or that A and B must not have simultaneous access to a shared resource. If false, it can be detected by finite sequences. For instance, confidentiality properties belong to this category.

- Liveness property: it asserts that something good eventually happens, e.g., a progress as in the opposite of starvation, if A is waiting to enter a critical section, it will eventually be allowed to do so. It can only be proved false by infinite sequences. For instance, availability properties belong to this category.

2.3.1 Formal logic

A logic is an organization of a set of symbols (called alphabet), a language over the alphabet to make statements, a set of rules that prescribe the construction of sentences (grammar or syntax rules) and a semantics (interpretations of sentences). Logics share some common characterizing properties:

- Consistency: all theorems can be true at the same time or that do not contradict each other.
- Validity: if the premises are true, it is guaranteed that the conclusions are also true.
- Completeness: every formula can be derived using the formal system, i.e., is a theorem of the system.
- Soundness: if and only if every formula that can be proved in the system is logically valid with respect to the semantics of the system, i.e., if any formula is a theorem of the system, it is true.
- Expressivity: the ability to express concepts in the system.

There are four types of Logic:

- Propositional logic: a formula is expressed by combining atomic propositions and logical connectives, e.g., traditional Boolean algebra
- Modal/temporal logics: modal logic affects the circumstances (the time in temporal logics) in which we deem an assertion to be true
- First-order logic (Predicate logic): uses quantified variables, e.g., *there exists* ($\exists$) and *for all* ($\forall$)
- Higher-order logic: an extension of the first-order logic, includes also quantifiers over sets and functions (predicates)

The application of a logic to verification is as follows:

- The *specification* is expressed as a *formula*
- The *implementation* is expressed as a *formula* or as a *semantic* model

There are two possible cases:

- $Formula \vdash Formula$: Verification of relationships (implications or equivalences) between the implementation and the specification. This is the case of theorem proving.
- $Model \models Formula$: Verification of a semantic relation between a model and a specification formula (property). In this case both model checking and theorem proving can be used.

Finally we express the relations between the implementation (Imp) and the specification (Spec) in the following manner:

- $Imp \equiv Spec$: the implementation is *equivalent* to the specification
- $Imp \implies Spec$: the implementation *logically implies* the specification
- $Imp \models Spec$: the implementation is a *semantic model* in which the specification is true

2.4 Formal verification

We are now going to talk more specifically about Formal Verification (FV). Let us remind some basic concepts.

"*Formal Verification is the process of constructing a proof that a target system will behave in accordance with its specification.*"

In addition, FV has the following features:

- Mathematical reasoning is used in order to prove that the implementation satisfies the specification
- Correctness holds no matter the input values, i.e., the specification is always satisfied
- Formal verification implicitly covers all cases
- Must provide:
  - A formal *specification* (high-level behavior or properties)

– A formal description of the *implementation* (a model, functionally and observationally, equivalent to implementation or design at higher level of abstraction)

Full coverage makes formal verification an excellent alternative able to track down bugs that are not detected by standard verification techniques. Formal verification can even detect bugs earlier in the development process than other techniques. Although reliable and exhaustive, this approach still needs improvements and it is often used in conjunction with standard simulation-based systems. While FV performs *complete* verification of correctness but requires *partial* (abstract) models, simulation is able to run the *complete* model leading to *partial* verification.

Formal verification techniques can be divided in three main classes, as depicted in Figure 2.1, and they are described in the following subsections.

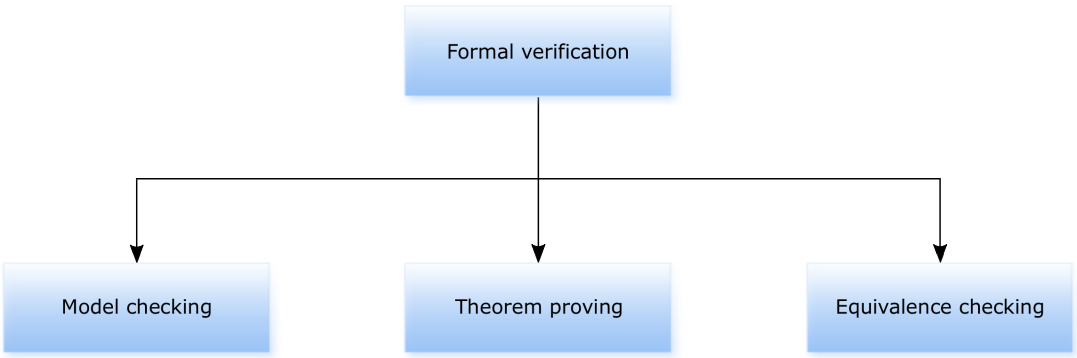

Figure 2.1: Formal verification techniques

2.4.1 Theorem proving

Theorem proving is the process of constructing and verifying a proof using mathematical reasoning. If the theorem is proven it means that the implementation meets design requirements (or specifications). Logic is used for reasoning about the system under consideration, it can express complete proofs of correctness or of absence of failures.

Theorem proving can be split in the following steps:

1. Using a formal mathematical logic, define mathematical definitions to create a model of the system.

2. From the definitions derive the properties that express some system's behaviours.

3. Properties and model are fed into a theorem prover to carry out the proof.

The same steps are depicted in Figure 2.2

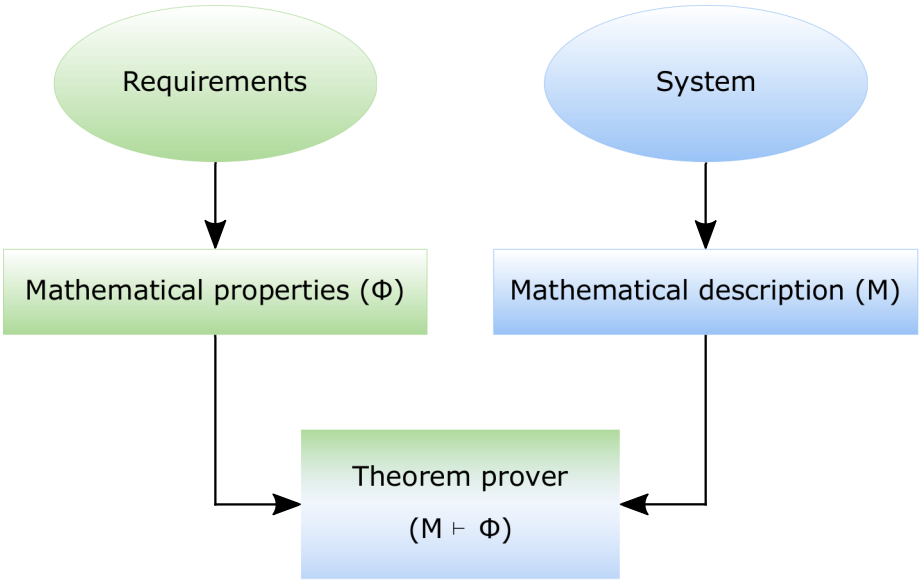

Figure 2.2: The procedure of Theorem proving

2.4.2 Equivalence checking

Equivalence checking is the process of verifying that two designs are functionally the same, that is: *Given two designs, prove that for all possible input stimuli their corresponding outputs are equivalent.* There are two variants of equivalence checking, they both verify that two models are functionally identical:

- At different levels of abstraction.
- At the same level of abstraction but with different implementations.

The process is depicted in Figure 2.3: select a reference model to be a role model and a target model to be the design under test, build the so-called *product machine* composed of the two models and a comparison circuitry for the outputs, provide the input stimuli and run the machine, eventually collect the binary output. If the

output is true, the two models are functionally equivalent, otherwise they differ and do not have the same behavior.

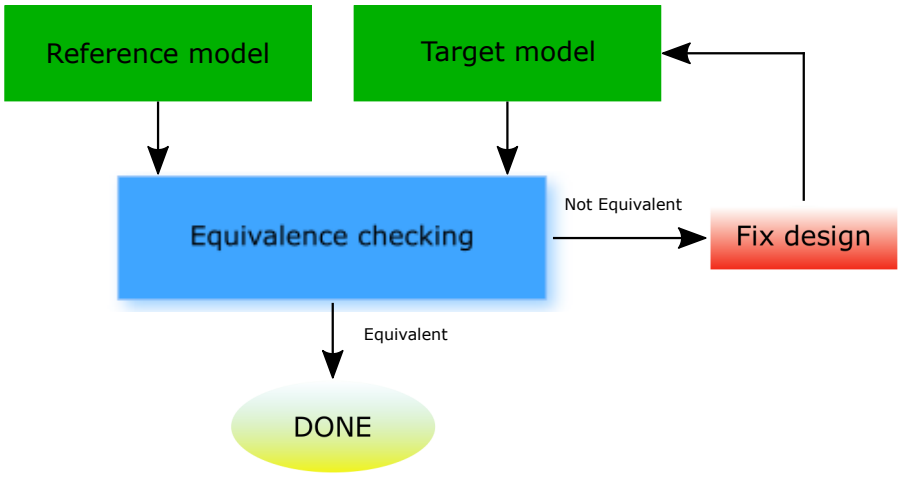

Figure 2.3: The procedure of Equivalence checking

2.5 Model checking

Model checking, also known as Formal property verification, is a state-based approach to formal verification. It is used to verify finite state concurrent systems. A big advantage of model checking is that the verification process can be automated through a proper model checking tool.

The essential idea behind model checking is depicted in Fig. 2.4:

1. Modeling: describe the system with a formal language. The design is usually modelled as a set of states and a set of transitions that define how the system evolves from a particular state to another in response to some stimuli. This step produces a formal system model M.

2. Specification: define the desired properties using a proper specification language (usually the same as in the previous step). This step produces a property ϕ or a set of properties.

3. Verification: M and ϕ are fed to a model checking tool (or *model checker*) that runs automatically. If the answer of the model checker is positive (true) then the model is proven to satisfy the property. Otherwise, the model checker

generates a counter-example, i.e., the stimuli that violate the property. The counter-example is then used to simulate the model M and find the unwanted behavior.

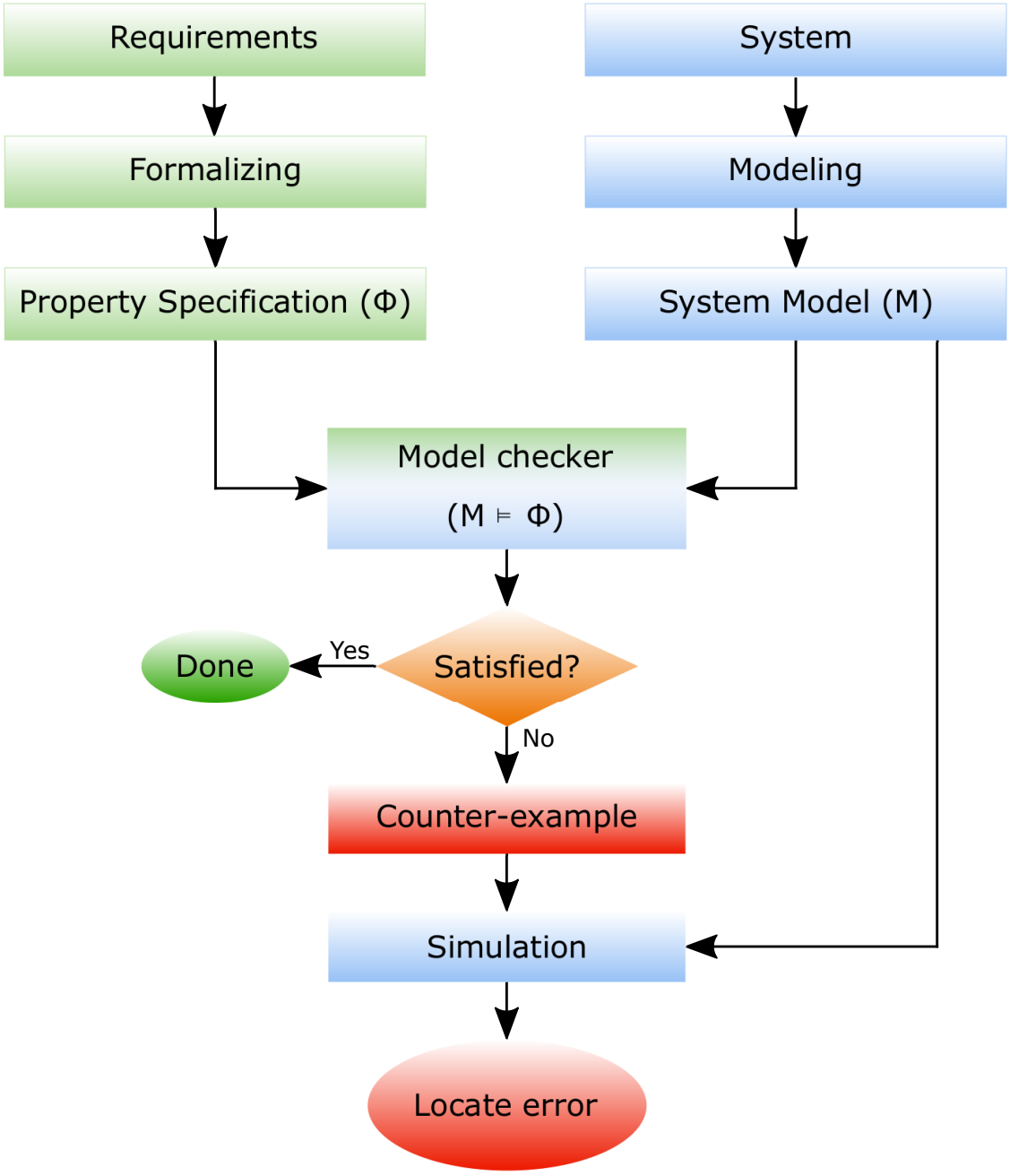

Figure 2.4: The procedure of Model Checking

Research over model checkers have brought significant enhancements in terms

of efficiency, automatism, supported languages, features, etc. A standard model checker nowadays is able to, given sufficient resources, terminate with a binary answer in reasonable time. There are some caveat though. Since the model checking tools usually perform an exhaustive search over the finite state space of the system, large designs represent a serious problem for model checkers for a problem called *state space explosion* which we are going to talk about below. The theory of computation provides limitations on the decidability of an algorithm. Notably, an important undecidable problem is the *halting problem*: it cannot exist an algorithm that decides if an arbitrary program eventually terminates. As a consequence, there are restrictions on what can be verified automatically.

2.5.1 Modeling

The first step is the conversion of the design into an abstract formal model accepted by a model checker. As the state space could be very large, the model should be as small as possible in order to minimize the size of the state space search. This step is sometimes performed automatically, for example for hardware systems. In the majority of cases, however, human effort and guidance is necessary. The input design could vary for its form and its nature, it could be hardware or software, it could represent a gate level description, a RTL description, a behavioral model, or even high level specifications. The choice of the formal language to describe is particularly relevant for efficiency, support, expressivity, etc. In some cases, even an ad-hoc and fine-tailored logic is used. The property specification (described in the next subsection) is usually performed with the same logic or language. Despite this variety, a generic design is commonly seen as a set of concurrent (i.e., interacting) systems. Each system is represented as a *Finite State Machine* (FSM) and has a finite number of states. Certainly there are systems with infinite states and the literature is plenty of contributions on this topic but they are out of scope for this work. When given an input, the FSM can change from one state to another. This is called *transition*. The set of states, the initial state and set of transitions characterize the FSM. Following this approach, the whole system can be seen as a FSM, which is the composition of the FSMs associated with each component. The process of modelization therefore consists of creating a complete FSM description of the system. Automata theory provides a plethora of methods to describe finite

state automata: Pushdown automata (PDA), Linear bounded automata (LBA), Turing machines, Buchi automata, etc. Model checking methods generally model the behavior of the system by means of Kripke structures. A Kripke structure is a variation of a transition system and is essentially a node-labeled graph. System states are represented by the nodes of the graph and each node is labeled with binary information (i.e., atomic propositions described with the chosen logic) at that state. State transitions, that is the evolution of the system, are represented by the edges of the graph.

2.5.2 Specification

This process involves the following question: *What properties must the system design satisfy?* This is the step where we define the key properties to be verified: they could describe a particular system's behavior or model a certain feature, e.g., the confidentiality of a secret key. Generally, the specification is expressed with the same formalism as the previous step. The class of temporal logics has been proven to be very useful because of their inherent capability to describe how the system evolves over time. Although the meaning of a statement is constant in time, its truth value may vary in time. Temporal logic always has the ability to reason about a timeline. In particular, time is implicitly built within the logic, without the need for an explicit definition. A common classification for temporal logics is between linear time logics (e.g., LTL) and branching time logics (e.g., CTL and CTL*). The difference is whether time is assumed to have a linear or branching structure. In the last years, though, new types of temporal logics have been proposed to respond to specific problems. Key examples are HyperLTL and HyperCTL* [34]: they have been proposed to provide support for a new class of properties, namely *hyperproperties* [35]. Overall, this process is particularly time consuming because of some crucial issues: for example, consistency (i.e., whether the given specification is consistent with higher level requirements) or completeness (i.e., whether the given specification cover all the behaviors of the system).

2.5.3 Verification

In theory, the verification step is completely automatic. In practice, however, human effort is often needed. For instance, the analysis of the verification results can provide useful insights on the system specification and implementation. If the model does not satisfy the given property, the model checker generates a counter-example, i.e., an error trace that can be used to simulate the system. The simulation helps the engineer to track down where the error occurred. Once the bug is found, the engineer modifies the model and repeats the verification process. The verification process can also lead to false negatives and false positives, that can be introduced by:

- inconsistent specification
- incorrect formalization of the specification
- incorrect modeling of the system

The verification step could fail to terminate normally or not terminate at all. This is due to the size of the model, which leads to the state space explosion. The major limitation of model checking is the *state explosion problem*. The problem occurs when the whole model is computationally too big, that is composed of too many components or few large components. The composition of two FSMs, representing two concurrent components, is performed by taking the Cartesian product of the corresponding state spaces. This leads to an exponential grow, in the number of concurrent processes involved, of the overall state space of the system. An exhaustive search over this large state space may be unfeasible even for well-optimized linear-time algorithms. The verification community, in order to cope with this problem, has offered two approaches:

- Symbolic algorithms
- Partial order reduction

2.6 Modelling Concurrent systems

2.6.1 Transition systems

Transition systems are usually used to describe and model system behaviors. A transition system are represented as directed graphs: edges represent transitions (i.e., state changes), nodes model states. A state provide information on the system at a certain moment of time. The state for a traffic light, for example, is the current color of light. Likewise, the state for a sequential computer program is represented by the set of variables, their current values and the value of program counter. In the hardware field, a state is typically the set of registers and their current values. Transitions describe the evolution from state to another. A transition for the traffic light is the change of color. A transition for a program is the execution of an instruction and the corresponding change of variable values as well as the program counter. For hardware circuits, a transition is the change of registers and signals.

Regardless of the actual formalism employed for their specification, the (operational) semantics of reactive systems is fully described by the general framework provided by transition systems.

DEFINITION 1. *A labeled transition system is a tuple* $\mathcal{T} = (S, I, E, \delta)$ *composed of:*

- *a set* S *of states*
- *a subset* $I \subseteq S$ *of initial states*
- *a set* E *of (action) labels*
- *a transition relation* $\delta \subseteq S \times E \times S$

A run of $\mathcal{T}$ *is an* ω*-sequence* $\rho = s_0 \overset{e_0}{\Rightarrow} s_1 \overset{e_1}{\Rightarrow} \dots$ *of states* $s_i \in S$ *and labels* $e_i \in E$ *such that* $s_0 \in I$ *is an initial state and* $(s_i, e_i, s_{i+1}) \in \delta$ *is a transition for all* $i \in N$*. A state* $s \in S$ *is called reachable in* $\mathcal{T}$ *if there exists some run* $s_0 \overset{e_0}{\Rightarrow} s_1 \overset{e_1}{\Rightarrow} \dots$ *of* $\mathcal{T}$ *such that* $s_n = s$ *for some* $n \in N$*.*

The goal of verification algorithms is to decide whether a transition system satisfies a given property. Properties represent system behaviors that verification

engineers want to check. Some elementary binary (i.e., true or false) propositions form a property. This idea is behind Kripke structures, a variation of the transition system. Kripke structures extend a transition system by adding an interpretation of atomic propositions over states.

DEFINITION 2. *Let AP be a set of atomic propositions. A Kripke structure $K = (S, I, E, \delta, \lambda)$ extends a transition system by a mapping $\lambda : S \to 2^{AP}$ that associates with every state $s \in S$ the set of propositions true at state s. The runs of a Kripke structure are just the runs of its underlying transition system.*

The labeling function λ maps any state s to a set $\lambda(s) \in 2^{AP}$ of atomic propositions, this mean we can evaluate formulae of propositional logic built from the propositions in AP.

$\lambda(s)$ intuitively stands for exactly those atomic propositions $a \in AP$ which are satisfied by state s. Given that Φ is a propositional logic formula, then s satisfies the formula Φ if the evaluation induced by $\lambda(s)$ makes the formula Φ true; that is: $s \models \Phi$ *iff* $\lambda(s) \models \Phi$.

2.6.2 System invariants

Safety properties usually describe that "something bad should never happen". For example, a computer program should never go in deadlock. Or at most one process is in its critical section at any time. This particular type of properties states that a condition must always be true, that is $s \models P$ holds for all reachable system states s. We call this a *system invariant* or simply *invariant*. Invariants are, as we showed, a special case of safety properties.

DEFINITION 3. *A property P_{inv} over AP is an invariant if there is a propositional logic formula Φ over AP such that*

$$P_{inv} = \left\{ A_0 A_1 A_2 \cdots \in (2^{AP})^\omega \mid \forall j \geq 0.\ A_j \models \Phi \right\}.$$

Φ is called an invariant condition of P_{inv}.

Model checking algorithms that decide whether a finite Kripke structure $\mathcal{K}$ satisfies an invariant property P are conceptually easy: they can simply enumerate

the reachable system states and check that P holds true in every one of them. Termination is guaranteed by the finiteness of $\mathcal{K}$.

2.7 Linear Time Temporal Logic

Given a Kripke structure $\mathcal{K}$, we are sometimes interested in some properties, such as:

- Is it always possible to reach an initial state? In other words, can the system be re-initialized?
- Does $\mathcal{K}$ contain states that do not satisfy an invariant?
- Does $\mathcal{K}$ have "bad" states, such as deadlock states where only the τ action is enabled?
- Does the system have livelocks (i.e., when the system is not in deadlock but some process is)? In other words, are there traces of $\mathcal{K}$ such that, after some time, a certain action is never executed or a "good" state is never reached?

Temporal logic provides a language in which such properties can be formulated.

Temporal logics can be categorized for their syntactic features, or for their semantic structures. In particular, the underlying nature of time in temporal logics can be either linear or branching. If at each moment of time there is only one possible successor we are talking about linear time logic. On the contrary, if we have a tree-like time structure we say it is a branching time logic. This section describes Linear Temporal Logic (LTL).

Linear temporal logic has been introduced by Pnueli [36] as a tool for the specification and verification of reactive systems.

The syntax of LTL-formulae is made up by the following basic ingredients:

- atomic propositions $a \in AP$ (that corresponds to state label a in a transition system)
- Boolean connectives such as conjunction $\wedge$ and negation $\neg$

- basic temporal modalities $\bigcirc$ (pronounced "next") and U (pronounced "until")

Usually, the atomic propositions are assertions over control variables (e.g., the program counter or program variables such as "$x \geq y$" or "$x > 5$"). The $\bigcirc$-modality is a unary prefix operator and takes a LTL formula as argument. Formula $\bigcirc\varphi$ holds at the current moment, if φ holds in the next "step". The U-modality is a binary infix operator and takes two LTL formulae as arguments. Formula $\varphi_1 \mathsf{U} \varphi_2$ holds at the current moment, if there is some future moment for which φ_2 holds and φ_1 holds at all moments until that future moment. LTL formulae over the set AP of atomic proposition are formed according to the following grammar:

$$\varphi ::= true \;\Big|\; a \;\Big|\; \varphi_1 \wedge \varphi_2 \;\Big|\; \neg\varphi \;\Big|\; \bigcirc \varphi \;\Big|\; \varphi_1 \mathsf{U} \varphi_2$$

where $a \in AP$.

The U-modality lets us derive the temporal modalities $\Box$ ("always", from now on forever) and $\Diamond$ ("eventually", sometimes in the future) as follows:

$$\Box\varphi \stackrel{\texttt{def}}{=} \neg\Diamond\neg\varphi \qquad\qquad \Diamond\varphi \stackrel{\texttt{def}}{=} true\,\mathsf{U}\,\varphi$$

$\Box\varphi$ is satisfied if and only if it is not the case that eventually $\neg\varphi$ holds. This is equivalent to the fact that φ holds from now on forever. $\Diamond\varphi$ ensures that φ will be true eventually in the future. By combining the temporal modalities $\Diamond$ and $\Box$, new temporal modalities are obtained as follows:

$\Box\Diamond\varphi$ "infinitely often φ"

$\Diamond\Box\varphi$ "eventually forever φ"

2.8 Branching Time Temporal Logic

Instead of relying on a linear notion of time (i.e., an infinite sequence of states), branching time temporal logic uses a branching notion of time (i.e., an infinite tree of states). Branching time is generally represented as an infinite tree of states because at each moment there may be several different possible futures. Branching time logic allows us to express properties of *some* or *all* computations starting

from a particular state. The support is provided by new path quantifiers, such as an universal path quantifier (denoted $\forall$) and an existential path quantifier (denoted $\exists$). For example, the property $\exists\Diamond\Phi$ indicates that there exists a computation along which $\Diamond\Phi$ holds. That is, it states that there is at least one possible computation in which a state that satisfies Φ is eventually reached.

A notable branching time logic used in formal verification is Computation Tree Logic (CTL) and it has been introduced by Clarke and Emerson [37]. CTL formulae are classified into state and path formulae. The former are assertions about the atomic propositions in the states and their branching structure, while path formulae express temporal properties of paths. Given a set AP of atomic propositions, CTL *state formulae* are generated by the following grammar:

$$\Phi ::= \ true \ \Big| \ a \ \Big| \ \Phi_1 \wedge \Phi_2 \ \Big| \ \neg\Phi \ \Big| \ \exists\varphi \ \Big| \ \forall\varphi$$

where $a \in AP$ and φ is a path formula. CTL path formulae are generated by the following grammar:

$$\varphi ::= \ \bigcirc\Phi \ \Big| \ \Phi_1 \,\mathsf{U}\, \Phi_2$$

where Φ, Φ_1, Φ_2 are state formulae.

2.9 CTL*

CTL* is an extension of CTL and it is a superset of both CTL and LTL. It allows path quantifiers $\exists$ and $\forall$ to be arbitrarily nested with linear temporal operators such as $\Box$ and U. As for CTL, CTL* formulae are classified into state and path formulae. Given a set AP of atomic propositions, CTL* *state formulae* are generated by the following grammar:

$$\Phi \ ::= \ true \ \Big| \ a \ \Big| \ \Phi_1 \ \wedge \ \Phi_2 \ \Big| \ \neg\Phi \ \Big| \ \exists\varphi$$

where $a \in AP$ and φ is a path formula.
CTL* *path formulae* are generated by the following grammar:

$$\varphi ::= \Phi \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \mathsf{U} \varphi_2$$

where Φ is a state formula, and φ, φ_1, and φ_2 are path formulae.

2.10 ω-automata

Transition systems, even if they are finite, usually generate an infinite set of runs, each of which is an infinite sequence of states. ω-automata are usually used to specify behavior of systems that are not expected to terminate, such as control systems, operating systems, hardware systems. For example, we would like to verify, for such systems, a property that states "for every action, there eventually be a counter-action". A finite sequence of states cannot prove that the property is satisfied. This theory has been given notable contributions over the years (Büchi [38], Muller [39], Rabin [40], etc.) but we limit the discussion to Büchi automata.

DEFINITION 4. *A Büchi automaton $\mathcal{B} = (Q, Q_0, \Sigma, \delta, F)$ is given by:*

- *a finite set Q of states*
- *a set $Q_0 \subseteq Q$ of initial states*
- *Σ is an alphabet*
- *a transition function $\delta : Q \times \Sigma \to 2^Q$*
- *a set $F \subseteq Q$ of accepting (or final) states.*

A run of $\mathcal{B}$ over a sequence $\sigma = a_0 a_1 \ldots$ where $a_i \in \Sigma$ is a sequence $\rho = q_0 q_1 \ldots$ of states $q_i \in Q$ such that $q_0 \in Q_0$ is an initial location and $(q_i, a_i, q_{i+1}) \in \delta$ holds for all $i \in N$. The run ρ is accepting if it contains an infinite number of states $q_k \in F$.

Büchi automata, as we have defined them, operate on ω-sequences of subsets of Σ, and this is in close correspondence with the interpretation of (linear time) temporal logic over Kripke structures. Indeed, any run $s_0 \overset{e_0}{\Rightarrow} s_1 \overset{e_1}{\Rightarrow} \ldots$ of a Kripke

structure $\mathcal{K}$ can be identified with the corresponding sequence $\lambda(q_0)\lambda(q_1)\ldots$ where λ is the propositional valuation of states of $\mathcal{K}$. In this sense, propositional temporal logic formulae and Büchi automata operate over the same class of structures.

2.11 Symbolic Model Checking

The notions we have seen so far (transitions systems, Kripke structures, Büchi automata, etc.) are intrinsically based on states, thus model checking algorithms operate on states too. Efficient implementations of data structures manipulating states are therefore needed. Explicit enumeration (i.e., brute force) is not a viable approach since it only allows few millions of states. An efficient alternative that has been proposed is the symbolic representation of states, since this method would allow compact data structures and therefore more efficient algorithms that operate on them. The most prominent algorithms depend on a binary encoding of the states. Also switching functions have been targeted for performance optimization: ordered *binary decision diagrams* (BDD) are special structures designed to make them compact and efficient.

Originally proposed by Bryant [41], ordered BDDs perform a compactification of binary decision trees. The basic idea is to prune redundant fragments of a binary decision tree. For example, subtrees with all leaf nodes having the same value (i.e., constant subtrees) can be collapsed into a single node. Same for isomorphic subtrees.

BDDs are extensively used in the context of SAT-based model checking where the model-checking problem is reduced to the satisfiability problem for propositional formulae (SAT). BDDs have represented a technological breakthrough for the implementation of model checking algorithms. A major benefit of BDDs is that they provide canonical representation of sets, and therefore set equality can be decided by simple pointer comparison. Moreover, Boolean operations can be performed in polynomial time.

2.12 Bounded Model Checking

Even though BDDs are a powerful data structure used for the representation of Boolean functions, other techniques can be quite useful, especially when a complete verification is not feasible. SAT algorithms, for example, decide the satisfiability of formulae of (non-temporal) propositional logic. This approach is not based on canonical representations of Boolean functions, ruling out the calculation of fixed points. Bounded Model Checking (BMC) [42] algorithms attempt to find an execution of finite length that violates the property of interest. The maximum size of a potential counter-example can (at least in theory) be determined from the size of the model and the formula to be verified. For example, an invariant holds for a Kripke structure if it is true for all prefixes of runs whose length is at most that of the longest loop in the graph of $\mathcal{K}$ (called the diameter of $\mathcal{K}$).

Bounded Model Checking (BMC) [42] is an iterative search for counterexamples to prove false a desired property p up to a given bound k . The bound k greatly limits the state space size so that it can be used in some cases where other model checking techniques fail. On the contrary, the bound k has the disadvantage to make this approach not complete, that is BMC cannot prove the absence of errors. After a run with a certain k, if no counterexample is given then the user increases the bound until either some pre-known upper bound is reached, the problem becomes intractable or a bug is found.

The basic idea of BMC is to express all state paths of length k by unrolling k times a system transition relation T.

$$T^k(X^{0\ldots k}) = \bigwedge_{i=0}^{k-1} T(X^i, X^{i+1}) \tag{2.1}$$

Although BMC has become an effective approach in the verification process, even with large models, it can only guarantee the correctness of a property until a given bound, i.e., the verification method is not complete. This is where Unbounded Model Checking (UMC) comes into play. The main difference between BMC and UMC is scalability. Support for UMC has been improved over the years, with a number of techniques proposed. BDDs are now an established major approach for UMC and BDD-based tools are very efficient, often with large designs, but there

are other contributions. For instance, the ability to check reachability fix-points is crucial for UMC. Property Directed Reachability (PDR) aims to prove completeness for the circuit unrolling or backward, forward or mixed reachability analysis.

The industry nowadays typically uses both BMC and UMC as complementary methods. A typical approach is to use BMC first to find shallow bugs very quickly, the bound is incremented until the problem becomes intractable, then UMC is applied to prove that the property is correct. In other cases BMC and UMC run in parallel, as the first technique finds a solution, the other is aborted.

Chapter 3

Secure Embedded Architectures: Taint Properties Verification

The following chapter has been published as peer-reviewed article in [43].

3.1 Introduction

A large number of assorted embedded systems nowadays constitute a rising trend in industry, forming the so-called Internet of Things (IoT). This direction will lead, analysts say, to billions of connected devices that enable new functionalities and services. The applications are everywhere: self-organizing devices connected in a common network (e.g., robot swarms), connected smart environments, industrial control systems. Most of these devices do not operate alone, their work is coordinated and parallel, so that the key feature is the interconnection between them. Whenever they send or receive sensitive or critical information using public networks or communications channels accessible to potential attackers, they should ideally provide basic security functions such as data confidentiality, data integrity, and user authentication. These basic security functions are the assumptions that underlie the paradigm of *secure communications*. Nowadays, most of the embedded systems use this paradigm in automotive, sensing, medical, financial, military and many other applications [44]. As a consequence, providing a proper security verification is a crucial task in this kind of environments, not only at the device level but also from the networking point of view, thus protecting them from a wide

range of attacks [45, 46, 47, 48]. However, the major research focus is on verifying only the correctness of encryption algorithms and their implementation in software and hardware, neglecting the stream of data (i.e., information flow). Admittedly, our research has been guided by the following questions:

- Is there a way to *read* sensitive data from a specific location?
- Is there a way to *modify* sensitive data?
- Are there failure states that would compromise the security of the system?

Taking these questions as a starting base, we chose to address them with a specific information flow approach in security.

In [49] Taint properties are taken as a role model for expressing many security requirements that involve information flow and access control. A taint is a malicious seed that is injected into source, flows through the system until it reaches its sink. It is easy to notice that this is a natural way to capture confidentiality requirements as we will in Section 3.7. For instance, confidentiality can be verified by setting a hardware secret as the source and an untrusted identity, e.g., a user program location, as the destination. In a similar way if we set an untrusted identity as the source and a security-sensitive signal as the destination we are verifying integrity [50].

This chapter introduces a formal methodology to express generic security requirements as information flow properties in terms of Taint Properties. In details, our contributions are the following:

- we perform abstraction and refinement steps on two secure embedded architectures: SMART [51] and SANCUS [52]. They are two prominent representatives of Remote Attestation: a procedure to attest (i.e., check) the current internal state of an untrusted remote embedded device (*Prover*) by a trusted entity (*Verifier*) [53, 54]. SMART is a simple architecture that provides support for dynamic root of trust in a remote embedded device. It requires small changes and little cost as it is based on hardware/software co-design. SANCUS is a comprehensive framework for Remote Attestation and message authentication. It is module-based and it provides also a networking scheme for communications. It works with clusters (i.e., sets of microprocessor-based systems).

- we show how to express security requirements, such as confidentiality, integrity, key secrecy, isolation, etc., as Taint properties.
- we define a portfolio of Taint properties for the selected abstract models.
- we perform formal verification on the set of Taint properties and we present experimental results supporting our claim

Section 3.2 provides some background knowledge on the topics we use in this work. Section 3.3 recalls related works and compares them with our approach. Section 3.6 introduces a methodology to perform abstraction to extract an abstract model of the two selected architectures. Section 3.7 presents a set of Taint Properties and we provide a formal analysis for security requirements. We also show how to use them with a standard model checker. In Section 3.8 we offer experimental results that support our claim. Finally, Section 3.9 concludes the chapter with summarizing conclusions along with some remarks on future work.

3.2 Background

3.2.1 Trusted Computing Base and root of trust

There is plenty of empirical experimental evidence that conventional embedded systems are not secure. This simple assertion has become evident in the past and is nowadays an universally accepted truth. The ubiquity of flaws and secure-related issues makes this a pressing matter. Furthermore, reconciling **after** discovery of the issue or even after the public disclosure of an attack has been shown an inadequate approach to providing truly secure systems. Though there have been plenty of improvements over the years in terms of efficiency for verification methods, both formal and non-formal, to date there is no technique which is proved to be complete and sound for *every* CPS. Most of them, indeed, are not able to give the upmost assurance that all security vulnerabilities have been removed from the system. Therefore, the only sound approach to this end is deemed to be the so called *security by design*: security must be embedded into the system in the design phase. In other words, the fundamental proof that a system is secure must be focused, **first**, on the security analysis of its design and, eventually, on its implementation. Hence, a secure computer system must involve mechanisms that are sufficient

to guarantee its security in all circumstances, and it must be possible to provide compelling evidence that those mechanisms are entirely adequate to their task. Such *trustworthy computing* or, as it widely known, Trusted Computing requires a Trusted Computing Base (TCB), the core set of functionality that is assumed to be secure, to provide secure primitives to the system. The TCB typically comprises hardware and software components, e.g., hardware accelerated cryptographic engines, software libraries, secure OS services, etc. During an execution of the system, the chain of TCB involved in the computation forms the root of trust, a way to establish trust in the execution environment. So only an execution environment equipped with a root of trust is really "real" trusted. Efforts have been made to regulate this approach and led the Trusted Computing Group (TCG) to produce a standard, later accepted by ISO, In particular, TCG defines a Trusted Platform Module (TPM) as a specialized hardware with security enforced capabilities (e.g., encryption/decryption) and provides the foundation for the host root of trust. For further details, refer to the corresponding ISO standard [55].

3.2.2 Remote attestation

Connected to the TCB and root of trust concepts, is remote attestation: a procedure to attest (i.e., check) the current internal state of an untrusted remote embedded device (*Prover*) by a trusted entity (*Verifier*) [53, 54]. In general, remote attestation is used to establish a root of trust between Prover and Verifier and can help to build other security services such as secure update [56], secure boot [57] and secure erasure [58]. Many existing attestation proposals are aimed at specific use cases. Indeed, many definitions of attestation primarily focus on defining a set of particular properties [59, 60]. For example, attestation might be understood as the process of transmitting a sequence of hashes and the corresponding signature [61]. In Microsoft NGSCB [62], it refers to identification and authentication of known code via digital signatures. Copilot [63] makes use of direct hashes of kernel memory. Authors in [64] propose a composable architecture based on predefined attestation principles. They also provide a quite extensive survey on existing attestation approaches. Recently, a fully verified RA architecture has been proposed by

Nunes et al. [65]. It is, to the best of our knowledge, the first formal verification-oriented approach targeting a remote attestation architecture and the first one to be implemented as a HW/SW co-design.

Historically attestation schemes belong to two opposite classes: *hardware*- and *software*-based techniques. The former can provide strong guarantees in terms of security because they involve specialized hardware such as cryptographic accelerators or security co-processors. This approach, because of its high cost, is often not viable for low-end devices though. On the other hand, software-based techniques tend not to be complete, i.e., they are unable to cover all possible cases, or involve restrictive assumptions on adversarial capabilities. As a consequence, low-end embedded devices tend to resort to a particularly promising hybrid approach, implemented with HW/SW co-design. Software and hardware are designed together to meet not only the functional and performance requirements but also security goals [66]. In the following we present two remote attestation architectures specifically designed for cost-effective embedded devices: SMART and SANCUS.

3.2.3 SMART

SMART [51] introduces a light-weight architecture, designed for low-end embedded systems, for establishing dynamic root of trust. SMART is based on a hardware-software co-design approach: two major advantages are that it requires minimal hardware changes and it assumes few limitations on attacker's capabilities. The attestation model is the following: before, during and after the execution of the attestation routine, SMART assumes that the attacker has complete control over the system. In particular, the attacker can modify any part of the software inside the Prover that is not explicitly protected by the hardware. In addition, the attacker has also the chance to manipulate all connections coming from and to other devices. Prover and Verifier share a secret K which is memory protected. The only assumption is that the hardware components must not be tampered and the attestation routine cannot be interrupted.

At the beginning of SMART (3.1), Verifier sends a set of parameters to Prover: attestation boundaries a and b, address x where control returns if control flag x_{flag} is set, a nonce n to prevent replay attacks. A cryptographic checksum C of Prover's

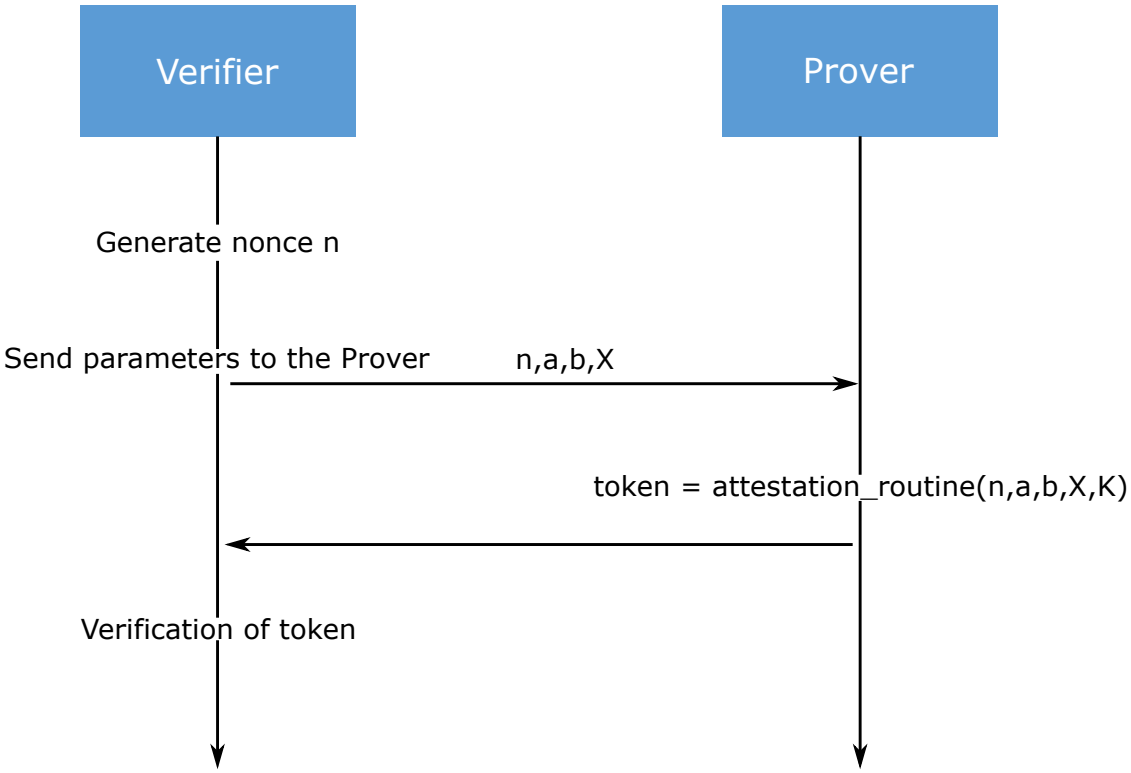

Figure 3.1: An example of Remote Attestation protocol

region [a,b] is computed by a ROM-resident protected code segment. The checksum is an HMAC keyed with the secret K. The control passes to x. After execution of code starting at x, Prover sends the checksum C to Verifier. The latter verifies the correctness of C using the same parameters and the key K. Secrecy of K is achieved by means of secure hardware storage. The MCU restricts access to K so that only (trusted and immutable) ROM code is allowed to use it. Secure access to K is implemented by modifying the memory bus controller as follows:

1. Only the instructions stored in a trusted memory region (e.g., ROM code) are allowed to access the secret K
2. An instruction is allowed to access the trusted region only from its initial address and it can leave only from its last address.

This translates to: if the current and the previous Program Counter (PC) values both point to the trusted region then access to K is granted. If the memory access control detects an invalid memory access, the processor is promptly reset. The

attestation code enforces a memory cleanup whenever there is a memory violation. However, if the cleanup is interrupted by a hardware reset, this might lead to sensitive data leakage. Therefore, a full memory cleanup is enforced upon each boot.

In addition, when SMART (i.e., the attestation code) is invoked, interrupts are disabled. After the execution of SMART, if the control passes to the attested code (i.e., x_{flag} is set), interrupts remain disabled. This is to ensure that the code to attest is actually executed, thus preventing Time-of-check-to-time-of-use (TOCTTOU) attacks.

The authors state that this scheme guarantees that, even though the whole Prover system is tampered (except SMART ROM code), the attested code is eventually executed. Summarizing, SMART has three security objectives based upon successful completion of the attestation protocol:

- Prover Authentication: Verifier attains entity authentication from Prover
- External Verification: Verifier verifies that memory segment [a; b] on Prover contains the expected content
- Guaranteed Execution: Verifier obtains the guarantee that code at location x has been indeed executed by Prover.

3.2.4 SANCUS

SANCUS [52] is a module-based remote attestation architecture. The architecture, as for SMART, has been designed to target small low-end embedded devices, that usually are being deployed in clusters. A cluster is a set of microprocessor-based systems referred to as nodes N_i. An Infrastructure Provider (IP) controls and supervises these nodes. IP and N_i share a fixed and hardware-etched secret K_N. Service Providers (SP_i) can deploy software modules (SM_i) into nodes. A unique public identifier, which is used to derive a key $K_{N,SP} = kdf(K_N, SP)$, is assigned to each SP. Communications between SP and nodes are secured with this key. *kdf* is a generic *key derivation function*, whose details depend on the implementation. These functions are usually realised in hardware, in order to achieve a zero-software TCB for security and performance reasons. Modules are just binary files with two required parts: a protected code section and a protected data section,

respectively *text section* and *data section*. Besides these two mandatory sections, a module is not limited to a certain number of unprotected sections. The limitations of two protected memory regions is enforced to limit the amount of protected memory, that is to keep the trusted regions as small as possible. Each module is uniquely identified by the content of protected sections and their layout (i.e., entry and exit points). This identity, referred to as SM, is then used to compute another key $K_{N,SP,SM} = kdf(K_{N,SP}, SM)$. This symmetric key is specific to the module SM loaded on node N by provider SP. Software module's identity SM and its derived key are stored in a protected area storage, as shown in Figure 3.2. Sancus ensures the keys can never leave the protected storage area by only making them available to software in indirect ways through new processor instructions.

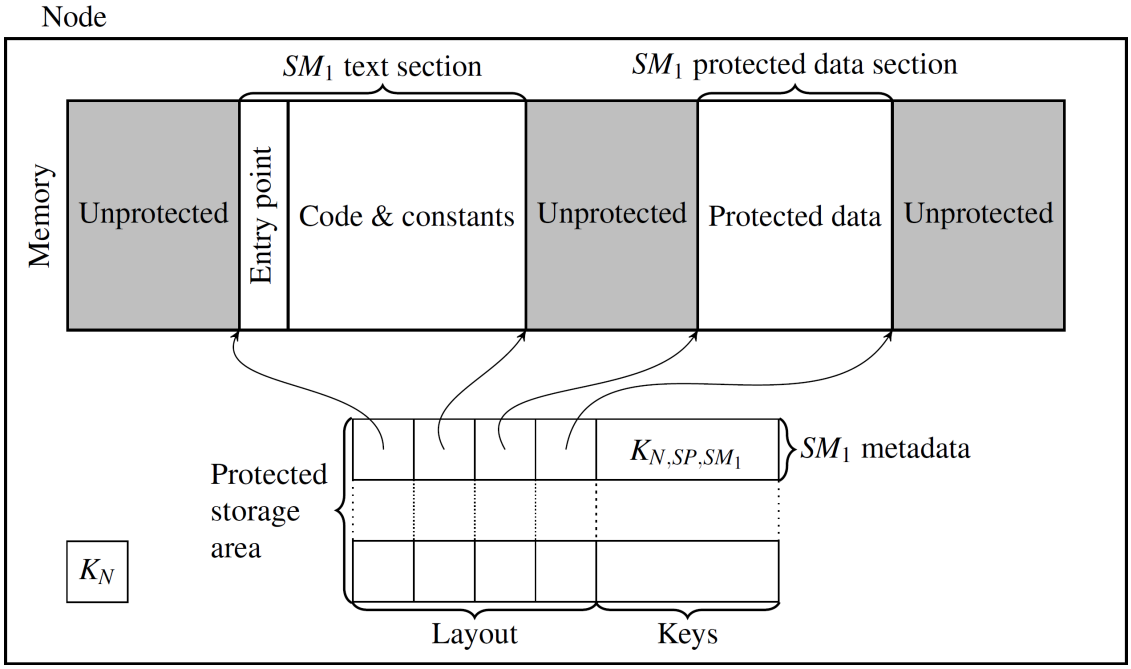

Figure 3.2: Schematic of a SANCUS node with a software module loaded.

A software provider SP would perform Remote Attestation, using module key $K_{N,SP,SM}$, to check that the correct software module SM is running on an expected node N. For example, the authentication proof might be very useful when an SM provides functionalities for another protected module: remote attestation can be used by the user module to check whether the provider module has been manipulated.

Sancus architecture introduces additional CPU instructions that can be used to setup trusted software modules at runtime. For this purpose, multiple memory protection regions are implemented, each containing a code and data section. An extended processor instruction set enables dynamic measurement and loading of code into protected regions in order to query the protection status of modules and request tokens for authenticated communication between processes. These mechanisms ensure that:

1. The text section code is executed only by jumping to a predefined entry point.
2. The software module's protected data section is accessed only during the execution of the text section code of that module

These tasks are implemented and performed in hardware. The memory access control mechanism is in charge of granting access rights. The control is carried out on a program counter-basis, i.e., it depends on the current value of the Program Counter (PC). In particular, reading from the protected text section is always allowed, because it must be included in Message Authentication Code (MAC), attached to the authentication response. On the other hand, execution of the text section is allowed only when needed, i.e., when PC value is within the module's text section. The entry point, which is always executable, is the only exception. In SANCUS, these protection mechanisms are implemented with a special CPU instruction. Such an instruction takes in as parameters SP identity and layout information, then computes the node key $K_{N,SP,SM}$ and stores it in a protected storage area, jointly with the layout. As in SMART, when a memory violation is detected the CPU is reset. Availability, in this case, has been excluded from the attacker model.

3.3 Related work

Among many others within the field of hardware security verification, we refer to a couple of related works which are closely related to ours.

Subramanyan et al. [50] gave a first definition of a Taint property and used it to create a methodology for hardware security requirements verification. They

applied their Taint propagation approach for automatic formal analysis of a commercial SoC design. The experimental results were carried out with a standard verification tool and showed some security vulnerabilities in the target architecture. The most important aspect of a Taint property is that it can capture information flow-related behaviors. The authors express this key concept as follows: *"Many security requirements can be formulated as taint propagation properties that verify information flow between a set of signals in the design."*. We take this concept as a starting base upon which we build our extended approach of taint propagation.

Francillon et al. [67] proposed an organized analysis of Remote Attestation: definition of security within remote attestation protocols, desired services, deconstruction of necessary and sufficient set of properties needed for remote attestation support. The set of properties is expressed in natural language, then they are converted into hardware and security components of a secure architecture.

In this work we provide a formalization of generic security requirements and the definition of related Taint Properties for a secure embedded architecture. Experimental results are being conducted on remote attestation architectures SMART and SANCUS whose abstract models we present hereby. We also present verification results of a set of properties for both architectures. Compared to the ones of [50], the abstract models we employ in this chapter are more detailed, that means we are able to capture finer-grained behaviors. We also define security properties that take into account the temporal correlation among critical operations.

3.4 Adversarial Model

As per [67], we also assume that the attacker or adversary can compromise (i.e., obtain partial or full control over) the system, at any time, with any method. There must exist, though, a secret (i.e., a key) that the attacker cannot access, even if the system is fully compromised. Once the control is gained, the adversary can manipulate executing code on the CPU at will. The control spread further to interrupt scheduling, accessing readable memory (e.g., ROM) and modifying writable memory (e.g., RAM). The only limitations are those provided by the hardware, i.e., the attacker can not write to ROM. These restrictions could be lifted by hardware tampering, but that is not the scope of this work. The hardware is therefore

deemed to be unmodifiable. Side channel attacks and software vulnerabilities can also pose a serious threat over the system and the literature is full of papers on these topics (we will discuss about side channel attacks in Chapter 6), but they too are out of scope. We assume that there is a mechanism to protect the system under verification from software side-channel attacks, e.g., a software-only time channel attack. Other types of attack such as fault-based attacks (e.g., glitches on power or signals) that could lead the processor to execute instructions incorrectly, or skip some instructions, are also out of the scope of this work. Finally, we assume that the system is not prone to software vulnerabilities, implementation flaws and side channel attacks.

3.5 Security Requirements

The security notions and features we refer to in this chapter are as follows:

- **Key secrecy**: even though it could be any secret that must remain inaccessible to the attacker, in this Chapter we make use of an encryption (and decryption if we deal with symmetric protocols) key k, used as the foundation of a TCB to build support for higher level security feature, e.g., secure communications. The presence of a secret key, in conjunction with other related mechanisms, can be used to implement a remote attestation algorithm. We therefore assume that a secure algorithm is able to deterministically compute an attestation token α from a given key k and the device state s. Since k is the only secret, if the adversary gains access to it they could simulate the attestation protocol and forge a seemingly legitimate token α.

- **Exclusive Access**: the attestation procedure has exclusive access to k. This feature does not imply that the attacker cannot acquire bits of information that leak from intermediate algorithm steps. We then need to express another property that expands this one.

- **No leakage**: intermediate results must not be accessible from outside the cryptographic routine when they are no longer needed, i.e., after the completion of the routine. This could be rephrased as: the system's state is statistically independent from k.

- **Isolation (Immutability and Mode separation)**: in addition to the key secrecy we need another guarantee to preserve attestation token's validity. Let us suppose that key secrecy is indeed assured by hardware mechanisms, the attacker could invariably modify the attestation code itself to tamper the result or leak the key. From this discussion the need for the immutability of the code. This feature can be achieved by, for example, executing the critical code *in place*. If not the case (performance or cost constraints), the critical software should be, at least, isolated, i.e., reading and writing the code or its internal state must be prevented. Hardware or software, this is matter of implementation choice. For example, SMART decides to place the attestation routing in ROM, which is available in many platforms and is considered an inexpensive solution. Immutability and mode separation, paired together, prevent a well known class of attacks known as time-of-check-to-time-to-use (TOCTTOU), e.g., the adversary tampers the attestation routine after it is loaded into RAM, but before it is executed.

As properly underlined by [67], the above conditions are not sufficient to guarantee the validity of the attestation protocol. In their work, the authors show a number of scenarios in which attackers can exploit existing vulnerabilities to influence control flow or alter the behavior of the attestation procedure, thus compromising the entire system. For this reason, we need two more security requirements:

- **Uninterruptibility**: execution of the attestation routine must be uninterruptible. A simple way to do this is to disable all interrupts. Hardware modifications or special instructions are two among several solutions, depending on the architecture constraints. There is only caveat: disabling the interrupts must be itself uninterruptible.

- **Invocation from start**: the attestation procedure must be invoked from its very first instruction. This could be enforced by OS-dependent features or protected CPU mode. If these solutions are not viable on the selected architecture, custom hardware is needed.

The last two requirements guarantee that the execution of the attestation routine is atomic. The above five requirements form the minimal set of properties to support secure remote attestation on a generic device.

3.6 Security Model Abstraction

Along the years, several proposal of languages to describe finite state machines have been discussed and analysed: from very early temporal logic model checking to symbolic model checking, until binary decision diagrams (BDDs) led to greater increase in design size. Despite of great advancements in the field of formal verification, large designs are still computationally too intensive for common verification approaches. To handle such designs, model abstraction is necessary. This approach [68] takes in the original design, described in some hardware description language (HDL), abstracts away the details and reduces components' size. The output is a higher-level abstract description of a non-deterministic automaton that encapsulates the behavior of the original model. This new abstract representation replaces the original one, it is proven to be functionally equivalent and it has the great advantage of being computationally feasible, i.e., the number of states is smaller.

We now present a methodology to carry out model abstraction on a generic remote attestation architecture, starting from the specifications we described above. Figure 3.3 shows the building blocks of a generic secure embedded architecture that supports remote attestation. Since implementation details are the scope of this chapter, security components (dashed boxes) that enforce the above requirements are either embedded in regular hardware (e.g., CPU, RAM, ROM, etc.) or they are implemented as custom modules (as previously discussed). The selected components are:

- Attestation Entity (AE): the component that implements the attestation protocol, as in Figure 3.1. It runs the attestation routine with its arguments (see 3.2), and outputs the attestation token α. The target code, here the Code to Attest (CA), is verified against a predetermined (expected) state. This component may be split into several hardware pieces if needed, but here it is represented as a whole. As described in Section 3.2, the Attestation Entity is the root of trust for remote attestation, enforcing confidentiality and authentication.

- Loader: its primary purpose is loading the AE code in memory. This component is considered to be untrusted, i.e., an adversary can seize control over it. The implementation vastly depends on the architecture: a standalone

hardware component or an OS kernel module.

- Memory Access Control (MAC): imposes that

  1. protected memory (PD) can be accessed only from inside the AE routine (i.e., when the routine is running)
  2. the AE routine starts only from its predefined entry point
  3. interrupts are disabled when AE code starts

  This is the implementation of the requirements discussed in Section 3.5.

We just composed our abstract model starting from real architectures, indeed the above blocks can always be mapped back to actual hardware modules. For example, if we take a look at the secure architectures we picked, the AE role can be played by the ROM for SMART or by the RAM for SANCUS. Instead, for both cases, the CPU itself realizes the Loader and the MAC. Table 3.1 shows the exhaustive mapping between our abstract model and SMART and SANCUS.

The detailed descriptions of SMART and SANCUS are reported in Section 3.2. The Attestation Entity (AE) provides support and is part of the attestation protocol. The corresponding attestation routine (i.e., the attestation code) is stored in ROM. The main memory (in the figure is represented by the RAM) includes the Code to Attest (CA) and the Protected Data (PD). The code to attest is identified by its boundaries: entry point A and exit point B (*Code(A,B)*). CA and PD are, in SMART terminology, a software module SM (consisting of a text section and a protected section). The Memory Access Control enforces the access control checks providing a secure support for the attestation entity. We also need a component which is responsible of loading software modules: the Loader. The last two components could integrated inside the CPU, but we decide to leave them outside for sake of simplicity. From now on, we will refer to Table 3.1 as our naming convention for our model components.

3.7 Taint Properties Verification

We now derive a set of security properties from the high level requirements discussed in Section 3.5. Then, the properties are formally verified with a standard

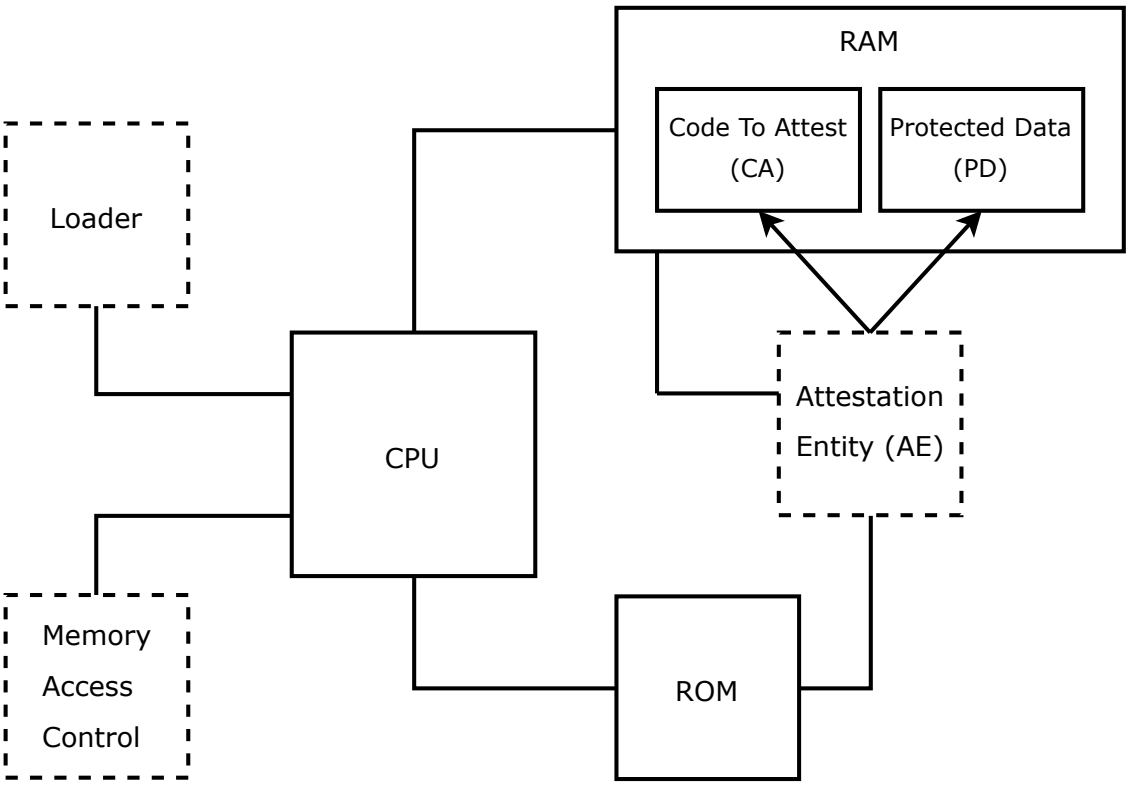

Figure 3.3: Secure Abstraction Model

Abstract Model	**SMART**	**SANCUS**
Attestation Entity (AE)	Attestation Routine (ROM)	Text Section of SMs (RAM)
Code to Attest (CA)	Code (A,B)	SM
Memory Access Control (MAC)	Bus Controller	Memory Access Logic (MAL)
Loader	-	Modified CPU

Table 3.1: SMART and SANCUS: Abstract Model mapping

model checker and the verification results are shown in Section 3.8. As already asserted in the Introduction (Introduction), a Taint Propagation Property is a means to express information flow-related behaviors of a system. We adopt the same notations of [50], that is we describe the system as an RTL representation, hence a taint property expresses an information flow between a set of signals:

- *src*: a source signal where the taint is injected (i.e., introduced)
- *dst*: a destination signal where the taint is detected (i.e., the sink or target)
- *conditions*: a set of statements (expressed by a chosen description logic) that

must be satisfied by the property (they can be viewed as constraints for the model checker).

In this scenario a taint is a malicious seed that is injected into the source, flows through the system until it reaches its sink. If the conditions are satisfied and the taint is detected at the sink, the property is considered as having failed. Following the abstract model we introduced in Section 3.6, Figure 3.4 is an exemplification of how a taint property works in practice. In particular, the Figure depicts a property that captures the key secrecy requirement introduced in Section 3.5. The legal path, depicted as dashed line, represents the legitimate flow between the CPU, the AE routine in RAM, and the secret key. On the other hand, in the illegal path (solid line) the attacker replaces the AE routine with its malicious code and it is able to access the key (i.e., the taint is inserted between the AE routine and the CPU).

We choose to express our properties in Computation Tree Logic (CTL) [69], as the behaviors we want to capture are inherently non linear in time, thus needing a branching time logic.

Let us now define some variables we are going to use in the formalisation of our taint properties.

1. AE_start_addr and AE_end_addr : represent respectively the first and the last address of the AE code
2. $is\_in\_AE\_scope = PC \geq AE\_start\_addr \wedge PC \leq AE\_end\_addr$: Program Counter (PC) belongs to the AE address scope
3. $was\_in\_AE\_scope = PPC \geq AE\_start\_addr \wedge PPC \leq AE\_end\_addr$: PC for the previous instruction (PPC) pointed to the AE address scope
4. $AE\_started$: RTL signal high if the Attestation Routine started
5. AE_done : RTL signal high if the Attestation Routine ended

3.7.1 Temporal Operators

Let us suppose p to be a boolean formula, we define the following operators [70]:

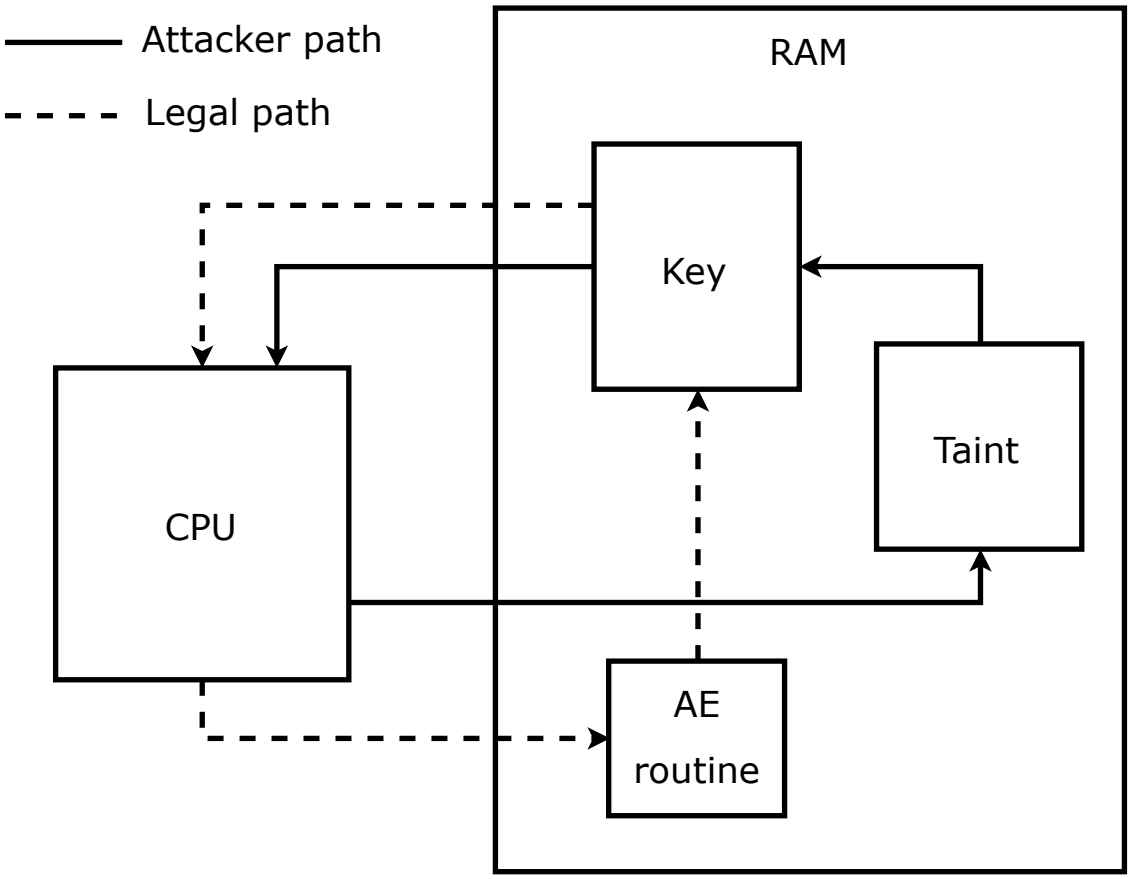

Figure 3.4: Taint Property Example

- Quantifiers over paths
  - **A**(p): p has to hold on all paths starting from the current state.
  - **E**(p): there exists at least one path starting from the current state where p holds.
- Path-specific quantifiers
  - **X**(p): p has to hold at the next state.
  - **G**(p): p has to hold on the entire subsequent path.
  - **F**(p): p eventually has to hold somewhere on the subsequent path.

The following subsections show the analysis and then the formalisation of the requirements as taint properties.

3.7.2 Key secrecy

Attestation routines resort to encryption/decryption algorithms to guarantee confidentiality. For the sake of simplicity, we focus on symmetric key algorithms and call K the encryption/decryption key. K is used by the Prover and the Verifier for the attestation routine. However, K is stored in a RAM memory location protected by a hardware cryptoengine. This is a common practice for secure embedded architectures. Only the Attestation Entity (AE) is allowed to access the key. The Memory Access Control (MAC) logic implements the controls related to key secrecy. In particular, if an instruction wants to access K, the MAC checks if the address in the Program Counter (PC) belongs to the AE address space. Consequently, only AE-resident code is allowed to access K. We now present two CTL properties that capture key secrecy requirements.

Exclusive access to the key: prop ks1

All user programs (untrusted entities) are not allowed to load K into CPU registers.

$$AG\Big(\neg is\_in\_AE\_scope \wedge fetchedInstr = load(key\_addr, CPU.registerX) \rightarrow AX\Big(\neg isIn(K, CPU.registerX)\Big)\Big) \tag{3.1}$$

where $isIn(arg1, arg2)$ is a function returning true if arg1 is included in the subset arg2, false otherwise. The primitive $load(arg1, arg2)$ loads the content of the address arg1 in arg2 register key_addr is the address of encryption/decryption key. As we are interested in the behavior verification we do not consider initial value $CPU.registerX$ equal to the key K.

Taint Untrusted entity accessing the key memory region. Entering the key memory region implies loading the content of the specific memory location in a CPU register. In our scenario only the Attestation Entity can access the key.

No key leakage: prop ks2

Once the cryptography routine is completed, K should not be present either into RAM locations or into CPU registers used during the encryption/decryption procedure.

$$AG\Big(AE\_done \rightarrow AX\Big(\neg isIn(K, RAM.AElocations) \wedge \neg isIn(K, CPU.AEregisters)\Big)\Big) \tag{3.2}$$

where $RAM.AElocations$ and $CPU.AEregisters$ are the RAM locations and the CPU registers, respectively, used during the encryption/decryption stage. Notice that we do not verify the encryption/decryption algorithm correctness since it is out of the scope of this paper.

Taint Interruption of the attestation procedure after the elaboration on the key but before cleaning out the memory from temporary results. This could allow an attacker to obtain information about the key using intermediate elaboration of it.

3.7.3 Isolation

In a secure architecture some operations and resources should be accessible by the supervisor only, guaranteeing key isolation, memory safety and atomic execution of AE code. Any user software (untrusted entity) has the total control of RAM memory, except for key location, after and before AE code. It can also modify any writable code and learn any secret that is not explicitly protected by MAC. User software can even invoke the AE code. An untrusted entity cannot interrupt the AE code execution, which must be executed atomically and cannot be invoked partially. In our scenario, the mode switching, from user to supervisor, occurs anytime a user's application calls AE code. We present a set of statements useful to check correct behaviors w.r.t. mode separation requirements.

Statement 1 (S1)

The current program counter PC points to the AE address space whereas the previous program counter PPC does not:

$$is\_in\_AE\_scope \wedge \neg was\_in\_AE\_scope \tag{3.3}$$

Statement 2 (S2)

The current program counter PC does not point to the AE address space whereas the previous program counter PPC does:

$$\neg is\_in\_AE\_scope \wedge was\_in\_AE\_scope \tag{3.4}$$

Mode separation: prop_ms

We focus on correct usage of the encryption/decryption (attestation) routine checking whether it starts by calling the entry point instruction and ends by calling the last AE instruction. We define two sub-properties to check the correct use of the encryption/decryption routine on entry (P_entry) and on exit (P_exit).

$$P\_entry = AG(S1 \rightarrow fetchedInstr! = middleInstr) \tag{3.5}$$

$$P\_exit = AG(S2 \rightarrow PPC = AE\_end\_addr) \tag{3.6}$$

where $fetchedInstr$ is the instruction is going to be executed and middleInstr is a generic AE instruction but the entry point.

Finally we combine the two above properties as follows:

$$P\_entry \wedge P\_exit \tag{3.7}$$

Taint Starting from the hypothesis that the attestation code is immutable and public, the proposed Taint tries to execute the attestation procedure skipping

the entry point. This Taint is implemented by jumping from any memory region (not involved in the attestation process) into the attestation code (with the exception of the first attestation instruction).

3.7.4 Availability

In order to avoid Denial-of-Service Attacks, untrusted software shall not be able to write to internal registers, to reset the CPU or to sweep all RAM memory. We consider the scenario where the attacker can perform an infinite loop in two ways:

- manipulating the cryptographic routine (AE code) return address
- interrupting the cryptographic (attestation) routine.

Availability: prop_av

We assume that the Attestation Entity (AE) return address x is stored in RAM. We deem that if the attestation routine started, it must end. We also verify whether an adversary can run malware during the attestation routine. First, we check that x does not belong to the AE address space (P_ret), then we verify that if the attestation routine started it ends ($P\_completed$).

$$P\_ret = AG(AE\_done \rightarrow AX(x \notin AE\_addr)) \tag{3.8}$$

where ($x \notin AE\_addr$) means that the return address does not belong to the AE address space.

$$P\_completed = AG(AE\_started \rightarrow AF(AE\_done)) \tag{3.9}$$

Finally we combine the two above properties as follows:

$$P\_ret \wedge P\_completed \tag{3.10}$$

Taint The idea behind this Taint is to make attestation procedure endlessly controlling the "return address after the attestation" x. The proposed Taint uses

as x an address included in the scope of the attestation entity. Since the PC always belongs to the attestation routine address space then the attestation task never ends.

Notice that Attestation and Confidentiality requirements are achieved if Key Secrecy and Isolation properties (defined above) hold. As described in [67], Attestation strictly depends on

1. Exclusive Access to K
2. No key leakages
3. Uninterruptibility
4. Invocation from start

Properties 1 and 2 correspond to the CTL key secrecy properties, whereas properties 3 and 4 correspond to the availability and mode separation requirements, respectively.

3.8 Experimental results

We applied our methodology, presented in Section 3.6, on real use cases. We extended and enhanced two basic CPU models, provided by the VIS-model checker [71]. New features have been added to support a remote attestation protocol. Namely we implemented two simplified models of the two selected remote attestation architectures, namely SMART and SANCUS. The setup for our experiments is an ordinary workstation with quad-core CPU 2.5 GHz and 16 GB of main memory. We set time and memory limits as 900 seconds and 4 GB, respectively. The rest of this Section reveals the experimental results in 3.8.1 for the formal verification of security properties defined in 3.7. In all our experiments we employed our custom PdTrav [72] model checker. We also provide a comparison in 3.8.2 between different PdTrav verification strategies in terms of performance.

3.8.1 Model Checking Results

We now present verification results for the security properties we defined in Section 3.7. Two experimental prototypes based on the SMART and SANCUS architectures have been implemented in Verilog. The properties have been then adapted for the prototypes and both have been given to PdTrav model checker. In Table 3.2 we list the properties (column "Property"), and for each of them: the verification time (column "Time") and whether the property has succeeded or not (column "Result"). bounded vs unbounded model checking *PASS* and *FAIL* mean that the model checker has proven the property to be true or false, respectively. In case of *FAIL*, the model checker returns a counterexample which shows how the property is failing. As already stated in its description in Section 3.2 SANCUS does not use ROM, therefore its model complexity is smaller than SMART's. The implication is that SANCUS's verification times are smaller than SMART's, for all properties. Every property but one is *PASS*: $prop\_ks1$, $prop\_ks2$ and $prop\_ms$ are proven to hold up to the model description; for $prop\_av$ the model checker found a security bug on both architectures. Namely, SMART and SANCUS are prone to DoS attacks. As per adversarial model, the attacker controls all the parameters given to SMART and SANCUS. Therefore, the attacker is able to manipulate the *return address after the attestation* x, and then easily chooses the code to execute after the termination of the attestation routine. By just setting x as the last instruction of the attestation routine itself the routine will enter an infinite loop. A very straightforward countermeasure is to check, at the beginning of attestation routine, that x does not point to the AE code itself.

Property	**SMART**		**SANCUS**	
	Time[s]	*Result*	*Time[s]*	*Result*
$prop\_ks1$	648.22	PASS	518.82	PASS
$prop\_ks2$	357.45	PASS	254.35	PASS
$prop\_ms$	156.7	PASS	116.74	PASS
$prop\_av$	378.24	FAIL	245.29	FAIL

Table 3.2: Unbounded Model Checking results SMART and SANCUS models

3.8.2 Verification Strategies

Table 3.2 shows verification results for unbounded model checking carried out using the Property Directed Reachability (PDR) verification strategy. PdTrav is a state-of-the-art model checking framework that implements different verification strategies. PdTrav is a set of model checking engines that support several model checking methods and symbolic reachability analysis. We wanted to know which strategy performs better on this particular kind of properties so we performed unbounded model checking and for each set of properties we changed the verification strategy. Here, we show and compare the results.

We selected the following verification strategies:

- Interpolant-based verification (ITP), with ad-hoc abstraction and tightening techniques [73], [74], integrated SAT-based approaches [75], [76], Interpolant reduction techniques [77] and Guided Refinement [78]
- Property Directed Reachability (PDR) verification strategies [79]
- BDD-based representations and traversals (BDD), including forward, backward, combined (approximate) forward/ (exact) backward reachability algorithms [80], [81] partitioned BDDs and/or image computation procedures [82], [83]

Figure 3.5 presents verification times considering the selected verification strategies. The most promising strategy for this type of properties is PDR. Since its appearance, PDR is known to perform better than other techniques and has the remarkable feature to finding deep counterexamples. Even though, on average, Bounded Model Checking outperforms PDR, there are many cases where PDR is able to find counterexamples that elude both BMC and BDD reachability. Moreover, as Bradley explained in his original work, PDR naturally tends to be suitable for parallel implementation.

3.9 Conclusions

This chapter presents abstract models for two remote attestation architectures. It also shows a new approach towards information flow-related security properties

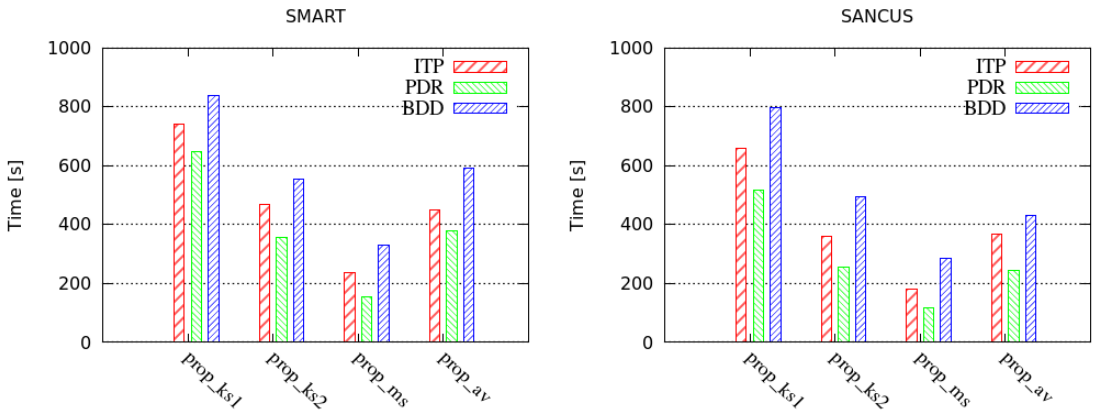

Figure 3.5: Pdtrav verification strategies results for SMART and SANCUS

on this type of architectures. From those models, we defined a set of properties which capture confidentiality, integrity and availability behaviors. We performed verification experiments with a standard model checker and then presented the results. They show that our approach is feasible and suitable for secure architectures like SMART and SANCUS. Though experimental results prove the effectiveness of this methodology, we still need an integrated framework that would comprise the entire verification process (i.e., model description, model abstraction, property extraction, property transformation and refinement, verification runs, etc.). Overall, this work opens new scenarios on formal verification of embedded systems security properties: new ad-hoc verification approaches shall be proposed both to reduce verification times and enhance bug coverage.

Chapter 4

Secure Path Verification

The following chapter has been published as peer-reviewed article in [84].

4.1 Introduction

Many modern embedded architectures and frameworks rely on two communication schemes: a centralized service that provides functionalities to client nodes or a decentralized solution in which every node contributes with its own task to the overall system. The two approaches are fueled by development costs reduction, reducing nodes complexity, decreasing maintenance time, etc. For instance, a centralized service moves all the costs and efforts to a single node, while unburdening the user nodes from their work load. Over time, the network growth might lead to an excessive load for the central node, thus a decentralized solution is needed. The choice must be made during the design phase to prevent such an unfortunate scenario that could be very costly and time consuming. The convenience of the two approaches has made embedded systems spread over a plethora of different applications: smart grids, robotic swarms, control systems, even smartphone-related scenarios. Either cases, a key feature is that applications running on top of these frameworks combine data coming from remote services with information provided by local sensors such as accelerometer, GPS, camera, microphone, etc. Users applications often have genuine and legal reasons for accessing this sensitive data, but providers on the other hand would also like assurances that their data is not disclosed or improperly used. We have seen attacks that exploited the trust developers

and users put in remote services [14, 85, 86] or privacy threats posed by seemingly innocent sensors [87]. Therefore, a critical challenge for embedded systems is about resolving the tension between the industrial drive (e.g., cost reduction and time preserving) and the security concerns. Most of the embedded, hardware and software, security components (e.g., OS) currently provide only coarse-grained controls for regulating whether an application can access sensitive information, but provide little insight into how data is actually used. As most of the functionalities and application in today's embedded systems involve security aspects, the demand for trustworthy and reliable security-driven design techniques increases dramatically. At the same time, the integration effort increases as the impact of these techniques on existing designs is very complex and hard to predict at the security level. Therefore, only extensive verification can overcome this issue, leading to cost explosion. For instance, a common measure to protect data confidentiality is through access control: every request must be granted by a supervisory entity. Access control introduces limitations on the release of information but it does not restrict data propagation. After data leaves its source storage, it might be improperly transmitted to unexpected destinations or might take incorrect/inadmissible paths. Real computing systems are not trustworthy by definition and the developers should not assume their reliability for granted. Security mechanisms such as remote attestation, identity verification and secure boot are vastly studied but up to date they do not provide full assurance that security goals (i.e., confidentiality, integrity, availability) are met in every scenario. Therefore, there is a need for guaranteeing that information is used exactly how policies and requirements specify. It is crucial to analyse how information flows throughout the system or the application and it can not be done manually considering the increasing complexity of current embedded systems. Proof that a system meets its security requirements must then proceed from a thorough formal analysis. It must be proved that sensitive data cannot flow to locations where the confidentiality requirement is infringed. The security requirements we want to guarantee are, in this case, confidentiality-driven and they are expressed as information-flow policies. As a consequence, the approach we are going to apply is information-flow tracking. In an ideally secure system, these information-flow properties could be accurately defined and translated into mechanisms that actually enforce them. However, despite great efforts

towards that aim, practical methods for controlling information flow are not mature yet. Recently, new encouraging approaches have been proposed about this matter. Among many others, some authors (e.g., [88, 89]) focus on static analysis and use specifically designed type systems for information flow. Notably, they introduce a variety of security-typed languages: alongside their corresponding programming language specifications, enhancements on data types and expression logic evaluation have been put in place. These annotation-augmented languages are able to specify information-flow schemes over typed variables. Then, a compile-time type checking system, meaning that the overhead is little or even none, enforces the security policies. Other authors propose reasoning techniques on how properties can be described in a security-typed system using semantics-based security models (e.g., [90, 91, 92, 93, 94]). As for dynamic techniques, researchers have proposed different tools and models to monitor execution flow ([95, 96, 97, 98, 99, 100, 101]). However, this approach requires runtime support, it is known to achieve poor coverage and it might lead to false positives and missed vulnerabilities. On the line between static and dynamic, some researches have focused on symbolic execution, which uses dynamic information along with static support to detect individual specific paths ([102, 103, 101, 104]). Yet, this methodology requires significant runtime support, it is still commonly OS-dependent, and it relies heavily on a great human effort as it needs extensive decision procedures to produce targeted queries. Another similar, but very different from a certain point of view, is the formalization of generic security requirements as propagation properties expressed in a formal logic and then formally verified with a model checker. This is the case of taint properties, which we extensively examined in the previous Chapter.

We present a new kind of properties, which are built on top of and are an extension of taint properties: Path Properties. The name has been chosen to remind that they are able to capture security vulnerabilities concerning the flow of data propagating from an attacker-controlled input source to selected sink, i.e., they cover a path from one entity source to an entity destination. For example, the source may be a potentially malicious (i.e., untrusted) RTL (Register Transfer Level) register or signal and the destination a sensitive RTL output signal that we do not want to be reached by confidential data. This is the case of a confidentiality requirement. In this Chapter we introduce a formal definition of Path properties, described above

and expressed in CTL [105], and compare them with Taint properties. We provide also a set of properties, making a distinction between advantages and drawbacks of both approaches. We also deliver our experimental results of formally verified properties on a selected secure architecture model. A comparison of two verification techniques used in a standard model checker is carried out.

4.1.1 Contributions

In details, our major contributions are the following:

- given some reference security requirements, we define a set of Path properties and we compare them with Taint properties.
- we introduce a methodology for the formal verification of Path properties.
- we perform experiments on the two selected architectures (SMART [51] and SANCUS [52]) with a model checker and present the corresponding results
- we provide a thorough comparison between the current experimental results and with those of [43]

Section 4.2 recalls related works and compares them with our approach. In Section 4.3 we depict the attacker model we refer to in this work. Section 4.4 reports the abstraction steps to build up a general abstract model of a secure architecture. Section 4.5 presents an analysis of security requirements and shows how to extract security properties out of them. A portfolio of Taint and Path properties is given with a comparison on advantages and disadvantages. In Section 4.6 we discuss security properties verification approaches in details. In Section 4.7 we offer experimental results that support our claim. Eventually, Section 4.8 completes the chapter with summarizing conclusions along with some remarks on future work.

4.2 Related work

As this is a direct follow-up of the previous Chapter, we refer to the same related works.

Among many others within the field of hardware security verification, we refer to a couple of related works which are closely related to ours.

Subramanyan et al. [50] gave a first definition of a Taint property and used it to create a methodology for hardware security requirements verification. They applied their Taint propagation approach for automatic formal analysis of a commercial SoC design. The experimental results were carried out with a standard verification tool and showed some security vulnerabilities in the target architecture. The most important aspect of a Taint property is that it can capture information flow-related behaviors. The authors express this key concept as follows: *"Many security requirements can be formulated as taint propagation properties that verify information flow between a set of signals in the design."*. We take this concept as a starting base upon which we build our extended approach of taint propagation.

Francillon et al. [67] proposed an organized analysis of Remote Attestation: definition of security within remote attestation protocols, desired services, deconstruction of necessary and sufficient set of properties needed for remote attestation support. The set of properties is expressed in natural language, then are being converted into hardware and security components of a secure architecture.

In this chapter we provide a new formalization of generic security requirements and their related Taint Properties for a generic embedded architecture. We also provide the abstract models of SMART and SANCUS, while presenting verification results of a set of properties for both architectures. Compared to the ones of [50], the abstract models we employ in this chapter are more detailed, that means we are able to capture finer-grained behaviors. We also define security properties that take into account the temporal correlation among critical operations.

The application of an abstraction approach to build up an abstract model of secure architectures is given in [49] and [43]. The authors specifically target remote attestation architectures and provide a portfolio of Taint properties related to security requirements such as confidentiality and integrity. Another key contributions is that they couple their approach with a standard model checker and apply Bounded (BMC) and Unbounded (UMC) Model Checking techniques for their experiments.

4.3 Attacker Model

The attacker model that we report here is the same as in the Section 3.4.

As per [67], we also assume that the attacker or adversary can compromise (i.e., obtain partial or full control over) the system, at any time, with any method. There must exist, though, a secret (i.e., a key) that the attacker cannot access, even if the system is fully compromised. Once the control is gained, the adversary can manipulate executing code on the CPU at will. The control spread further to interrupt scheduling, accessing readable memory (e.g., ROM) and modifying writable memory (e.g., RAM). The only limitations are those provided by the hardware, i.e., the attacker can not write to ROM. These restrictions could be lifted by hardware tampering, but that is not the scope of this work. The hardware is therefore deemed to be unmodifiable. Side channel attacks and software vulnerabilities can also pose a serious threat over the system and the literature is full of papers on these topics (we will discuss about side channel attacks in Chapter 6), but they too are out of scope. We assume that there is a mechanism to protect the system under verification from software side-channel attacks, e.g., a software-only time channel attack. Other types of attack such as fault-based attacks (e.g., glitches on power or signals) that could lead the processor to execute instructions incorrectly, or skip some instructions, are also out of the scope of this work. Finally, we assume that the system is not prone to software vulnerabilities, implementation flaws and side channel attacks.

4.4 Model Abstraction

The abstract model that we describe here is the same as in the Section 3.6.

Along the years, several proposals of languages to describe finite state machines have been discussed and analysed: from very early temporal logic model checking to symbolic model checking, until binary decision diagrams (BDDs) led to greater increase in design size. Despite of great advancements in the field of formal verification, large designs are still computationally too intensive for common verification approaches. To handle such designs, model abstraction is necessary. This approach [68] takes in the original design, described in some hardware description language (HDL), abstracts away the details and reduces components' size. The output is a

higher-level abstract description of a non-deterministic automaton that encapsulates the behavior of the original model. This new abstract representation replaces the original one, it is proven to be functionally equivalent and it has the great advantage of being computationally feasible, i.e., the number of states is smaller.

We now present a methodology to carry out model abstraction on a generic remote attestation architecture, starting from the specifications we described above. Figure 4.1 shows the building blocks of a generic secure embedded architecture that supports remote attestation. Since implementation details are the scope of this chapter, security components (dashed boxes) that enforce the above requirements are either embedded in regular hardware (e.g., CPU, RAM, ROM, etc.) or they are implemented as custom modules (as previously discussed). The selected components are:

- Attestation Entity (AE): the component that implements the attestation protocol. It runs the attestation routine with its arguments, and outputs the attestation token α. The target code, here the Code to Attest (CA), is verified against a predetermined (expected) state. This component may be split into several hardware pieces if needed, but here it is represented as a whole. The Attestation Entity is the root of trust for remote attestation, enforcing confidentiality and authentication.

- Loader: its primary purpose is loading the AE code in memory. This component is considered to be untrusted, i.e., an adversary can seize control over it. The implementation vastly depends on the architecture: a standalone hardware component or an OS kernel module.

- Memory Access Control (MAC): imposes that

    1. protected memory (PD) can be accessed only from inside the AE routine (i.e., when the routine is running)
    2. the AE routine starts only from its predefined entry point
    3. interrupts are disabled when AE code starts

We just composed our abstract model starting from real architectures, indeed the above blocks can always be mapped back to actual hardware modules. For example, if we take a look at the secure architectures we picked, the AE role can

be played by the ROM for SMART or by the RAM for SANCUS. Instead, for both cases, the CPU itself realizes the Loader and the MAC. Table 4.1 shows the exhaustive mapping between our abstract model and SMART and SANCUS.

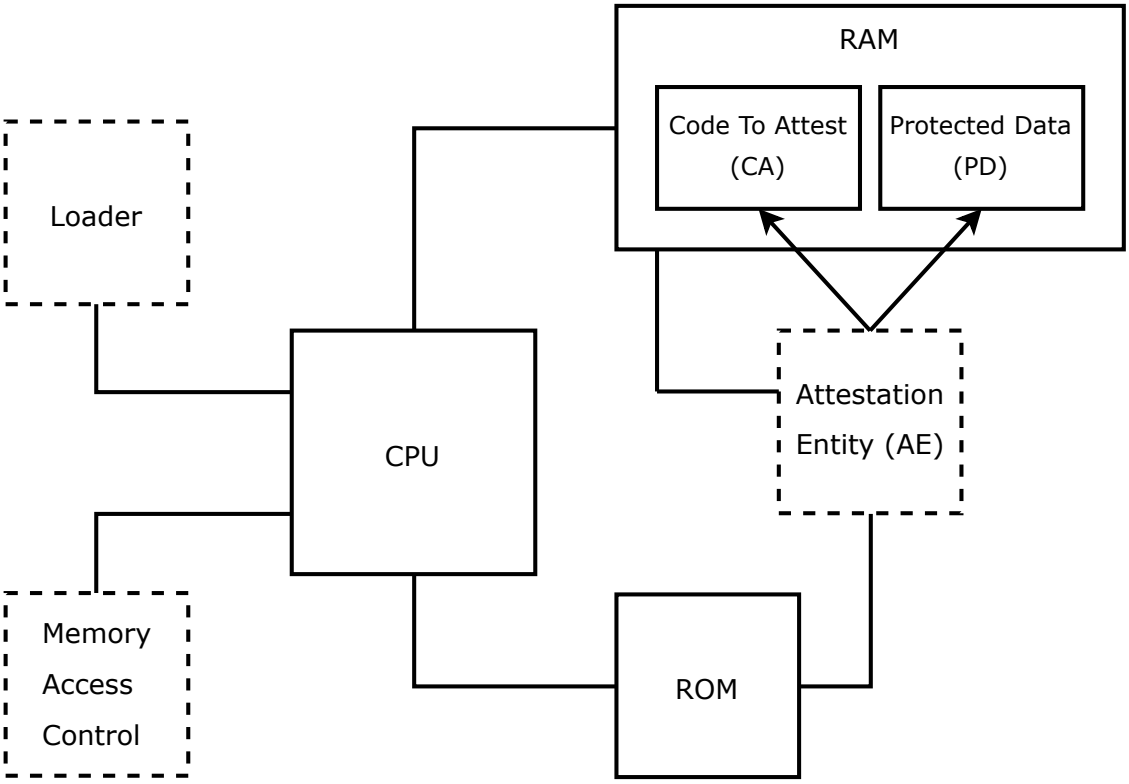

Figure 4.1: Secure Abstraction Model

The Attestation Entity (AE) provides support and is part of the attestation protocol. The corresponding attestation routine (i.e., the attestation code) is stored in ROM. The main memory (in the figure is represented by the RAM) includes the Code to Attest (CA) and the Protected Data (PD). The code to attest is identified by its boundaries: entry point A and exit point B (*Code(A,B)*). CA and PD are, in SMART terminology, is a software module SM (consisting of a text section and a protected section). The Memory Access Control enforces the access control checks providing a secure support for the attestation entity. We also need a component which is responsible of loading software modules: the Loader. The last two components could integrated inside the CPU, but we decide to leave them outside for sake of simplicity. From now on, we will refer to Table 4.1 as our naming convention for our model components.

Abstract Model	SMART	SANCUS
Attestation Entity (AE)	Attestation Routine (ROM)	Text Section of SMs (RAM)
Code to Attest (CA)	Code (A,B)	SM
Memory Access Control (MAC)	Bus Controller	Memory Access Logic (MAL)
Loader	-	Modified CPU

Table 4.1: SMART and SANCUS: Abstract Model mapping

We also define some variables for the Model Abstraction introduced above.

1. *AE_start_addr* and *AE_end_addr*: represent respectively the first and the last address of the AE code.

2. $is\_in\_AE\_scope = PC \geq AE\_start\_addr \wedge PC \leq AE\_end\_addr$ Program Counter (PC) belongs to the AE address scope

3. $was\_in\_AE\_scope = PPC \geq AE\_start\_addr \wedge PPC \leq AE\_end\_addr$ PC for the previous instruction (PPC) pointed to the AE address scope

4. *AE_started*: RTL signal high if the Attestation Routine started

5. *AE_done*: RTL signal high if the Attestation Routine ended

4.5 Security Properties Definition

In this Section, we present security properties that can be checked by a Model Checker, named Taint properties, and properties used to check whether there are possible paths outside the normal specification scope, named Path properties. Notice that the latter can be verified by a Model Checker as long as the model is modified (see Section 4.6 for further details).

Taint properties are composed of a pre-condition and a Taint. In this scenario, a Taint is an untrusted code within the RAM that tries to reach a destination following an illegal path. The idea of this approach is to write security properties, using CTL logic, and emulate hacker behavior trying to break the specification. In this way the properties are specified to verify that the corrupted software cannot violate the security constraints. Notice that (1) to write these properties, security knowledge is needed, (2) even considering a huge amount of attacks the verification can not be complete.

On the other hand, the idea behind Path properties is that the possible hacker code is not known but the correct behavior of the system is. Thus we can isolate the legal information flow and disable it, in order to check whether there exist other possible flows. In that case they are not legal. Starting from this assumption, the properties are written in such a way to verify if the hardware and software constraints properly guarantee the system fidelity to the project requirements ensuring that no other behaviors are admitted except for the expected one. The Path properties are composed by $from$ and to statements that indicate the source ($from$) and the destination (to) of the path under evaluation, respectively. The statement *pre_cond* is used to introduce constraints [106]. The main advantages of using Path Properties are:

- Security experts are not needed to define them.
- They can be used to find out paths difficult to discover using Taint Properties.
- In contrast to the Taint Properties, Path Properties are able to find out illegal paths also considering complex model architectures, i.e., architectures embedding more than one CPU, having external devices connected and so on.

In the following, we define the security requirements for a generic remote attestation architecture. For each of them we present a set of Taint and Path properties related to the security model abstraction and variables presented in Section 4.4.

Key secrecy

K is the encryption or decryption key and can be used in Attestation or Message Confidentiality application flows.

In our approach we consider the secret key K stored in a protected memory location in RAM. This is true for most of the common secure embedded architectures.

The key is only accessible by the Attestation Entity (AE). All controls relative to key secrecy are implemented in the Memory Access Control logic (MAC) and concern the CPU Program Counter (PC). More in details, when an operation wants to read K, MAC monitors PC checking if it belongs to the AE address space; as a result, K should be accessed only from within AE-resident code.

To verify the correctness of an architecture w.r.t. the key secrecy requirement, in the following we present two properties.

prop_ks1*: exclusive access to the key*
All user programs (untrusted entities) are not allowed to load K into CPU registers.

Taint property:

$$AG(\neg is\_in\_AE\_scope \wedge$$

$$fetchedInstr = load(key\_addr, CPU.registerX) \rightarrow$$

$$AX(\neg isIn(K, CPU.registerX))$$

Taint
Untrusted entity accessing the key memory region. Entering the key memory region implies loading the content of the specific memory location in a CPU register. In our scenario only the Attestation Entity can access the key.
Path property:

$$assert\_no\_path \quad from \quad \textbf{AE} \quad to \quad \textbf{CPU.registerX}$$

$$pre\_cond:$$

$$\neg is\_in\_AE\_scope \wedge load(key\_addr, CPU.registerX)$$

where

- $isIn(arg1, arg2)$ is a function returning $true$ if $arg1$ is included in the subset $arg2$, $false$ otherwise.
- $load(arg1, arg2)$ loads the content of the address $arg1$ in $arg2$ register
- key_addr is the address of encryption/decryption key

Notice that, we assume the content of $CPU.registerX$ differs from K before verification starts.

prop_ks2*: no key leakage*
Once the cryptography routine is completed, K should be present neither into RAM locations nor into CPU registers used during the encryption/decryption procedure.

As we are interested in the behavior verification we do not consider initial value CPU.registerX equal to the key K.

Taint property:

$$AG(AE\_done \rightarrow$$

$$AX((\neg isIn(K, RAM.AElocations) \wedge$$

$$\neg isIn(K, CPU.AEregisters)))$$

Taint

Interruption of the attestation procedure after the elaboration on the key but before cleaning out the memory from temporary results. This could allow an attacker to obtain information about the key using intermediate elaboration of it.

Path property:

$$assert\_no\_path \quad from \quad \textbf{AE}$$

$$to \quad \textbf{RAM}.\textbf{AElocations} \wedge \textbf{CPU}.\textbf{AEregisters}$$

$$pre\_cond : AE\_done$$

where $RAM.AElocations$ and $CPU.AEregisters$ are the RAM locations and the CPU registers, respectively, used during the encryption/decryption stage. Notice that we do not verify the encryption/decryption algorithm correctness since it is out of the scope of this paper.

Isolation

In a secure architecture some operations and resources should be accessible by the supervisor only, guaranteeing key isolation, memory safety and atomic execution of AE code. Any user software (untrusted entity) has the total control of RAM memory, except for key location, after and before AE code. It can also modify any writable code and learn any secret that is not explicitly protected by MAC. User software can even invoke the AE code. An untrusted entity cannot interrupt the AE code execution, which must be executed atomically and cannot be invoked partially. In our scenario, the mode switching, from user to supervisor, occurs anytime a user's application calls AE code.

We present a set of statements useful to check correct behaviors w.r.t. mode separation requirements.

Statement 1 (S1)
The current program counter PC points to the AE address space whereas the previous program counter PPC does not:

$$is\_in\_AE\_scope \wedge \neg was\_in\_AE\_scope$$

Statement 2 (S2)
The current program counter PC does not point to the AE address space whereas the previous program counter PPC does:

$$\neg is\_in\_AE\_scope \wedge was\_in\_AE\_scope$$

prop_ms*: mode separation*
We focus on correct usage of the encryption/decryption (attestation) routine checking whether it starts by calling the entry point instruction and ends by calling the last AE instruction. We define two sub-properties to check the correct use of the encryption/decryption routine on entry (P_entry) and on exit (P_exit).
Taint property:

$$P\_entry = AG(S1 \rightarrow fetchedInstr \neq middleInstr)$$

where: $fetchedInstr$ is the instruction is going to be executed and $middleInstr$ is a generic AE instruction but the entry point.

$$P\_exit = AG(S2 \rightarrow PPC = AE\_end\_addr)$$

Finally we combine the two above properties as follows:

$$P\_entry \wedge P\_exit$$

Taint
Starting from the hypothesis that the attestation code is immutable and public, the

proposed Taint tries to execute the attestation procedure skipping the entry point. This Taint is implemented by jumping from any memory region (not involved in the attestation process) into the attestation code (with the exception of the first attestation instruction).

Path property:

$$assert\_no\_path \quad from \quad \textbf{AE}$$

$$to \quad \textbf{CPU}.\textbf{Registers}$$

$$pre\_cond:$$

$$(S1 \wedge fetchedInstr = middleInstr) \wedge$$

$$(S2 \wedge PPC = AE\_end\_addr)$$

Availability

In order to avoid Denial-of-Service Attacks, untrusted software shall not be able to write to internal registers, to reset the CPU or to sweep all RAM memory. We consider the scenario where the attacker can perform an infinite loop manipulating the cryptography routine (AE code) return address.

prop_av: *availability*

We assume that the Attestation Entity (AE) return address x is stored in RAM. We check that x does not belong to the AE address space.

Taint property:

$$AG(AE\_done \rightarrow AX(x \notin AE\_addr))$$

where $(x \notin AE\_addr)$ means that the return address does not belong to the AE address space.

Taint

The idea behind this Taint is to make attestation procedure endlessly controlling the "return address after the attestation" x. The proposed Taint uses as x an address included in the scope of the attestation entity. Since the PC always belongs to the attestation routine address space then the attestation task never ends.

Path property:

$$assert\_no\_path \quad from \quad \textbf{RAM}$$

$$to \quad \textbf{CPU.PC}$$

$$pre\_cond:$$

$$AE\_done \wedge PPC \in AE\_addr$$

4.6 Security Properties Verification

In this section we present methodologies to verify Taint and Path properties. As described above, Taint properties can be verified using well known Model Checking techniques. In order to verify Taint properties we focus on Bounded Model Checking. As already described in the Chapter 2, given a sequential system M and an invariant property p, SAT-based Bounded Model Checking (BMC) [42] is an iterative process to check the validity of p up to a given bound k. Since we assume that the details of the architecture under verification are known, we could estimate the maximum bound of the model. As an example, let us consider the verification of the Taint property $prop\_ks2$ presented in Section 4.5. Since this properties implies that Attestation Entity (AE) routine must complete, the maximum bound cannot be longer than the execution of the AE routine.

Conversely, as discussed in Section 4.5, Path properties cannot be verified using standard verification techniques, then we present a variant of standard equivalence checking techniques. Standard equivalence checking is commonly used to prove that the RTL code and the netlist synthesized from it have exactly the same behavior in all (relevant) cases. Considering Path properties, in this paper we use an optimized version of equivalence checking to assert there is no path between a source (src) and a destination (dst).

Among others, we chose the following way to specify whether there is no legitimate path from location src to location dst in a model:

1. disconnecting signal src from its driving logic

2. making two copies of the model obtained in step 1), generating two identical

versions of the model, say $C1$ and $C2$

3. ensuring that sensitive data is only stored in *dst* location of $C2$

4. applying equivalence checking to verify whether the two models are equivalent. In the affirmative case, there exists an illegal path from *src* to *dst*; otherwise, it does not

Figure 4.2 shows an example of Path property $prop\_ks1$ verification considering the model Abstraction presented in Section 4.4. The left hand box represents the model where AE and MAC, i.e., modules can access to the key (K), are disconnected and the CPU (*dst*) does not contain K (any value that differs from the K, ? <> K). The right hand box shows the model where *dst* contains the sensitive data K. If the two models are equivalent then there exists an illegal path from RAM (*src*) to the CPU (*dst*), i.e., a security bug is found.

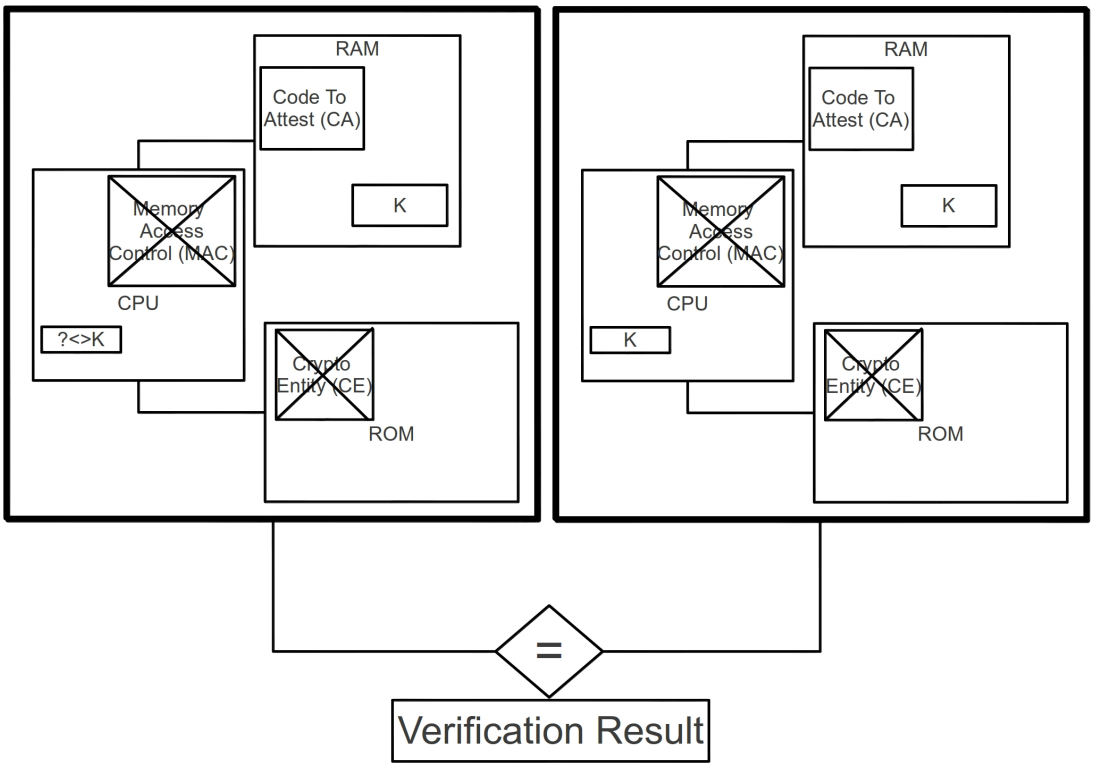

Figure 4.2: Secure Path Verification

4.7 Experimental Results

We applied our methodology, presented in Section 4.6, on real use cases. We extended and enhanced two basic CPU models, provided by the VIS-model checker [71]. New features have been added to support a remote attestation protocol. Namely we implemented two simplified of the two selected remote attestation architectures, namely SMART and SANCUS. The setup for our experiments is an ordinary workstation with quad-core CPU 2.5 GHz and 16 GB of main memory. We set time and memory limits as 900 seconds and 4 GB, respectively. We applied bounded (BMC) and unbounded (UBMC) model checking on both models. Through an analysis of the implementation details of the models, therefore we set the maximum bound for BMC as 800 for both.

We have verified the compliance of SMART and SANCUS with the security requirements defined in Section 4.5. The state-of-the-art PdTrav [72] model checker has been used for our experiments. PdTrav is a state-of-the-art model checking framework that implements different verification strategies. PdTrav is a set of model checking engines that support several model checking methods and symbolic reachability.

Figure 4.3 and Figure 4.4 depict a comparison between BMC results (this work) and UMC data presented in [43] for SMART and SANCUS respectively.

As shown, all properties except “prop_av” pass for UMC, same for BMC verification with a maximum bound of 800. The availability property “prop_av” fails for both verification methods (it fails at bound 357 for BMC). This means that we discovered a security bug on both architectures. The selected secure architectures, limited to model abstraction correctness and implementation errors, is sensitive to Denial of Service (DoS) attacks, as they disrupt the abiliyy of the system to function properly. As expected, verification times for UMC are higher than BMC. This is true for both models and all properties. A first set of experiments without the proposed approach gave higher times (which we do not show here) due to the state explosion. We then applied the proposed approach, thus preventing the state explosion, and drove the verification process. The verification times, Figure 4.3 and Figure 4.4, are significant smaller than before. This shows the effectiveness of our methodology.

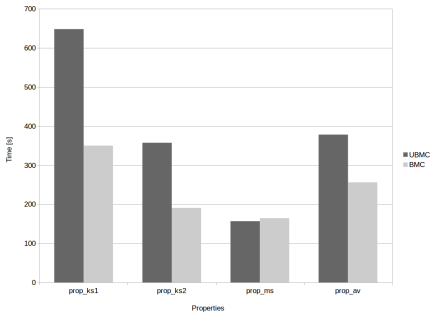

Figure 4.3: Verification time for SMART architecture

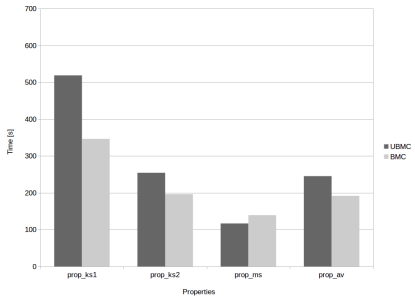

Figure 4.4: Verification time for SANCUS architecture

4.8 Conclusions

In this work we present Path properties as a natural expression of security requirements, such as confidentiality and integrity. We provide a methodology for formal verification of such properties within the framework of a standard model checker. We perform also a comparison between Path properties and Taint properties, highlighting advantages and disadvantages. Overall, verification results, for two selected secure embedded architectures, show the effectiveness of our approach. This work opens new scenarios on formal verification of embedded systems' security properties. New ad-hoc verification techniques can be exploited either to reduce the verification time and/or discovering security bugs. Future works will be focused on automated generation of Path properties.

Chapter 5

Embedded Systems Secure Path Verification at the HW/SW Interface

The following chapter has been published as peer-reviewed article in [107].

5.1 Introduction

The growing need in the silicon industry has raised the request for IP cores. The IP market has witnessed a steady growth during the last two decades due to several factors, such as lower design cost and reduced time to market. A major contribution to this proliferation is the use of System-on-Chip (SoC) platforms for mobile devices. A SoC platform is designed and then implemented by integrating multiple IP cores from trusted and untrusted IP vendors. Likewise, software is delivered by third party developers. Both hardware and software industries have risen security concerns about the rising number of untrusted third parties. As a consequence, great efforts have been made into researching and developing new methodologies to provide trustworthiness of third party resources. Until two decades ago, the low number of direct connections to the Internet and the plethora of embedded OSs contributed to a certain degree of security: obscurity was still a common approach to security and the targets were not worthy of being attacked. The new direction towards the Internet of Things (IoT), machine-to-machine (M2M) communications,

and remotely-controlled industrial systems, however, have boosted the number of connected devices. This trend, at the same time, has paved the way to new security attacks at the expense of such devices. Although there exists a considerable collection of attacks, this chapter specifically targets those who exploit implementation details of platform at hardware/software interface. Methods for prevention and detection of malicious behaviors have been introduced in the software domain, both via static and dynamic approachs. In the hardware domain, the scientific community and the industry have developed several proposals for verification and validation of SoC's at pre- and post-silicon level [108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118]. Formal methods (both automated and deductive), among all the existing techniques, have been very effective in detecting design flaws in both hardware and software [111, 119, 120]. The deductive approach, involving theorem proving techniques, requires significant manual effort for converting HDL code to a formal representation and constructing proof of security properties based on the design. Moreover, any modification on the design forces the entire deductive process to be repeated. Due to the time required for verification extending existing deductive methods to large scale designs has been a major challenge. To address this problem, new methods have been proposed in automated formal verification of SoC-based embedded systems.

Nonetheless, compelling challenges must be tackled down for a complete realisation of the potential benefits of formal methods within the security field [121]. First of all, limited scalability is a big issue, where large size models are computationally expensive to process. There is also a lack of proper techniques:

- formal modeling of the system and adversary behaviour
- formal specification of security properties
- systematic (and possibly automated) verification of properties

On the other hand, we still need a comprehensive approach for their integration across the layers of abstraction of real systems.

This work tries to push towards a solution of above issues picking up a specific class of security for low-end embedded devices: remote attestation. Recall from the previous chapters, Remote Attestation provides support for a trusted party (Verifier) to check Devices based on remote attestation allow a trusted party (Verifier)

to test the internal state of a remote untrusted party (Prover).

Several methods have been proposed for addressing information flow security, both for inter- and intra-devices communications. As for the first, namely network-related data flow workflows, Smith [122] and Russo et al. [123] perform a flow-sensitive analysis of programs. Knorr [124, 125] uses Petri nets to model workflows and applies the Bell–LaPadula model to workflow security. Watson [126] proposes a partitioning method to enforce that security requirements. Zeng et al. [127] introduced a flow sensitive security model to capture information flow in FCSs systems. Zeng [128] applies formal verification of secure information flow in cloud computing.

A second approach of information flow is for intra-devices communications/-computations and in particular it concerns software security. As with the huge expansion of the mobile market, it is not unusual to find lots of research in assuring security in this type of devices. Solutions range from the architectural support [129, 130] up to the OS level [131, 132, 133].

Although it has been mostly used in software security and networking, application of information flow tracking at the hardware/software layer has been growing steadily over the past decades. More precisely, the definition of security-oriented information flow properties has been defined and then verified with formal methods.

Gamage et al. [134] adopt information flow based enforcement mechanisms for a CPS security framework and provide a semantic model to analyze confidentiality violations. Palmer et al. [135] propose a formalized access control logic and an algorithm to provide inputs to that logic. This approach has been proven to be effective in industry, Intel has been using it for its pre-silicon validation phase.

Regarding the HW/SW interface, Subramanyan et al. [50, 136] recently formalized security requirements as propagation properties. gave a first definition of a Taint property and used it to create a methodology for hardware security requirements verification. They applied their Taint propagation approach for automatic formal analysis of a commercial SoC design. The experimental results were carried out with a standard verification tool and showed some security vulnerabilities in the target architecture. The most important aspect of a Taint property is that it can capture information flow-related behaviors. The authors express this key concept as follows: *"Many security requirements can be formulated as taint propagation*

properties that verify information flow between a set of signals in the design.". Cabodi et al. [43, 84] contributed with portfolios of information flow properties and introduced a formal framework to verify them.

In [43] we introduce an abstraction-based modelisation approach for secure architectures, we define a portfolio of Taint properties connected to remote attestation security requirements. We provide also a method to integrate such an approach with standard verification strategies (Bounded and Unbounded Model Checking) implemented in common model checkers. In [84] we present Path Properties, useful to formalize many information flow-related requirements as propagation properties. Hanna [106] has proposed Jasper as an effective alternative to standard model checking methods. The author starts from the question "*is it possible to access our sensitive information from outside the pin interface (or external access points) to the system?*" More specifically:

- Is it possible to read any security related data from a certain location A to location B through some control C?
- Is it possible to modify any security related data?
- Are there failure modes that would compromise the safety of any security related data?

The answer is the *Jasper Secure Path Verification App (SPV)* that partially automates the tedious structural analysis otherwise made manually by a verification engineer. The idea behind it is to apply structural analysis to traverse the cone of influence of the target signal B backwards and determine whether the traversal can reach signal A. If signal A in unreachable, it is proven that there is no path from A to B. However, if signal A is reached, there might be a functional path. In other words, backward traversal reachability is necessary but not sufficient for information flow between a source and a sink. The sufficient condition depends on the model logic and its assumptions. The Jasper SPV App applies the structural analysis as a simple pre-process method to rule out false paths.

Jasper Secure Path Verification is inherently dissimilar from a standard formal verification approach. It essentially checks whether there are possible paths outside the normal specification scope by mimicking the hacker approach: the specification is analysed and divided into smaller parts to discover information leaks through

unexpected paths. On the contrary, formal property verification checks whether the property is always (i.e., mathematically proven) correct under some constraints.

While Jasper has the hacker point of view, and thus is more focused on the input language/formalism and it is more simulation based, we essentially take the notion of propagation property and integrate it in a formal verification framework using a SAT-based verification tool. Moreover, in [106] only a custom closed-source use case is provided, instead our contribution includes a set of properties for an entire class of secure architectures.

"Traditional" properties, described in Linear Temporal Logic (LTL) or Computational Tree Logic (CTL), fail when it comes down to information-flow security. For instance, they cannot express properties like non-interference. New contributions have been proposed to solve this issue. On the semantic logic side, Clarkson et al. [35] propose a new class of properties, namely Hyperproperties. Finkbeiner et al. [34] introduce new logics for hyperproperties: HyperLTL and HyperCTL*. Even though it is a promising approach, it only focuses on the logic and their integration methods. It does not encompass a more general and comprehensive approach for a holistic integration process. For example, in [136] the authors use dynamic and symbolic simulation to verify flow properties. Differently, our approach is alternative: we provide support for a standard model checker for the verification of such properties.

In the current Chapter we redefine the class of properties presented in [43] as State properties since they are meant to express state-related properties, in contrast to information flow-related properties such as Path properties. The contribution of this Chapter is an accurate comparison between State Properties and Path Properties, that are two orthogonal approaches in security verification of different domains. We intend to provide a set of such properties and a formal framework to verify them. This framework starts from reduction and refinements steps of an abstract model, the derivation of properties from security requirements and the formal verification within a standard model checker. In addition, the formalization of the model and the properties still needs human effort and it is time-consuming, thus our goal is towards a partial or complete automation of this process.

Our contributions can be summarized as:

- we define a portfolio of security properties, driven by confidentiality, integrity

and availability requirements

- we provide formal definitions of two classes of properties: State Properties and Path Properties
- we perform a comparison and a formal analysis between them
- we show advantages and disadvantages of the two classes: in particular, when Path Properties perform better than State Properties and vice versa (e.g., for coverage results or for expressive power)
- we provide a method to integrate Path properties within the framework of a standard model checker
- we compare State and Path Properties also from an experimental point of view, showing promising experimental results that support our approach.

5.1.1 Roadmap

Section 5.2 presents the attacker model we refer to in this work. Section 5.3 provides an overview about the security requirements for remote attestation systems and presents State and Path properties definition and analysis. A brief discussion about security properties verification approaches is found in Section 5.4. Section 5.5 illustrates experimental results. Finally, Section 5.6 concludes the chapter with some summarizing remarks.

5.2 System and threat model overview

We address microcontroller-based platforms such as [51], [52]. The model we use in our experiments is an abstract and memory-reduced variant based on the above architectures.

The aim of our work is the HW/SW boundary, we provide both hardware and software implementations for some components. However, the correctness of the software is out of scope, even though we provide an implementation of the remote attestation routine. Therefore, our contribution can be seen as an empirical

(model checking oriented) step towards a co-verification approach, rather than a full HW/SW co-verification methodology.

This section summarizes the system model and the threat model.

5.2.1 Model abstraction and reduction

As the verification of the full (concrete) model would be impractical due to the state space explosion problem, we adopt abstraction and reduction techniques in order to achieve equivalence.

Model transformations can be carried out by automated tools or techniques, but they are out of scope. In this work we focus on manual decompositions, transformations and reductions.

There are several abstraction techniques: from descriptive language-based to manual decomposition or even automatic methods (e.g., localization abstraction) [137].

Abstraction creates a simpler and smaller model that approximates the concrete model. Typically, abstraction leads to an over-approximation, that is the abstract behavior includes the concrete one. The ultimate goal of abstraction is to preserve equivalence between the models, thus it is required to perform reduction (or *refinement*) techniques to narrow the abstract behavior and guarantee completeness [138]. Reductions are usually expressed as equivalence-preserving transformations. These transformations must be proven to have guarantee: if a property holds in the abstract model, then it must hold also in the concrete model. Under this assumption, we can say that verifying a property in the abstract model is equivalent to verifying the same property in the concrete model.

Typical examples of hardware-related reductions are memories and register files.

The model we hereby present is not a full System-on-Chip. Some components have been removed or abstracted away, because they are irrelevant for our verification aims. Memory, also, has been reduced to the minimum size, only the remote attestation routine and some additional data must fit in it.

In the following there is a list of selected components which we deem relevant for the verification process:

- *Cryptography Entity (CE)*: it is the hardware support for security-critical functions, e.g., cryptography functions.

- *Security Memory Access Control (SMAC)*: it is responsible for memory access control. Notably, it allows access to the protected memory region only when CE code is executing. Moreover, it checks that CE code is executed only from a predefined entry point and returns only from a predefined exit point.
- *Memory Data Register (MDR)*: a special CPU register that acts as a buffer between memory and the microprocessor.
- *Main Memory*: standard main memory with RAM and ROM. RAM and ROM have a single address space.

5.2.2 Threat model

We do not arbitrarily restrict the attack surface because a design that is verified secure under the most open assumptions will be still secure against less powerful attackers. In addition, the nature of our verification approach does not require the modelisation of a specific attack. Rather, our threat model includes any threat to the system, thus we assume that any system component can be a potential attack target.

In our threat model, an adversary is capable of launching remote attacks against the attester, but does not have physical access to the attester's computer. In other words, the attacker cannot launch hardware attacks (e.g., resetting the platform configuration registers (PCRs) in the TPM without rebooting the computer [34]). Even though the adversary can gain full control (e.g., root) over the attester's system by exploiting application level vulnerabilities.

Our attacker model is open, that is the attacker is unknown and is capable of launching arbitrary attacks against the system. The attacker does not have physical access to the hardware. Moreover, the hardware-based TCB imposes other few restrictions: the attacker cannot write to ROM, cannot raise interrupts if they are disabled [54]. Fault-based attacks and side channel attacks are out of scope.

5.3 Definition of Security Properties

In this Section we present (1) State Properties (SP) and (2) Secure Path Properties (PP) on a set of security checks. Our main targets are:

- finding a comprehensive set of properties able to represent and cover our security objectives;
- compare the power and ability of the two classes of properties on specific verification targets/problems.

We express State Properties as safety properties (invariants) in CTL. For Path Properties, though based on the theoretical background of hyperproperties [35] and [139], we prefer a formalism inspired to [106]. Path Properties are thus described by:

- From
- To
- Pre-condition (Optional)
- Post-condition (Optional)

State and Path Properties are conceptually different. The idea of the classical model checking approach (that we call State Property) is to capture the fact that the system state does not contain an unwanted behavior. Expressing a State Property often entails to emulate the hacker's behavior trying to leak or corrupt the system state. In our case, for instance, we check confidentiality of secure data by verifying that malicious code is not allowed to obtain a key. A State Property will basically check whether the key itself could, at a certain moment, be present in a CPU register. No propagation path is directly involved in this process, unless manually specified as a patch to the model description. Furthermore, no transformation on secure data is checked. On the other hand, a Path Property will provide two potential improvements: explicitly expressing a propagation path from a source to a destination and enabling to check data transformation through the path. Path Properties are thus more suited to capture/analyze information flows.

In the rest of this Section, we list and compare State and Path Properties for confidentiality, integrity and availability.
A naming convention, that we will use from now on, follows:

- CE_start_addr and CE_end_addr: represent respectively the first and the last CE code address

- CE_addr_space: represents the CE routine address space, that is $\{addr \mid addr \in [CE\_start\_addr, CE\_end\_addr]\}$
- $is\_in\_CE\_scope = PC \in CE\_addr\_space$: it means that Program Counter (PC) belongs to the CE address space
- $was\_in\_CE\_scope = PPC \in CE\_addr\_space$: it means that the PC at the previous instruction (PPC) pointed to the CE address space
- $CE\_started$: register-transfer level signal asserted during the attestation routine execution
- CE_done: register transfer level signal asserted when the attestation routine ends and for the duration of the instruction that follows the last one in CE
- R/W_signal: read-write signal asserted when the current instruction is a memory write
- $isIn(arg1, arg2)$: is a function returning $true$ if $arg1$ is included in the subset $arg2$, $false$ otherwise. From a verification point of view $arg1$ is always a single element, $arg2$, instead, can be a single element or a set of elements (for example addresses). If it is a single element this function can be exploded as $arg1 = arg2$. If it is a set of elements it can be seen as a sequence of equalities between $arg1$ and all the elements in the set $arg2$ (all these equalities are logically ORed)
- key_addr: is the address of the encryption/decryption key
- $trans\_in = is\_in\_CE\_scope \wedge \neg was\_in\_CE\_scope$: the program counter PC points to the CE address space whereas the previous program counter PPC did not
- $trans\_out = \neg is\_in\_CE\_scope \wedge was\_in\_CE\_scope$: the program counter PC does not point to the CE address space whereas the previous program counter PPC did
- ret_addr: Attestation Entity (CE) return address.

5.3.1 Confidentiality

Exclusive access to the key

(**prop_ks1**)

We call K the encryption/decryption key that in our scenario is used for attestation and for exchanging confidential messages.

K is stored in a protected memory location in RAM that is accessible only by the CE. From a design point of view all controls relative to key secrecy are implemented in the Security Memory Access Control logic (SMAC) and concern the Program Counter. The SMAC allows the read/write from/to the key only if PC belongs to the CE address space and MDR contains the key address. Access is denied for any other component.

User programs (untrusted entities) are not allowed to read K into CPU registers (represented here by the Memory Data Register (MDR) (Section 5.2). We reduce the set of possible key data values by adopting a unique value for K different, by definition, from all other memory/register values. Illegal reading is thus represented by the presence of $UNIQUE\_K$ in MDR.

State property

$$AG(\neg is\_in\_CE\_scope \wedge R/W\_signal = 0) \rightarrow$$
$$AX(\neg isIn(UNIQUE\_K, MDR))$$

An untrusted entity tries to access the key memory region. It means that someone is trying to load the content of a specific memory address (we look at the control bus signal) into any CPU register (we generalize using MDR).

Path property

FROM:	key_addr
TO:	MDR
PRE_COND:	$\neg is\_in\_CE\_scope \wedge RW\_signal = 0$

The Path Property states that any kind of data could not flow from the key

location to the MDR, so this is a more powerful check than simply verifying the presence of $UNIQUE\_K$ in MDR. The precondition is restricting the observed behavior to read operations, out of the CE address scope.

No key leakage

(**prop_ks2**)

Once the cryptography routine is completed, K should be present neither in RAM locations nor in CPU registers used during the encryption/decryption procedure.
Obviously, the initial value of CPU registers should exclude the unique key value K.

State property

$$AG(CE\_done \rightarrow AX((\neg isIn(UNIQUE\_K, RAM.CElocations) \wedge \neg isIn(UNIQUE\_K, CPU.CEregs)))$$

where $RAM.CElocations$ and $CPU.CEregs$ are the RAM locations and the CPU registers, respectively, used during the encryption/decryption stage. The verification of the encryption/decryption algorithm is out of scope.

Path property

FROM:	key_addr
TO:	$RAM.CElocations \wedge CPU.CEregs$
POST_COND:	CE_done

The property describes a path from the key location to the RAM or the CPU registers leading to potential leakages. The post condition is that the attestation routine has completed.
The taint reaches the destination and it is spot when the post-condition becomes true.

5.3.2 Integrity

Integrity is about maintaining consistency, accuracy, and trustworthiness of data over the entire execution. Data must not be changed in transit, and appropriate countermeasures are taken to ensure that data cannot be altered by unauthorized people. In a secure architecture some operations and resources should be accessible to the supervisor only, guaranteeing key isolation, memory safety and atomic execution of CE code. Any user software (untrusted entity) has the total control of RAM memory, except for key location, after and before CE code. It can also modify any writable code and learn any secret that is not explicitly protected by SMAC. User software can even invoke the CE code. An untrusted entity cannot interrupt the CE code execution, which must be executed atomically and cannot be invoked partially.

In this Section, we do not show how to verify integrity as key modification, because the corresponding properties would be dual to the ones related to confidentiality. Rather, we focus on an aspect of integrity, atomic execution, which means to guarantee the atomicity of the attestation routine in order to prevent data corruption. Here we define the corresponding atomic execution property (**prop_ms**).

Starting from the assumption that the attestation code is immutable and public, the proposed attack tries to execute the attestation procedure by skipping the entry point. This attack is thus implemented by jumping from any memory region (not involved in the attestation process) into the attestation code (with the exception of the first attestation instruction).

We check whether the encryption/decryption (attestation) routine starts by calling the CE entry point instruction and ends (returns) by calling the last CE instruction.

State property

$$AG(trans\_in \rightarrow PC = CE\_start\_addr \wedge$$
$$trans\_out \rightarrow PPC = CE\_end\_addr)$$

We did not express path properties on atomic execution, as this clearly is a state-based property.

5.3.3 Availability

As availability of information means ensuring that authorized parties have access to the information when needed, we consider Denial-of-Service as the main threat to availability. Here we define the corresponding Denial-of-Service property (**prop_av**).

In order to avoid Denial-of-Service Attacks, untrusted software shall not be able to write to internal registers, to reset the CPU or to sweep all RAM cells. We consider the scenario where the attacker can trigger an infinite loop manipulating the cryptography routine (CE code) return address.

We assume that the Attestation Entity (CE) return address ret_addr is stored in RAM.
We check that ret_addr does not belong to the CE address space.

State property

$$AG(CE\_done \rightarrow AX(ret\_addr \notin CE\_addr\_space))$$

where $(ret\_addr \notin CE\_addr\_space)$ means that the return address does not belong to the CE address space.

The idea behind this attack is to make the attestation procedure endlessly reloading the "return address after the attestation" ret_addr.

Path property

The State Property checks the behavior upon return from the attestation routine. This is clearly a check on the system state, that cannot be expressed as a Path Property. We rather express as a Path Property the integrity of the return address ret_addr while executing CE.

FROM:	MDR
TO:	ret_addr
PRE_COND:	$is\_in\_CE\_scope$

5.4 Verification strategy

In this section we show how to use an existing model checker in order to handle the properties mentioned in Section 5.1.

State properties can be checked using well known Model Checking techniques. We primarily exploit Bounded (BMC), as the duration of the remote attestation routine has a known upper bound, which makes a BMC-based approach complete.

In the experimental section, we will also show results of Unbounded Model Checking (UMC), when feasible within the adopted time limits, for sake of comparison, and in view of more general verification problems.

As an example, let us consider the verification of the State property $prop\_ks2$ presented in Section 5.3. Since this property implies that the Crypto Entity (CE) routine must complete, the maximum bound cannot be longer than the execution of the CE routine.

Conversely, as discussed in Section 5.3, Path Properties cannot be verified using a standard model checker, unless a specific (ad hoc) verification procedure is implemented, or the model is manually transformed in order to embed some information flow checking circuitry.

As an initial attempt to show the feasibility of the approach, we followed the latter strategy, keeping an hoc model checking engine as a future development.

In order to model information flow from source to destination, we follow a product-machine strategy typically adopted in equivalence checking. We generate two instances of the same model, where path sources are injected with different data (free inputs) and destinations are checked for equivalence. Any difference found means that a path from source to destination exists. More in detail, given src and dst, we:

1. disconnect signal src from its driving logic, or (equivalently) inject an arbitrary initial value in src register

2. make two copies of the model obtained in step 1), generating two identical versions of the model, say $M1$ and $M2$

3. apply any additional environment constraint (if needed) to src

4. verify whether the two models are equivalent at *dst*. If not, there exists an illegal path from *src* to *dst*; otherwise, it does not.

5.5 Experimental Results

In this section we report verification results for two selected Secure Embedded Architectures, considering the properties defined in the core section of this work.

5.5.1 Use cases

We focus our attention on two use cases: SMART [51] and SANCUS [52], that are Remote Attestation architectures.

SMART is based on hardware-software co-design, for establishing a dynamic root of trust in a remote embedded device. The attestation algorithm uses the hashing algorithm HMAC, which is located in a Trusted Region *TR* in ROM; whereas the secret key K is stored in a particular RAM location. In order to perform secure Remote Attestation SMART implements two control accesses on the memory bus controller: 1) K can be accessed only by the instructions located in the *TR* 2) an instruction can access *TR* only from the initial *TR* address and it can leave *TR* only from its last address. As a result, the secret key is only accessible by the trusted code stored in ROM and (except for the first instruction in *TR*) if the current Program Counter points into the *TR* region, then the previous Program Counter value must point also into *TR*.

SANCUS can remotely attest to a software provider that a specific software module is running untampered and it can authenticate messages from software modules to software providers. SANCUS achieves these security guarantees without trusting any infrastructural software on the device. The Trusted Computing Base (TCB) on the device is only the hardware. SANCUS consists of an Infrastructure Provider *IP* that manages a set of microprocessor-based nodes N and a Software Provider SP_i that deploys Software Modules *SM* into N: these software modules contain a text section (protected code and constants) and a protected data section. *IP* generates (exploiting the key generation functions) and manages three types of secret keys: 1) K_N, shared between node N and *IP* 2) $K_{N,SP}$, shared between a provider *SP* and node N 3) $K_{N,SP,SM}$, shared between a node N and a provider

SP, only accessible by *SM*.

A software provider *SP* can enforce Remote Attestation to verify that the correct software module *SM* is running on an expected node *N* through the module key $K_{N,SP,SM}$. The attestation code is saved into the text section of *SM* and the security memory access control logic (bus controller) ensures that: 1) The protected data section of a software module is only accessible while code in the text section of that module is being executed 2) The code in the text section can only be executed by jumping to a well-defined entry point.

5.5.2 Bounded and Unbounded Model Checking comparison

We have implemented prototype versions of the SMART and the SANCUS architectures on top of an extended version of the CPU model provided by the VIS model checker. We have performed bounded and unbounded model checking on both architectures. All our experiments were run on a Quad-core workstation, with CPU frequency at 2.5 GHz and equipped with a 16 GB main memory. The time and memory limits were always set to 3500 seconds and 4 GB, respectively.

We have verified if SMART and SANCUS are compliant to the security requirements defined in Section 5.3 applying BMC on the defined State properties using the PdTrav [72] checker. We have defined the bounds exploiting the implementation details of the attestation routines of both the proposed models. Considering the execution time of the CE routine, we set the maximum bound for BMC verification to 800.

Figure 5.1 shows a comparison between BMC results (this work) and UMC data presented in [43] for SMART and SANCUS respectively. We do not report experimental results of the PP/UMC verification since they time-out, as expected.

A more accurate analysis would show that the verification result of all properties, except for *prop_av*, is *PASS* for UMC, and *PASS* within the bound (800) for the BMC verification. Property *prop_av* fails due to abstraction.

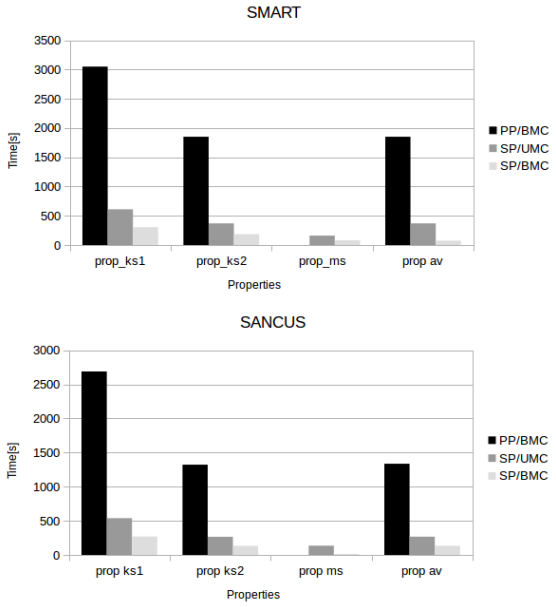

Figure 5.1: Verification time for SMART and SANCUS architectures.

5.6 Conclusions

This work addresses a specific and relevant class of embedded systems and of security properties. Remote attestation on low-end devices represents a meaningful security problem. On this example we identify a set of properties that cover the most significant aspects related to security: confidentiality, integrity and availability.

State and Path Properties are presented and their pros and cons are discussed. We highlight their different scope of use and coverage and we verify them using a standard model checker. Regarding Path Properties, we discuss how to verify them following an equivalence checking approach.

This work opens new scenarios on formal verification of embedded systems security properties. Novel ad-hoc verification techniques could be exploited either to reduce the execution time and/or discovering security bugs.

Future works will focus on automated model abstraction and generation of properties.

Chapter 6

Model-Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification

The following chapter has been published as peer-reviewed article in [140] and then extended in [141].

6.1 Introduction

Security management is rapidly changing to compete with the complexity of upcoming threats. Awareness that security affects privacy, business and public health safety is of upmost importance in this highly dynamic scenario. Both software and Hardware are growing more sophisticated, more compounded, more interconnected. Moreover, CPU performance optimization has been continuously pushing further by increasing clock frequency and shrinking processing technology. Recently, pipelined CPUs have been massively parallelized allowing mechanisms such as out-of-order instruction execution and speculative execution. As a consequence, the attack surface and the number of potential vulnerabilities are becoming bigger.

For example, *Speculative execution* is an industry common technique that works at the microarchitectural level used to improve performance in most of all modern CPUs. Lately, novel research work has demonstrated that speculative executed instruction sequences, under certain circumstances and in some architectures, leave measurable side-effects in the shared components (such as processor caches), even if instructions are not committed and their transient effect is discarded.

The recent Spectre [17] and Meltdown [16] attacks exploited this modern CPU optimization and produced a risky landscape that arises more questions than answers.

These two attacks belong to the side-channel attacks class as they extract sensitive data (or sensitive data-dependent information) by taking advantage of the physical environment, rather than exploiting implementation flaws (e.g., software bugs or buffer overflows).

Side channels are not a new concept, they have been around since the 80's. They have been the topic of quite an extensive research through the years [142, 143, 144, 145], but until recent times they were never considered to top class security threats. The reason behind that is that usually Side channels attacks demand thorough knowledge and very peculiar expertise about the target. Instead for Meltdown and Spectre the attacker would need relatively little technical expertise to perform the attack.

The speculative attacks, as we call the class of attacks Meltdown and Spectre belong to, combine speculative execution features and side channels attacks. Since speculative execution and branch prediction are implemented at microarchitectural (MA) layer. Typically a side channel attack will create a covert channel to transfer the microarchitectural state, transient and formally isolated from retired execution, to the architectural state where it could be retrieved by the attacker. Hence, the foundation of speculative attacks resides in this discrepancy [146] between the Instruction Set Architecture (ISA) and the MicroArchitecture (MA).

Even if the ISA has been demonstrated to be functionally correct by validation and verification processes, the actual MA could be in certain transient states prone to potential threats, e.g., cache prefetched data exploited by established techniques [147, 148, 149].

In other words, Spectre and Meltdown are variants of side channel attacks that

are able to make observable formerly isolated states of the microarchitecture. The attacker is then capable of transferring secrets from shared microarchitectural resources before any access or validity check is enforced.

Although these vulnerabilities are theoretically fascinating, industry has to cope with their practical impact. The scope of speculation-related attacks such as Meltdown and Spectre is so broad that it is extremely troublesome to address. A large variety of systems are likely to be affected, potentially any system that allows to run untrusted code: web application or browsers, workstations used by employees such as laptops, desktops, mobile devices, hypervisors in the public cloud, etc. Therefore Spectre and Meltdown are exceptionally difficult to triage. Since the outbreak it was clear that specific processor families were affected, but the full scope was suspected to be much wider. Spectre and Meltdown had a great impact which has not entirely been recognized yet [150]. Fixing these flaws has been a hard job because speculative execution is largely baked into processor hardware.

Even though several countermeasures have been proposed to relieve the impact of Spectre and Meltdown [151, 152], they usually are specific for one aspect of the attack, leaving other attack vectors open. In addition, these patches heavily affect performance as they come with high overhead. The reported magnitude of this overhead changes depending on the expected workload characteristics and the industry sector [153]. Besides, many solutions employ threat models that are too conservative and make assumptions that can be bypassed by future attacks.

Despite all the research efforts and the adopted countermeasures, security community still wonders: "*how do we prevent such attacks?*" and "*are our current verification methods suitable for such attacks?*". Sadly, most of hardware and software security methods focus on operating systems, applications, and computer networks. Even worse, they do not focus on early verification of those systems. Therefore there is a need for new approaches in hardware and software verification of out-of-order and speculative execution processors.

In this work, we propose a formal verification approach for security properties in pipelined out-of-order processors. Our focus is on the recently emerged speculative execution attacks such as Spectre [17] and Meltdown [16]. We address also scalability issues on large hardware design by providing an abstraction and reduction-based methodology. Correctness follows by well known literature achievements on formal

processor verification.

Even though automation of the approach is not full, we demonstrate that it is a viable approach: we perform experimental results on processor model inspired by the Hennessy and Patterson's DLX RISC processor [154]. The original design has been modified (e.g., reorder buffer, pipelining) to support Tomasulo's algorithm [155].

A summary of our transformations is the following:

- data abstraction
- refinement
- taint encoding
- pipeline reduction

After the application of model transformations, which preserve functionality and correctness, we feed the final abstract model to a model checker.

The present work is an extended version of [140].

The chapter is organized as follows:

- Section 6.2 provides state-of-the-art related work and background notions.
- Section 6.3 presents the processor architecture model.
- Section 6.4 gives a detailed description of an implementation of Spectre and Meltdown.
- Section 6.5 shows the verification approach we adopt on our case-study processor model.
- Section 6.6 produces experimental results to support the viability of our methodology.
- Section 6.7 proposes remarks and future directions in this field.

6.2 Background and Related Works

In the following, we shortly introduce notions and related work for better understanding of the following sections.

6.2.1 Side channels

Functionally correctness is not enough in order to guarantee confidentiality: confidential data must not be leaked to unauthorized parties. A particular class of attacks, called *side channel attacks*, is able to leak data through unforeseen information channels. Channels that take advantage of mechanisms whose main function is not information transfer are called covert channels [156]. A side channel is any measurable property of a system that could yield information about the system itself. Early practical side channel attacks have been demonstrated in [157]. Considering the nature of these attacks, engineers tries to carefully design their systems to remove as many side channels as possible. The literature shows that there is an amazing amount of attack vectors: microphones [158], power lines [159], energy consumption [160], electromagnetic emissions [161], IR photon detectors [162], ambient light sensor of smartphones [163], high resolution cameras [164]. Time itself can be used as a measurable property to obtain information: varying execution time for certain security-related operations provides information about the the operation itself or its secret data. This is the case of Time side channels that use differential analysis of the variation of time. The reason why time side channels are so widespread is because of the massive pressure on execution time reductions and performance optimizations of modern computing systems. Unfortunately, any shared resources can potentially be a side channel: disk [165], RAM [166], cache [167], memory bus [168], processor package [169], processor core [170], etc. Side channels can be found both in concurrent scenarios where different computational executions compete for shared resources or in time-sharing scenarios where states persist between context switches. A typical targeted resource is processor caches: accessing a cached memory location is faster than non-cached addresses, so this time variation tells what locations were recently accessed. One of the first examples of a side channels using cache timing is given in [167]. The authors in [171] provide a survey on timing channels that exploit microarchitectural components.

We now describe briefly three common cache timing attacks: Prime+Probe [172], Flush+Reload [142] and Evict+Reload [173].

Flush+Reload This cache side channel attack works on the last level cache (LLC). The attacker uses the *clflush* assembly instruction to flush out a cached memory location and then waits for the victim to access that address. The attacker

accesses again the target location and measures the access time. A short time implies the target content has been cached again, i.e., the victim accessed the target. Otherwise, the target has been retrieved from memory, i.e., the victim did not access the target.

Evict+Reload Evict+Reload is a variant of Flush+Reload because it does not use the clflush instruction. Cache eviction is achieved by using huge pages.

Prime+Probe The prime and probe technique is composed of two phases: first, the attacker occupies all cache sets (Priming) with known content and waits for the victim to use the same allocated cache lines. The attacker then looks for (Probing) cache sets that have evicted by the victim code and performs differential analysis on the access time.

6.2.2 Spectre and Meltdown Attacks

On the majority of modern microprocessors, single instructions are first decoded and sliced into micro-operations (μOPs) then executed. This design decision supports the extension of specific instructions and superscalar optimizations. Another level of performance optimization is the so called out-of-order (OoO) execution. Instructions are not executed in sequential order but they are dispatched to several execution units in parallel. This increases the parallelism and therefore the overall throughput. However, classic hazard are still applicable in this scenario. If the operands of a μOP are not available, the μOP is stalled until they become available. If both the instructions operands and the corresponding execution unit are ready, the microprocessors starts the execution even if previous instructions are have not finished yet. Only when all previous μOPs have finished, immediate results are committed to the architectural state. Instructions that are not yet committed are called *transient instructions*[16]. The microprocessor usually monitors the status of transient instructions with a specific component called Reorder Buffer (ROB). The ROB reorders the instructions as in the original instruction stream and decides whether to discard them or commit them to the architectural state. For example, interrupt and exceptions are treated by flushing the entire pipeline. The existence of transient instructions means that there is a time frame, between the execution and the retirement, in which temporary results reside somewhere at the

microarchitectural level (e.g in the shared LLC). This temporary non-coherence between microarchitectural and architectural states is the foundation of the so-called *speculative execution attacks*, to which Spectre and Meltdown belong. How can we exploit such scenario? Generally software contains data dependencies or control flow instructions such as conditional branches. In theory, the CPU would stall until the dependency or the branch condition are resolved. Speculative execution, however, implements mechanisms to predict or guess data dependencies or branch outcomes. The microprocessor continues executing along the predicted path and buffers the temporary results inside the ROB until the dependencies are resolved and the prediction is verified. If the prediction was exact, the CPU retires instructions and commits them. On the contrary, in the case of an incorrect prediction, the microprocessor performs a rollback to a previously correct state by flushing the pipeline and squashing (i.e., trashing all the results of) the ROB. From this point of view, transient instructions reflect the existence of unforeseen (and probably unauthorized) functional paths. As already stated, these unwanted states and their corresponding temporary results are never committed to the architectural state in order to guarantee functional correctness. Nevertheless, transient instructions may leave traces inside the microarchitectural state: the perfect scenario for microarchitectural side channel attacks. This behavior can be exploited to exfiltrate secret information about the system. Authors in [174] divide speculative execution attacks in two major classes: *control-steering* attacks and *chosen-code* attacks. Control-steering attacks (Fig. 6.1.a) allow the attacker's code to force the target control flow into a speculative wrong path. This is usually done by deliberately mis-train the Branch Target Predictors. Temporary execution results in this wrong path are then retrieved via a covert channel, even though instructions are eventually flushed out. For example, Spectre and its variants belong to this class. Code-chosen attacks (Fig. 6.1.b) enable privilege escalation: during wrong-path execution privileged memory contents are temporarily exposed to unprivileged malicious code. The attacker is able to choose, for both correct-path and wrong-path executions, arbitrary instruction sequences. For example, Meltdown and its variants belong to this class.

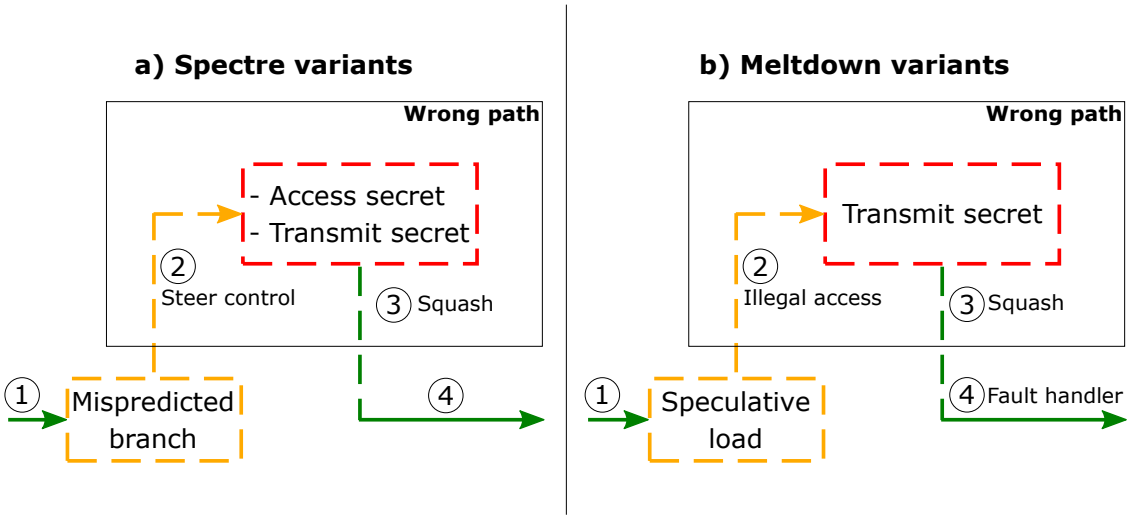

Figure 6.1: Spectre vs Meltdown attack classes

Speculative execution attacks generally follow the same abstract flow, depicted in Figure 6.2. An initial phase, not shown in the figure, is about preparing the system: the processor is mis-trained to expect a certain branch outcome, the cache is prepared for the side channel attack (i.e., flushing or evicting target cache sets). At this point, the *Access Phase* (1) starts: the attacker triggers the speculative execution of instructions that will be eventually squashed due to a mispredicted branch or an exception. The transient instruction sequence that follows acts as a microarchitectural covert channel. The secret or a secret-dependent memory location is loaded into a register. The sensitive data is then transferred, in the *Transmit Phase* (2), to the cache using micro-architectural side effects previously described. In the *Retrieve Phase* (3) a cache timing analysis indicates whether and what cache locations can be probed to retrieve the sensitive data and store them into a non-speculative state. Ultimately, the instructions involved in phases (1) and (2) are squashed, but phase (3) already committed the temporary results to the architectural state.

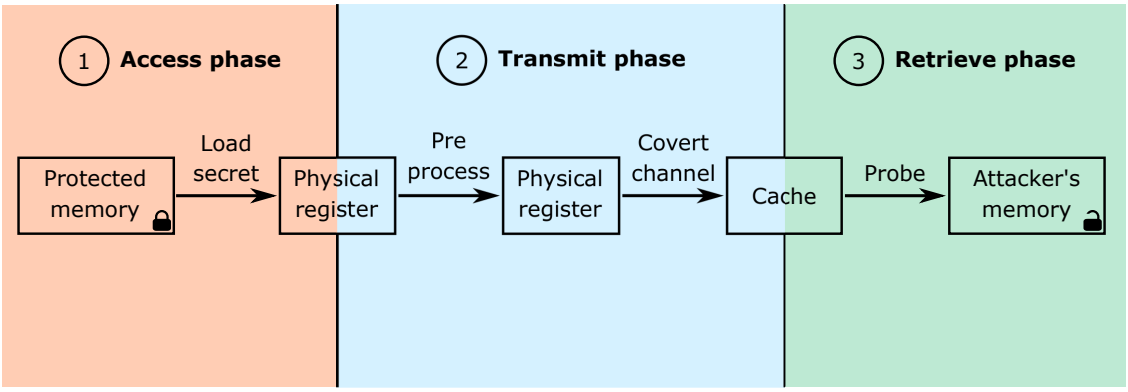

Figure 6.2: Three phases of speculative execution attacks

6.2.3 Formal Verification of Microprocessors with Out-of-Order Execution

A vast literature covers the formal verification of processors: techniques from automation-focused (yet poorly scalable) such as Model Checking [37, 175] to more complete and powerful ones such as Theorem Proving.

The major problem in the formal processor verification is the scalability. Processor designs usually tend to be very large with a huge number of states, thus leading to the problem of state explosion. Most techniques tackle down state explosion with model transformations:

- data abstraction: the model behavior is over-approximated by (partially) removing/transforming data (deemed) unnecessary to the proof; assumptions have to be made so that the soundness of the approach is guaranteed, e.g., arithmetic and logic functionalities are already verified. Possible refinement steps are needed, whenever the abstraction is unsound
- model reduction: a form of case split, where only a properly selected subset of possible execution traces is considered

Theorem proving seems to be the most common approach because of its robustness and flexibility. It is able to prove arbitrary processor configurations and it is a viable approach even for complex designs. However, Theorem proving techniques need substantial human efforts and tend to be labor intensive [176, 177].

Within the field of pipelined processor verification, theorem proving can be adapted and partially automated, thus leading to approaches that could even be considered to be generalizations of model checking.

For instance, the desired effect of transient (i.e., unfinished) instructions on the pipeline has been captured by complete functions, proposed in [178]. Complete functions can be seen as a map between a flushed state at the architectural level and any OoO processor state (i.e., microarchitectural). The idea of complete functions has been used to carry out independent compositional proofs that can be subsequently used as hints for other proofs. The authors in [179] give an example of this approach for the verification of a Tomasulo compliant OoO processor. The major drawback is the automation level: the verification engineer has to manually define a set of completion functions, one per unfinished instruction, and a way to compose them, in order to form the abstraction of the processor.

Burch and Dill [180] use a similar flushing technique. The abstract model of the processor is obtained by using what they call uninterpreted functions. The formal verification is then carried out by symbolic execution. They decompose the verification problem into three subproblems and require the user to provide some extra control inputs. This compositional approach solves the problem of state space explosion. Informally, the first property states that the implementation correctly executes instructions from a flushed state; the second confirms that stalling does not affect the specification state; and the third checks that instructions are fetched correctly. The main benefit of their contribution is that it can be applied without concrete implementation details. The other advantage is that it can be easily automated. The manual definition of special decision procedures for uninterpreted functions is still needed. Furthermore, Burch and Dill apply the technique only to in-order pipelined processors, without OoO support.

Skakkebaek et al. [181] present a two fold approach. The fist step is to derive an in-order abstraction model out of the OoO one. They define, using domain-specific knowledge, a functional equivalence relation between the in-order and the out-of-order models. The second step is the formal proof that two models are functionally equivalent by applying a technique called incremental flushing, based on the work of Burch and Dill. Although the proposed method effectively works for out-of-order processors, it heavily relies on human-guided theorem proving.

McMillan [182] verified Tomasulo's algorithm for out-of-order execution. The main drive of his work is the complexity reduction of the verification algorithm: a compositional approach is used to tackle the scalability issue. The major drawback of this methodology is that a good balance between state space explosion and the number of invariants must be manually/heuristically reached. Another issue is that it is not fully automatic.

Sajid et al. [183] extended and combined the BDD techniques with decision procedures for uninterpreted function symbols. Despite being a great contribution within the field of formal verification and model checking, this method cannot be easily integrated with symbolic model checkers. Moreover, they have not considered the application of their work on the verification of OoO processors.

Berezin et al. [184] incorporate uninterpreted function symbols into traditional model-checking and propose an scalable and effective propositional completion function for Tomasulo's algorithm. Their approach is able to verify Tomasulo's algorithm in any arbitrary configuration. Moreover, it can be extended for the automatic verification of complex generic parametrized designs.

6.2.4 Verifying Cybersecurity by Tainting

To the best of our knowledge, none of the above contributions is concerned with the verification of cybersecurity properties. Albeit formal verification has been effectively applied to cybersecurity, the main efforts have been devoted to the formalism and the definition of security properties or the mitigation of scalability issues by mixed dynamic/static techniques such as semi-formal and/or concolic (concrete symbolic) approaches [185, 186].

Information Flow Tracking (IFT) is a method that tracks propagation of information through a security-critical system. IFT has been extensively applied to security contexts for both hardware and software systems. In this context IFT has been proved to be a reliable technique to verify confidentiality and integrity. The core idea of IFT has been used for several formal and non formal approaches. Basically, IFT marks inputs from potentially malicious channels with labels. IFT then tracks how labeled data propagates through the system. For every instruction, the processor determines whether to propagate the label to the resulting destination according to a predefined set of rules. This way information flows become explicitly

visible to the verification process [130, 129].

IFT techniques have been implemented at different levels of abstraction: from hardware enforced IFT (both gate [129] and RTL [187] levels) up to modified kernels [188], VMs [132] and compilers [189].

Usually, in a IFT propagation scheme there are two actors involved (e.g., modules for hardware, variables for software systems). A general propagation rule states that the tag should flow from one object A to the object B if and only if a change in A affects the value of B (i.e., interference property).

Confidentiality and integrity can be expressed using this rule as follows:

- Confidentiality: object A is confidential, while object B is public. An attacker could gain information about A by observing variations on B. In this case, the property would be that A must not flow to B.
- Integrity: object A is untrusted, while object B is trusted. An attacker could gain access to B through malicious modifications on A. Again, the property would be that A must not flow to B.

A notable variant of IFT is Taint analysis. Security-critical data are *tainted*, i.e., marked as malicious or spurious, when they start from a target input (called *source*). The taint is then propagated through the system using the taint propagation rules and eventually reaches the target output (called *sink*).

Subramanyan et al. [136] propose the notion of taint-propagation properties, a novel type of security properties. Their approach is to create a model of the interactions between hardware and software by embedding information flow analysis with an instruction level abstraction (ILA). It is therefore best suited for hardware/-software co-verification as it is able to express security properties at the HW/SW boundary. Nevertheless, vulnerabilities at the microarchitectural level are not covered in their work.

Further notable research on adapting software taint analysis in the hardware/-software co-verification, in order to detect security vulnerabilities, has been pioneered in [50, 43, 187].

Secure Path Verification [107] represents an alternative to standard model-checking approaches. As already described in 5 we propose a new category of properties, namely Path properties, starting from the notion of the mentioned

taint-propagation properties. A path property includes a source, a destination and environmental constraints for taint propagation, same as the taint-propagation property, but it also introduces path constraints to enforce secure information flow properties. The contribution further introduces a verification engine based on equivalence checking, where taint propagation is embedded in the model checker engine.

As shown before, IFT can be applied at different levels of precision and abstraction and this can lead to different verification results in terms of soundness. The repercussions of this choice are discussed in Sections 6.5.1 and 6.5.3.

6.3 Processor Model

The processor we use in our methodology is based on the renowned Hennessy and Patterson's DLX architecture [154], a 32-bit generic RISC processor architecture that we enhanced by implementing pipelining and speculative execution.

The processor high-level design is shown in Figure 6.3. It employs a load/-store architecture and maintains a seven-stage pipeline. Support for speculative and out-of-order execution is provided by the reservation stations and the reorder buffer. The model is compliant with Tomasulo's algorithm [155]. The instructions are executed out of order rather than in sequential order, transient instructions are managed by the reorder buffer and the reservation stations. Wrong-path instructions (i.e., instructions that are mispeculated or violate access boundaries) raise misprediction exceptions, causing the CPU to stop the execution and the pipeline to be flushed. In case of exception, the instruction sequence is squashed and any microarchitectural state change after the exception is not committed to the architectural state. Data and control flow dependencies (e.g., instruction cache miss) are handled by stalling the pipeline until the dependency is resolved.

The pipelined processor model has 7 stages: **[IF1]** A 32-bit instruction is fetched into an instruction fetch buffer from the memory address given by the program counter PC (branch prediction logic is included).
[IF2] This is meant to be a virtual address translation unit, possibly coupled with a TLB, but in this case study it is just a combinational delay. We retain this stage for future work.
[ID] Instruction Decoding, fetching of values from registers, evaluation of branch

conditionals and target addresses, pipeline hazard detection and control flow of program execution. Opcodes and operands are fed to the Reorder Buffer (ROB); Inside the reservations stations transient instructions wait for the availability of their operands.
[**EX**] When ready instructions are issued to an available execution unit. The EX stage executes both integer and floating-point instructions and generates exceptions as needed. This stage includes reservations stations, providing operands to and acting as a scheduler for execution units.
[**M1**] access to data memory for load/store instructions. This stage enables a bypass towards the ROB, the result of the instruction is routed to reorder buffer through the common data bus (CDB).
[**M2**] access to data memory for load/store instructions. This stage is the actual memory write.
[**WB**] instruction commit, exception handling and storing values to the register file. This includes the ROB that retires instructions to the register file. The ROB writes the results to the register file restoring the original program order. The execution results are also forwarded through the CDB to the reservation stations, where other instructions may wait these results as operands.

Exceptions are raised when an instruction, for some reasons, cannot be executed. Our model includes three types of exceptions: ALU exceptions, mispredicted exceptions and memory exceptions. When the CPU receives an exception, executes all the instructions preceding the interrupted instruction, stops the execution, flushes the pipeline and starts the exception handler.

Without loss of correctness and for the sake of simplicity, we treat the execution units as fully pipelined, that is they execute one instruction every clock cycle. Instructions follow the same behavior: every instruction takes the same number of clock cycles to be executed, even though we are fully aware that this is not true in a real microprocessor.

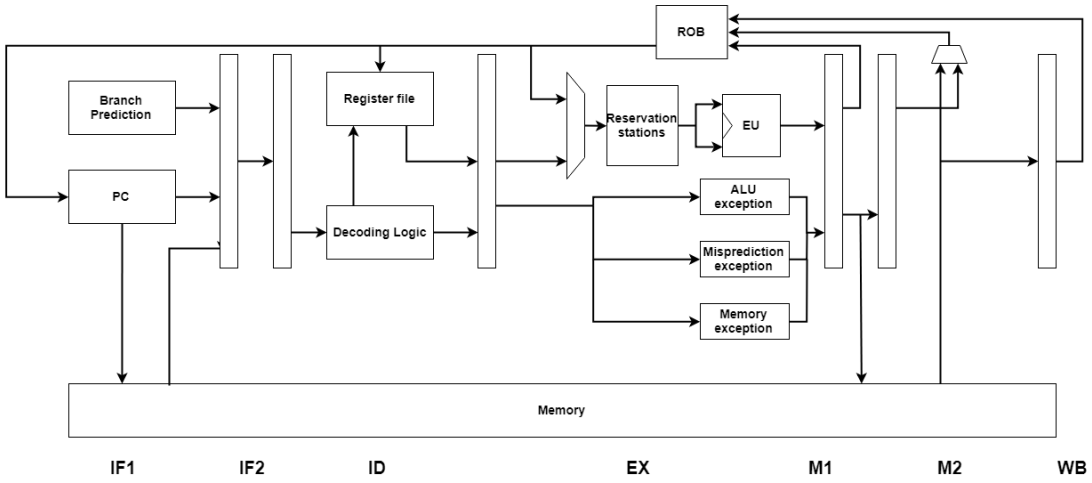

Figure 6.3: The 7-stage pipelined processor model

6.4 Attack Description

To show the weakness of CPU architectures, we decided to replicate the well-known Spectre [17] and Meltdown [16] attacks, which both abuse the out-of-order execution to disclose internal microarchitectural state information.

Both cases are based on side-channel attacks. A side-channel attack is any attack based on information gained from the implementation of a computer system, rather than the implemented algorithm itself or the Instruction Set Architecture. Side-channel attacks typically exploit timing, power, electromagnetic leaks, or other sources of information.

In the case of Spectre and Meltdown, the attack is based on a covert side-channel that exploits the timing of cache-based memory accesses. A malicious attacker intentionally induces a speculated execution of mispredicted instructions. The effects of these instructions (not yet retired, thus called transient) are not intended to be committed into the architectural state, as the instructions will be cancelled and their results discarded. However, they change the internal processor microarchitectural state, due to a memory read affecting the cache state, observable by a (timing-based) side-channel: retrieving data from (un-cached) memory takes longer than retrieving it from cached addresses.

A Meltdown attack provides a way for a user level attacker to read the entire kernel space memory, including all physical memory mapped in the kernel, being

able to bypass the privileged-mode isolation. A Spectre attack has a more limited scope, the memory within the address space of another (victim) process.

Although different, both attacks rely on common "building blocks". The first consists of executing one or more instructions which would never be computed in the proper execution path. Applying Meltdown's naming convention, we call transient instruction an out-of-order executed instruction, which will leave measurable side effects. We also call transient instruction sequence any sequence of instructions which contains at least one transient instruction. To be able to exploit transient instructions to perform an attack, the sequence needs to hold a secret value which will be leaked by the attacker.

The second building block consists of transferring the architectural state affected by the transient instruction sequence to propagate and extract the leaked information.

From a high-level perspective, our attack, based on Meltdown, consists of 3 steps:

1. the attacker chooses an inaccessible memory location, then the contents of that memory location is loaded into a register;

2. a cache line is accessed by a transient instruction based on the secret contents of the register;

3. the attacker exploits a side-channel to probe the previously accessed cache line and leaks information depending on the sensitive data saved at the chosen memory location.

Listing 1, written in DLX assembly code, presents the basic implementation of the transient instruction sequence of our attack that is then used to leak information with a side-channel. We describe now in detail the actions performed and the effect of every single step of our attack, considering the presented assembly code but also the processor model under analysis.

Listing 1. Attack transient instruction sequence in DLX assembly code.

```
1 ; R1 = invalid address
2 ; R3 = probe array
3 LW  R2, 0(R1)
4 ADD R4, R2, R3
5 LW  R1, 0(R4)
```

6.4.1 Step One

Referring to line 3 of Listing 1, `R1` register holds the invalid address, i.e., an attacker chosen target kernel address that is loaded into register `R2`.

The `LW` instruction is fetched, decoded into μOPs (Although this is true in general, in our case-study model there is no generation of micro-operations, i.e., every instruction is one μOP, due to the simplicity of the model itself.), allocated, and sent to the reorder buffer, waiting to be executed.

At this point, in order to enable out-of-order execution, register renaming maps the architectural registers (e.g., `R1`, `R2`, `R3` and `R4` in Listing 1) to underlying physical registers.

As for out-of-order execution, that aims at increasing CPU throughput, also the following instructions (lines 4 and 5) have already been fetched, decoded and allocated as μOPs, they are then sent on hold to the reservation stations, waiting to be issued to an execution unit.

In general, a μOP is delayed if operands are not available or if all execution units are already holding and executing other μOPs. In our example, the `ADD` instruction must wait for the result of the first `LW` instruction.

When the contents at the kernel address is loaded (line 3), subsequent instructions have already been issued in the reservation stations as speculated instructions, waiting for the kernel address to be available.

Once the needed data are fetched and available on the Common Data Bus (being accessible to the execution unit and being stored into registers), μOPs on hold can be executed. After the execution stage, completed μOPs are committed in-order (write-back stage), with their results affecting the architectural state of the CPU.

If any interrupt or exception, e.g., illegal access, arise during the execution stage, the corresponding exception unit handles it.

Therefore, when the `LW` instruction is retired, if an exception is thrown, then the pipeline is flushed to discard all the computations of the following speculated instructions.

At this point, there exists a race condition between raising an exception and the speculative execution of our attack, which is described in detail in Section 6.4.2 below.

6.4.2 Step Two

The set of instructions executed during Step One must be carefully crafted to make a transient instruction sequence.

To transmit the secret outside the CPU microarchitecture, first we define and allocate in memory a probe array, ensuring that no part of it will be cached. The original Meltdown attack prevents the hardware prefetcher from loading adjacent memory locations into the cache. For our purposes, in order to make our attack model as generic as possible, we avoid this operation.

Then, an address is determined, based on the (inaccessible) secret value, and it will be used by the transient instruction sequence to perform an indirect memory access.

The secret value is added to the probe array base address, in order to compose the target address of the covert channel. The target address is then read and used to store the corresponding cache line.

Therefore, the goal of our crafted instruction sequence is to alter the cache state, based on the secret value read during Step 1.

6.4.3 Step Three

In Step 3, the attacker leaks and obtains the secret value (from Step One) exploiting a microarchitectural side-channel attack which propagates the cache state (from Step 2) into an architectural state.

This is due to the race condition mentioned in Step One: line 3 gets speculatively executed, depending on the value of the secret, so a part of the probe array will be

accessed and, most importantly, cached.

To sum up, despite the program never executing lines 4 and 5, the cache state has changed.

As already stated, Meltdown and Spectre exploit well-known cache covert channel attacks but their specifics are considered out of scope.

6.5 Proof/Verification

To simplify the verification process, in our approach we propose several model transformations, oriented to scalability. The performed transformations can be aggregated into two different typologies:

- data abstraction: register values, along with all the units that handle these values, as well as the reorder buffer, reservation stations and execution units, are adequately abstracted. Tainting information and evaluation/propagation circuitry are then added to or replace them.
- model reduction: applying established processor verification approaches such as pipeline flushing and reduction by refinement map, we reduce speculation and parallel execution logic.

Considering that we strive to make our approach as general as possible, our CPU model does not explicitly feature any main memory, therefore confidential/secret information is implicitly linked to a given *protected/invalid* memory address. Consequently, we verify arbitrary sequences of instructions (i.e., no explicit program is provided).

The completeness of our approach, even though this is a pessimistic choice in the framework of our analysis, is guaranteed because all possible instructions sequences are covered, as well as being consistent with other state-of-the-art processor verification approaches.

In the following subsections we provide a detailed description of the main notions of the two typologies of transformations. Albeit a formal proof of correctness is not provided, we contribute with the basic intuitions that support their applicability.

6.5.1 Data Abstraction and Tainting

All related works cited in Section 6.2 follow the same verification approach: they over-approximate data (and consequently model behavior), provided that verification is sound. That is, this transformation must guarantee that an abstract counterexample always implies a concrete one.

In our case, for every register R_i we call V_i its contents. Then, without altering security properties, V_i is replaced by an abstract value V_i^+. As already stated, RAM memory is removed, adopting common strategies used for data abstraction, which means drastically reducing its impact on the state of the model. Partial abstraction of data, not relevant to the property under verification, is often performed. Despite having greatly reduced the size of the model, we can still perform abstraction: for example, processor functionality (e.g., ALU data computation/evaluation) can be considered correct (already verified). In all abstraction processes, properly choosing a correct/adequate abstraction, either by automated or manual selection, is a challenging task.

Following known methods in hardware verification, we augment each value V_i^+ with its corresponding taint value T_i, as depicted in Table 6.1. The ALU, the data evaluation logic, is enhanced with taint-propagation logic. Taint values are injected, propagated and observed through memory and data transfer components. Propagation or combination of multiple taints is responsibility of the ALU.

Table 6.1: Concrete versus Abstract model transformation.

Register	**Concrete**	**Abstract**
$\mathbf{R_1}$	V_1	(V_1^+, T_1)
$\mathbf{R_2}$	V_2	(V_2^+, T_2)
$\vdots$	$\vdots$	$\vdots$
$\mathbf{R_n}$	V_n	(V_n^+, T_n)

Table 6.2 shows that while the original processor model executes an arithmetic/logic operation $V_k = OP(V_i, V_j)$, the abstract model replaces it with $V_k^+ = OP(V_i^+, V_j^+)$ and provides that the transformation is sound. The first part is data

evaluation and it is independent from taint values, while the second is taint propagation, which in its broad form involves both data and taint values.

The tainting precision, i.e., the degree of over-approximation, swings between two corner cases:

- full data dependence: data values are fully involved in taint computation; whenever computing T_k, actual operand data values are considered; for instance, a bitwise OR operation with all $V_i^+ = 1$, or a multiplication with $V_i^+ = 0$, could mask (block) a taint on the other operand (T_j);
- full abstraction from data values: for instance, taint propagation through a binary ALU operation propagates a taint on any of the operand terms ($T_k = OP^T(T_i, T_j) = T_i \vee T_j$).

The level of abstraction certainly affects the soundness of the overall approach: for detailed comments and proof of correctness of the approach, see Section 6.5.3.

Table 6.2: Comparison of data evaluation and taint propagation between concrete and abstract models.

	Concrete	**Abstract**
Data abstraction + Tainting	V_i	(V_i^+, T_i)
Data evaluation	$V_k = OP(V_i, V_j)$	$V_k^+ = OP(V_i^+, V_j^+)$
Taint propagation	-	$T_k = OP^T(V_i^+, T_i, V_j^+, T_j)$

Also, the branch misprediction logic is abstracted and replaced by a non-deterministic value (choice); this operation can be performed without altering the correctness of our approach, since it generalizes (abstracts) the model under analysis.

Similarly, arithmetic/logic manipulation can be considered already verified, e.g., as pass-through circuitry for taints. Since we are interested only in data leakages from/to memory, we can move our focus taint propagation to memory access logic.

Let us now define the taint source/sink pair. A taint is thus injected at the memory data input, whenever a protected/invalid address is used, that is a protected address is loaded into the CDB. In other words, the taint (T_{MDR}) associated with the Memory Data Register (MDR) is set whenever the address stored in the

Memory Address Register (MAR) is not in the range of valid addresses ($VA.start$ and $VA.end$).

$$T_{MDR} = MAR < VA.start \vee MAR > VA.end$$

The taint is propagated through the abstract reservation stations, arithmetic/-logic execution units and reorder buffer. A taint can be cleared only by the branch misprediction circuitry, which means in case of an instruction not committed but aborted.

The target (taint sink) under observation is the memory address. From the information flow point of view, a taint in a memory address register corresponds to the property we are verifying.

$$Prop = \neg(T_{MAR} == TAINT)$$

Considering, from a tainting point of view, a generic instruction sequence common to both Meltdown and Spectre attacks, the following actions will be performed:

- inject a taint, exploiting a mispredicted instruction which performs an invalid/protected memory access;
- transfer the taint into the (abstracted) reorder buffer, as a data expected to be committed;
- use the taint, as a data, as part of the computation of the address of a successive mispredicted memory access: the taint hits the target.

The taint propagation path is described in Figure 6.4. In details, the taint is injected at the memory read (**1**), when the address bus is filled in with an invalid address. As the instructions in the pipeline move through the different steps, the taint reaches the reorder buffer (**2**), then it moves on to reservation stations (**3**) as new instructions are fed by the instruction cache. The taint enters reservation stations because of the propagation rules, as tainted data are fetched as operands of newly arrived instructions. As soon as all the operands are available, the taint propagates to the corresponding execution unit (**4**). Eventually it reaches the Common Data Bus (CDB) and consequently again the reorder buffer (**5**).

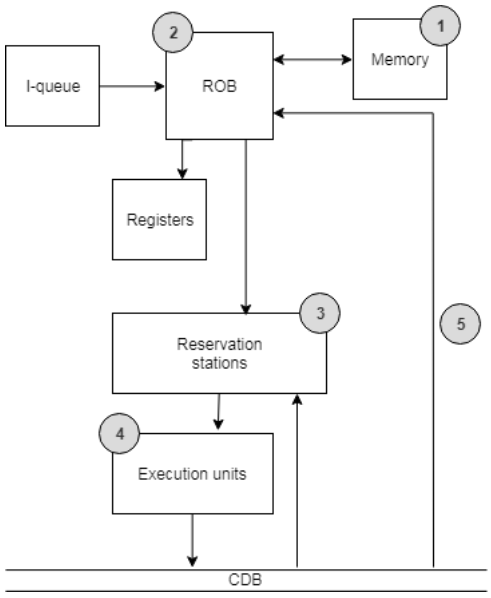

Figure 6.4: Taint propagation from source (memory read) to sink (reorder buffer) in our abstract model.

Table 6.3 shows how our model behaves with the instructions in Listing 1. The first instruction reads from memory (M) at an invalid address (IA) and stores an invalid data (ID) into the destination. This injects the taint into T_2. The second instruction computes the array displacement as it adds the probe array start address (PA) and the invalid data. The taint propagation logic, as previously stated, involves the OR operator on the taints associated with the operands, so in this case the taint propagates to T_4. The last instruction reads in from memory at the address computed in the second instruction. The taint propagates to T_1 and reaches the target where the property fails.

Table 6.3: Our model applied to a real use case: a basic implementation of Spectre/Meltdown.

Instruction	**Symbolic**	**Concrete**
LW $R2, 0(R1)$	$V_2^+ \leftarrow M[V_1^+]$ $T_2 \leftarrow T_1$	$ID \leftarrow M[IA]$ $1 \xleftarrow{inject} 0$
ADD $R4, R2, R3$	$V_4^+ \leftarrow ADD(V_2^+, V_3^+)$ $T_4 \leftarrow T_2 \vee T_3$	$PA + ID \leftarrow ADD(ID, PA)$ $1 \leftarrow 1 \vee 0$
LW $R1, 0(R4)$	$V_1^+ \leftarrow M[V_4^+]$ $T_1 \leftarrow T_4$	$Y \leftarrow M[PA + ID]$ $1 \xleftarrow{target} 1$

Table 6.4 shows, in the rightmost table, the same instruction sequence as in Listing 1, while the leftmost table displays the pipeline evolution. The sequence implements a generic and simplified version of Meltdown/Spectre attack (as described in Section 6.4), in which:

(A) performs a memory access to an invalid address;

(B) executes an arithmetic operation using the secret data;

(C) performs a read from an array with a displacement related to the secret.

Table 6.4: Pipeline evolution of the proposed attack.

Clock ↓	**IF**	**ID**	**EX**	**MEM**	**WB**	**Taint Status**
1	A					
2	B	A				
3	C	B	A			
4		B/C	S	A		Taint source
5		C	B	S	A	Taint in ROB
6		C	S	B	S	Taint in EX
7			C	S	B	Taint sink/property asserted
8				C	S	
9					C	

A	**LW**	$R2, 0(R1)$
B	**ADD**	$R4, R2, R3$
C	**LW**	$R1, 0(R4)$

The vertical axis represents time in terms of clock cycles. Columns from IF to WB represent the different pipeline stages. The last column presents the taint status and its propagation. Cells filled with "S" are stalls.

Instructions enter the pipeline and proceed without stalls until clock 3. At clock 4, B and C are stalled to wait for their operands to be ready. When at stage MEM A reads from memory, the value is placed into the Common Data Bus and becomes available also to the stalled instructions. At clock 5, B goes on to the next stage, while C must wait for its operands to be prepared by B, thus a new stall is inserted. Finally, from clock 7 to 9 the pipeline proceeds without stalls.

The taint source is at clock 4 inside the MEM stage of instruction A, when the access to the invalid address is performed. At the WB stage of A, the taint propagates to the reorder buffer. Then the taint propagates to the execution unit at the EX stage of instruction B. Afterwards during clock 7 the taint, reaching the CDB, is collected by the taint sink and captured by the previously defined security property.

Compositional verification schemes are a very common way to exploit abstraction, where a given module is taken and verified on a local basis, while removing (i.e., abstracting away) the remaining model components, considered to be the module's environment. A refinement of the environment, expressed as a set of constraints, is usually required to refine it to a sound abstraction, thus leading to assume-guarantee strategies where a given refinement is either a property (to be proved/guaranteed) when verifying a module, and an environment constraint (an assumption) for another module under verification.

Considering our case study, seeing the high simplification level reached by abstraction/refinement-based abstraction, we decided not to apply any compositional simplification [190].

6.5.2 Combining Model Reduction with Abstraction

The CPU model of our case study, at this point, is further simplified, applying state-of-the-art methodologies to formal verification of pipelines and speculation units [180, 191, 190].

Briefly, the processor model was already simplified by transforming data into taints. At this point instead, we are simplifying all intermediate states associated with the control logic for parallel execution, with a convenient reduction of the behavior and considering it as an abstraction, based on proper equivalence notions between the concrete model and the reduced one.

Pipeline flushing is often applied, in order to highly simplify model behavior, by removing all possible interleavings of pipeline executions. As this could lead to an incomplete solution (missing possible wrong behaviors) the reduced model is considered to be an abstraction of the real model, where each concrete state (among the set of all possible pipeline interleavings) is considered to be a possible refinement of an abstract state in the reduced model: the correspondence is handled by refinement maps. Similar strategies are also applied whenever reducing the bit widths of registers, memory words and of memory addresses, the size of the register file, the number of reservation stations and execution units, etc. All such reductions are deemed as complete by considering the reduced model as an abstraction of the concrete (refined) model and/or by proving that the absence of a bug in the reduced model implies that no bugs are possible in the concrete (non-reduced) one.

While considerably simplifying the model, a reduction process can thus still guarantee verification completeness.

A detailed description of the reduction strategies is clearly out of our scope in this work, as any property preserving reduction (with proper refinement map or alternative theory) is applicable, provided that it guarantees completeness.

We here quickly introduce the simplifications applied on our model:

- pipeline flushing: all pipeline stages are flushed (collapsed), which simplifies the execution model of an instruction, since the next instruction is initiated only when the previous one reaches the reorder buffer;
- reorder buffer removal: ROB is reduced into a FIFO queue, which essentially delays instructions between execution and results availability into the register file; to be noted that the FIFO strategy preserves the original instruction order, assuring data dependency;
- reservation station replacement: as straightforward effect of previous pipeline flushing and ROB removal, reservation stations are bypassed (performing in fact a model reduction);
- execution units merge: considering the data abstraction performed on our model, computation parallelism is unnecessary, so just one instance of each execution unit is useful.

As a result, of the listed above simplifications, the complete original behavior is significantly reduced, moving from a pipelined architecture with speculation, to a fully sequential model with a FIFO-based delay between execution and obtainable results.

The performed reduction simplifications guarantee completeness, which means that taint propagation sets of instructions are not removed, given the following two conditions:

- consecutive instruction sequences comprise mispredicted instructions, simulating real instructions sequences made by an actual out-of-order CPU; this behavior is assured by a non-deterministic tag associated with an instruction, which marks that instruction as mispredicted (this operation is performed during the abstraction transformation, as described in Section 6.5.1);
- the FIFO-based delay, replacing the ROB, simulates the (illegal) taint-propagation time from source to sink; this behavior is assured by a proper FIFO queue size and a proper non-deterministic queue control of *get* operations, which transfer data from ROB to the register file.

6.5.3 Correctness of the Approach

We now provide a concise proof of the correctness of our verification approach. Due to the descriptive nature of the work, where we omit a rigorous formalism for models, properties and transformation steps, the proof is limited to a sketch, outlining the theoretical bases that support the proposed methodology.

As with all verification approaches based on model transformations, formal correctness means *soundness and completeness* of all model simplifications (abstractions and reductions) performed. In our case, we operate two classes of simplifications:

- Abstractions and reductions that do not affect secure information flow by taint propagation. This is a set of preliminary model transformations oriented to reduce the data width and the model behavior (pipeline and speculative execution): transformations guarantee the model functionality, and they have already been proved correct by related and state-of-the-art works on formal

verification of processor designs. We do not claim any contribution in this field, and we assume them as correct

- Abstraction and reduction steps related to secure information flow by taint propagation. Though we resort to standard formalisms and transformations, we nevertheless need to show/prove that their combined application is complete and sound.

Model Abstraction and Reduction

As already written, our model abstraction and reduction (with refinement) steps are based on state-of-the-art formal verification approaches [180, 191], that have already been proved to be sound and complete by their authors. We can assume them as correct in terms of processor design, but nothing is said on the processor state, potentially observed by a side-channel attack. In other words, we exploit state-of-the-art model simplifications that guarantee model functionality by pruning and simplifying complex intermediate states and behaviors.

Therefore, the remaining critical issue in our methodology is the correctness of taint encoding and manipulation all over the abstraction and refinement steps. This is what was missing in all previous processor verification works, and represents the main contribution of our work.

To this respect, we need to show that

- tainting does not affect the correctness of the model
- secure information flow by tainting is sound and complete.

The first item is straightforward. We can claim that since a taint is actually an enhanced data, added to the original one, it does not affect model functionality, provided that tainting logic is just *reading* data, while not affecting data evaluation. More formally, this is clearly shown in Table 6.2, where a result taint (in the more general case) is a function of operand data and taints ($T_k = OP^T(V_i^+, T_i, V_j^+, T_j)$), whereas result data do not depend on operand taints ($V_k^+ = OP(V_i^+, V_j^+)$). The second item (soundness and completeness of taint propagation) is analyzed in the next subsection.

Taint Encoding and Manipulation

In the strict sense, the tainting and transformation steps we perform are unsound as, due to data abstraction in the taint-propagation circuitry, we could obtain false negatives (taint-propagation traces not feasible on the concrete model).

In fact, we admit abstract execution traces that propagate the taint through the ALU logic, whereas no information leakage would characterize the actual model.

This issue can be imputed to the taint-propagation strategy we decided to adopt: the applied abstraction ignores the fact that a taint could be blocked/hidden by a real data. The taint computation rules need to take into account both the operation and the data involved. Precise rules would impose more complexity, so we had to reach a trade-off between precision and complexity:

- apply precise and narrower tainting rules, cutting off (adopting a more precise and detailed taint-propagation model) all false negatives;
- adopt an imprecise but efficient approach, thus accepting false abstract counterexamples. If this would be the case, counterexamples could be either:
  - post processed, leading to subsequent model refinements;
  - be converted into actual (equivalent) concrete counterexamples, by just exploiting them partially (e.g., by removing data and keeping control bits), as constraints for a further Bounded Model-Checking (BMC) run on the concrete model.

In our opinion the second approach is to a great extent coherent with the final aim of detecting data leakages and solving them.

Therefore, in the end soundness relies on a proper notion of (bi-simulation) equivalence between abstract and concrete counterexamples: any abstract counterexample is required to be mapped to at least one concrete counterexample by simple data/behavior refinement: Abstractions done in taint-propagation logic are sound if, for any tainting blocking based on data values, other non-blocking data exist.

The approach is complete as no reduction is done on taint computation and

propagation (all reductions are done just based on equivalence/function preserving transformations, whose completeness has been proved for model-checking purposes).

6.6 Experimental Results

The approach presented in this chapter was tested and verified on the case study described in Section 6.3.

The main purpose of our experimentation was not to provide detailed performance measures of different model-checking engines/tools, rather to show that resorting to proper abstractions and reductions allows tackling state explosion. This makes previously unfeasible problems now solvable in matter of seconds with a state-of-the-art model checker.

The processor was described in Verilog, then converted into the AIGER format [192] and verified using PdTRAV [72], a state-of-the-art academic model-checking tool we developed. Both Bounded and Unbounded Model-Checking (interpolation-based UMC) algorithms were used, with a peculiar focus on model reductions and transformations [193, 194], multiple properties manipulations [195] and interpolants-based engines [78, 77].

In detail, taints were encoded as binary data, branch prediction/misprediction circuitry was entirely abstracted and substituted by a non-deterministic (random) Boolean value. Moreover, parallel execution units were substituted by a taint propagation *pass-through* circuitry.

A taint not reaching the address output of the microprocessor model corresponds to the encoding of the confidentiality property we want to verify.

As already stated in [190], the original full microprocessor model (inclusive of speculation logic, Tomasulo's module, pipeline, multiple execution units and data paths) would be very difficult to verify: the model consisted of more than 120 K gates and more than 3 K latches. The resulting model after all the simplifications (abstraction, reduction and tainting), was converted into an AIGER file consisting of 2724 AND gates and 106 latches, resulting by reducing the register file to 8 registers of 5 bits, and the rest of the control and data logic accordingly. We also tested larger versions, by expanding the number of data registers up to 16

and their size up to 32 bits. The largest AIGER file (after Cone-Of-Influence reduction) included 7245 AND gates and 395 latches. We could verify it by the BMC verification engine, finding a counterexample of 11 clock steps in times ranging from less than 1 s to 9 s.

The counterexample retrieved after the verification process presented a data leakage issue: this bug was made out of an instruction sequence starting with an invalid memory read (operation which injected the taint), followed by an arithmetic computation of a memory address (operation which propagated the taint), and finally a further memory read to the tainted address. This counterexample can be considered to be an abstracted/reduced version of the example presented in Table 6.3.

Then the cybersecurity flaw was eliminated (and formally verified by model-checking, again in less than 1 s and 15 s, with both a Bounded Model-Checking and an IC3 engine) by patching the model in this way: all speculated instructions with data dependencies are prevented by an instruction with invalid memory access. More in detail, designing a non-buggy (efficient) version of the processor was out of our scope, we rather wanted to have a working non-buggy model. We thus exploited the exception generation logic, making it active before committing the exception generation instruction: whenever an invalid read is done by a speculated instruction, an exception is not generated until confirming (committing) the instruction (due to branch prediction logic). The exception detection logic immediately catches the invalid access, then it blocks the read data (thus not available for subsequent operations) until the instruction is committed or aborted: if committed, the exception is raised, if aborted, execution resumes on the correct path.

6.7 Conclusions and Future Work

In this chapter, our goal is to describe a formal verification procedure able to find confidentiality security leaks in contemporary CPU microarchitectures featuring out-of-order/speculative execution, in order to prevent future cybersecurity speculation-based attacks (as Meltdown or Spectre).

The proposed technique is based on state-of-the-art formal verification concepts, as model abstraction and model refinement, as well as exploiting novel ideas from

the information flow tracking field, as taint injection and taint propagation, merging schemes taken from this two different areas of research to reach our goals.

In future works we will automate the simplifications performed to the model under analysis (abstraction/reduction), since in this chapter the described transformations were manually driven. Our final goal will be an automated (eventually in part) transformation and simplification mechanism.

Chapter 7

Conclusions

In this thesis, we have proposed our work on extending the state of the art with respect to verification of security-relevant properties in embedded systems. We selected the most common class of such systems, namely Cyber-Physical Systems (CPSs), that are networked environments of mingled hardware/software subsystems. The majority of CPSs are both safety- and security-critical since they interact with our daily lives. Therefore our primary security objectives are confidentiality, integrity and availability. Keeping those goals in minds, we proposed new verification approaches that we demonstrated viable and sound. As the growing size of CPSs leads to the well-known state space problem while verifying large designs, we then presented reduction and abstraction techniques to tackle this verification issue.

The first contribution, described in Chapter 3, we performed abstraction and refinement steps on two prominent representatives of Remote Attestation architectures: SMART and SANCUS. We showed how to express security requirements as Taint properties. We then extracted, from the two abstract models, a portfolio of such properties. Model checking results showed that our formal specification and verification method for security properties in remote attestation architectures is feasible.

An extension of taint properties is presented in Chapter 4. Path properties are a new class of properties that are able to capture security vulnerabilities concerning the flow of data propagating from an attacker-controlled input source to selected

sink, i.e., they cover a path from one entity source to an entity destination. For example, the source may be a potentially malicious (i.e., untrusted) RTL (Register Transfer Level) register or signal and the destination a sensitive RTL output signal that we do not want to be reached by confidential data. This is the case of a confidentiality requirement. As use cases we chose the same targets as in Chapter 3 and we defined a portfolio of Path properties. We employed model checking techniques to verify them and we presented a brief comparison between Path and Taint properties.

We performed a thorough analysis and comparison of Path properties with respect to standard state-related properties in Chapter 5. We redefined the second class State properties as they are meant to express state-related properties, in contrast to information flow-related properties such as Path properties. In particular, starting from a set of security requirements, we defined a portfolio of properties for each class. We showed advantages and disadvantages of the two classes: in particular, when Path Properties perform better than State Properties and vice versa (e.g., for coverage results or for expressive power). We also provided the implementation support to integrate Path properties within the framework of a standard model checker. The comparison was further carried out by an experimental point of view: they supported the claim of our approach.

The last contribution, presented in Chapter 6, a novel formal verification approach for security properties in pipelined out-of-order processors. The recently emerged speculative execution attacks such as Spectre and Meltdown have sparked concerns in the security community and even in the common public. Common microprocessor models are large designs and thus suffer from scalability issues. We presented an abstraction and reduction-based approach to produce an abstract processor model that is computationally feasible with a standard model checker. Experimental results supported our claim on scalability and they further demonstrated that it is a valid security verification method. The bug was indeed exposed and the model was therefore patched. A subsequent model checker run showed no sign of the bug.

Though preliminary, our results have been proved to be viable approaches in the security verification field. They can be used in conjunction with other techniques in order to improve the efficiency and coverage. Our focus is mainly towards hardware/software co-design, especially at the HW/SW boundary, but in this thesis we offered insights from other works that the same concepts can be developed and used also in other applications.

Bibliography

[1] Stefano Zanero. "Cyber-physical systems". In: *Computer* 50.4 (2017), pp. 14–16.

[2] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. "Modeling cyber–physical systems". In: *Proceedings of the IEEE* 100.1 (2011), pp. 13–28.

[3] László Monostori et al. "Cyber-physical systems in manufacturing". In: *Cirp Annals* 65.2 (2016), pp. 621–641.

[4] David Grimsman et al. "A case study of a systematic attack design method for critical infrastructure cyber-physical systems". In: *2016 American Control Conference (ACC)*. IEEE. 2016, pp. 296–301.

[5] Meiyuan Zhao, Jesse Walker, and Chieh-Chih Wang. "Challenges and opportunities for securing intelligent transportation system". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 3.1 (2013), pp. 96–105.

[6] Manuel Cheminod, Luca Durante, and Adriano Valenzano. "Review of security issues in industrial networks". In: *IEEE transactions on industrial informatics* 9.1 (2012), pp. 277–293.

[7] Yoshihiro Hashimoto et al. "Safety securing approach against cyber-attacks for process control system". In: *Computers & Chemical Engineering* 57 (2013), pp. 181–186.

[8] Flavia C Delicato et al. *Smart Cyber–Physical Systems: Toward Pervasive Intelligence systems*. 2020.

[9] Borja Bordel et al. "Cyber–physical systems: Extending pervasive sensing from control theory to the Internet of Things". In: *Pervasive and mobile computing* 40 (2017), pp. 156–184.

[10] Thomas Kropf. *Introduction to formal hardware verification*. Springer Science & Business Media, 2013.

[11] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[12] James Andrew Lewis. *Assessing the risks of cyber terrorism, cyber war and other cyber threats*. Center for Strategic & International Studies Washington, DC, 2002.

[13] Richard A Epstein and Thomas P Brown. "Cybersecurity in the Payment Card Industry". In: *The University of Chicago Law Review* 75.1 (2008), pp. 203–223.

[14] Ralph Langner. "Stuxnet: Dissecting a cyberwarfare weapon". In: *IEEE Security & Privacy* 9.3 (2011), pp. 49–51.

[15] Charlie Miller and Chris Valasek. "Remote exploitation of an unaltered passenger vehicle". In: *Black Hat USA* 2015 (2015), p. 91.

[16] Moritz Lipp et al. "Meltdown: Reading kernel memory from user space". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 973–990.

[17] Paul Kocher et al. "Spectre attacks: Exploiting speculative execution". In: *arXiv preprint arXiv:1801.01203* (2018).

[18] Thomas J Holt and Adam M Bossler. "An assessment of the current state of cybercrime scholarship". In: *Deviant Behavior* 35.1 (2014), pp. 20–40.

[19] Julian Jang-Jaccard and Surya Nepal. "A survey of emerging threats in cybersecurity". In: *Journal of Computer and System Sciences* 80.5 (2014), pp. 973–993.

[20] Johannes Geismann, Christopher Gerking, and Eric Bodden. "Towards ensuring security by design in cyber-physical systems engineering processes". In: *Proceedings of the 2018 International Conference on Software and System Process*. 2018, pp. 123–127.

[21] Alvaro Cardenas et al. "Challenges for securing cyber physical systems". In: *Workshop on future directions in cyber-physical systems security*. Vol. 5. 1. 2009.

[22] Roberto Vigo. "Availability by design: a complementary approach to denial-of-service". In: (2015).

[23] Ruggero Lanotte et al. "A formal approach to cyber-physical attacks". In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE. 2017, pp. 436–450.

[24] Mike Burmester, Emmanouil Magkos, and Vassilis Chrissikopoulos. "Modeling security in cyber–physical systems". In: *International journal of critical infrastructure protection* 5.3-4 (2012), pp. 118–126.

[25] Luis F Cómbita et al. "Response and reconfiguration of cyber-physical control systems: A survey". In: *2015 IEEE 2nd Colombian Conference on Automatic Control (CCAC)*. IEEE. 2015, pp. 1–6.

[26] Muhammad Usman Sanwal and Osman Hasan. "Formal verification of cyber-physical systems: coping with continuous elements". In: *International Conference on Computational Science and Its Applications*. Springer. 2013, pp. 358–371.

[27] Edmund M Clarke and Paolo Zuliani. "Statistical model checking for cyber-physical systems". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2011, pp. 1–12.

[28] Ravi Akella and Bruce M McMillin. "Model-checking BNDC properties in cyber-physical systems". In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. Vol. 1. IEEE. 2009, pp. 660–663.

[29] Yan Sun et al. "Verifying noninterference in a cyber-physical system the advanced electric power grid". In: *Seventh International Conference on Quality Software (QSIC 2007)*. IEEE. 2007, pp. 363–369.

[30] Michael Howard and Steve Lipner. *The security development lifecycle*. Vol. 8. Microsoft Press Redmond, 2006.

[31] Rolf Drechsler et al. *Advanced formal verification*. Vol. 122. Springer, 2004.

[32] William K Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series)*. Prentice Hall PTR, 2005.

[33] Jonathan Bowen and Victoria Stavridou. "Safety-critical systems, formal methods and standards". In: *Software Engineering Journal* 8.4 (1993), pp. 189–209.

[34] Bernd Finkbeiner, Markus N Rabe, and César Sánchez. "Algorithms for model checking HyperLTL and HyperCTL*". In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 30–48.

[35] Michael R Clarkson and Fred B Schneider. "Hyperproperties". In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[36] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE. 1977, pp. 46–57.

[37] Edmund M Clarke and E Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Workshop on Logic of Programs*. Springer. 1981, pp. 52–71.

[38] J Richard Büchi. "On a decision method in restricted second order arithmetic". In: *The collected works of J. Richard Büchi*. Springer, 1990, pp. 425–435.

[39] David E Muller. "Infinite sequences and finite machines". In: *Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design (swct 1963)*. IEEE. 1963, pp. 3–16.

[40] Michael O Rabin. "Decidability of second-order theories and automata on infinite trees". In: *Transactions of the american Mathematical Society* 141 (1969), pp. 1–35.

[41] Randal E Bryant. "Graph-based algorithms for boolean function manipulation". In: *Computers, IEEE Transactions on* 100.8 (1986), pp. 677–691.

[42] Armin Biere et al. "Symbolic model checking using SAT procedures instead of BDDs". In: *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*. IEEE. 1999, pp. 317–320.

[43] G Cabodi et al. "Secure embedded architectures: Taint properties verification". In: *2016 International Conference on Development and Application Systems (DAS)*. IEEE. 2016, pp. 150–157.

[44] Srivaths Ravi et al. "Security in embedded systems: Design challenges". In: *ACM Transactions on Embedded Computing Systems (TECS)* 3.3 (2004), pp. 461–491.

[45] Bill Miller and Dale C Rowe. "A survey SCADA of and critical infrastructure incidents." In: *RIIT* 12 (2012), pp. 51–56.

[46] Xu Li et al. "Securing smart grid: cyber attacks, countermeasures, and challenges". In: *IEEE Communications Magazine* 50.8 (2012), pp. 38–45.

[47] Andrei Costin et al. "A large-scale analysis of the security of embedded firmwares". In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 95–110.

[48] Fiona Higgins, Allan Tomlinson, and Keith M Martin. "Survey on security challenges for swarm robotics". In: *2009 Fifth International Conference on Autonomic and Autonomous Systems*. IEEE. 2009, pp. 307–312.

[49] G Cabodi et al. "Formal verification of embedded systems for remote attestation". In: *WSEAS Transactions on Computers* 14 (2015), pp. 760–769.

[50] Pramod Subramanyan and Divya Arora. "Formal verification of taint-propagation security properties in a commercial SoC design". In: *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association. 2014, p. 313.

[51] Karim Eldefrawy et al. "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust." In: *NDSS*. Vol. 12. 2012, pp. 1–15.

[52] Job Noorman et al. “Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base”. In: *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 2013, pp. 479–498.

[53] Siani Pearson and Boris Balacheff. *Trusted computing platforms: TCPA technology in context*. Prentice Hall Professional, 2003.

[54] Aurelien Francillon et al. “Systematic Treatment of Remote Attestation.” In: *IACR Cryptology ePrint Archive* 2012 (2012), p. 713.

[55] ISO. *Information technology — Trusted platform module library — Part 1: Architecture*. ISO 11889-1:2015. Geneva, Switzerland: International Organization for Standardization, 2015.

[56] Arvind Seshadri et al. “SCUBA: Secure code update by attestation in sensor networks”. In: *Proceedings of the 5th ACM workshop on Wireless security*. ACM. 2006, pp. 85–94.

[57] Steffen Schulz et al. “Boot attestation: Secure remote reporting with off-the-shelf iot sensors”. In: *European Symposium on Research in Computer Security*. Springer. 2017, pp. 437–455.

[58] Daniele Perito and Gene Tsudik. “Secure code update for embedded devices via proofs of secure erasure”. In: *European Symposium on Research in Computer Security*. Springer. 2010, pp. 643–662.

[59] Liqun Chen et al. “A protocol for property-based attestation”. In: *Proceedings of the first ACM workshop on Scalable trusted computing*. ACM. 2006, pp. 7–16.

[60] Ahmad-Reza Sadeghi and Christian Stüble. “Property-based attestation for computing platforms: caring about properties, not mechanisms”. In: *Proceedings of the 2004 workshop on New security paradigms*. ACM. 2004, pp. 67–77.

[61] David Grawrock. *The Intel safer computing initiative: building blocks for trusted computing*. Vol. 976483262. Intel Press Hillsboro, 2006.

[62] Marcus Peinado et al. “NGSCB: A trusted open system”. In: *Australasian Conference on Information Security and Privacy*. Springer. 2004, pp. 86–97.

[63] Nick L Petroni Jr et al. “Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor.” In: *USENIX Security Symposium*. San Diego, USA. 2004, pp. 179–194.

[64] George Coker et al. “Principles of remote attestation”. In: *International Journal of Information Security* 10.2 (2011), pp. 63–81.

[65] Ivan De Oliveira Nunes et al. “{VRASED}: A verified hardware/software co-design for remote attestation”. In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 2019, pp. 1429–1446.

[66] Wayne H Wolf. “Hardware-software co-design of embedded systems”. In: *Proceedings of the IEEE* 82.7 (1994), pp. 967–989.

[67] Aurélien Francillon et al. “A minimalist approach to remote attestation”. In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–6.

[68] Y-W Hsieh and Steven P Levitan. “Model abstraction for formal verification”. In: *Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society. 1998, pp. 140–147.

[69] E Allen Emerson and Edmund M Clarke. “Using branching time temporal logic to synthesize synchronization skeletons”. In: *Science of Computer programming* 2.3 (1982), pp. 241–266.

[70] Thilo Hafer and Wolfgang Thomas. “Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1987, pp. 269–279.

[71] Robert K Brayton et al. “VIS: A system for verification and synthesis”. In: *International conference on computer aided verification*. Springer. 1996, pp. 428–432.

[72] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. “Benchmarking a model checker for algorithmic improvements and tuning for performance”. In: *Formal Methods in System Design* 39.2 (2011), pp. 205–227.

[73] Gianpiero Cabodi, Paolo Camurati, and Marco Murciano. "Automated abstraction by incremental refinement in interpolant-based model checking". In: *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press. 2008, pp. 129–136.

[74] Gianpiero Cabodi et al. "Boosting interpolation with dynamic localized abstraction and redundancy removal". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 13.1 (2008), p. 3.

[75] Gianpiero Cabodi et al. "Trading-off SAT search and variable quantifications for effective unbounded model checking". In: *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. IEEE Press. 2008, p. 26.

[76] Gianpiero Cabodi et al. "Partitioning interpolant-based verification for effective unbounded model checking". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29.3 (2010), pp. 382–395.

[77] Gianpiero Cabodi, Carmelo Loiacono, and Danilo Vendraminetto. "Optimization techniques for craig interpolant compaction in unbounded model checking". In: *Formal Methods in System Design* 46.2 (2015), pp. 135–162.

[78] Gianpiero Cabodi, Marco Palena, and Paolo Pasini. "Interpolation with guided refinement: Revisiting incrementality in sat-based unbounded model checking". In: *2014 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. 2014, pp. 43–50.

[79] Niklas Een, Alan Mishchenko, and Robert Brayton. "Efficient implementation of property directed reachability". In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc. 2011, pp. 125–134.

[80] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. "Symbolic forward/backward traversals of large finite state machines". In: *Journal of systems architecture* 46.12 (2000), pp. 1137–1158.

[81] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. "Mixing forward and backward traversals in guided-prioritized bdd-based verification". In: *International Conference on Computer Aided Verification*. Springer. 2002, pp. 471–484.

[82] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. "Improving the efficiency of BDD-based operators by means of partitioning". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18.5 (1999), pp. 545–556.

[83] Gianpiero Cabodi. "Meta-bdds: A decomposed representation for layered symbolic manipulation of boolean functions". In: *International Conference on Computer Aided Verification*. Springer. 2001, pp. 118–130.

[84] G. Cabodi et al. "Secure Path Verification". In: *1st IEEE International Verification and Security Workshop (IVSW)*. 2016, pp. 1–6. DOI: https://doi.org/10.1109/IVSW.2016.7566608.

[85] Thomas Ristenpart et al. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds". In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 199–212.

[86] Karl Koscher et al. "Experimental security analysis of a modern automobile". In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 447–462.

[87] Patrick McDaniel and Stephen McLaughlin. "Security and privacy challenges in the smart grid". In: *IEEE Security & Privacy* 7.3 (2009), pp. 75–77.

[88] Andrei Sabelfeld and Andrew C Myers. "Language-based information-flow security". In: *IEEE Journal on selected areas in communications* 21.1 (2003), pp. 5–19.

[89] Geoffrey Smith. "Principles of secure information flow analysis". In: *Malware Detection*. Springer, 2007, pp. 291–307.

[90] Nevin Heintze and Jon G Riecke. "The SLam calculus: programming with secrecy and integrity". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1998, pp. 365–377.

[91] Geoffrey Smith and Dennis Volpano. "Secure information flow in a multi-threaded imperative language". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1998, pp. 355–364.

[92] Dennis Volpano and Geoffrey Smith. "Probabilistic noninterference in a concurrent language 1". In: *Journal of Computer Security* 7.2-3 (1999), pp. 231–253.

[93] François Pottier and Vincent Simonet. "Information flow inference for ML". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25.1 (2003), pp. 117–158.

[94] Andrei Sabelfeld and David Sands. "A per model of secure information flow in sequential programs". In: *Higher-order and symbolic computation* 14.1 (2001), pp. 59–91.

[95] Sofia Bekrar et al. "A taint based approach for smart fuzzing". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE. 2012, pp. 818–825.

[96] Ella Bounimova, Patrice Godefroid, and David Molnar. "Billions and billions of constraints: Whitebox fuzz testing in production". In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 122–131.

[97] Sang Kil Cha et al. "Unleashing mayhem on binary code". In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394.

[98] Istvan Haller et al. "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations". In: *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 2013, pp. 49–64.

[99] Alexandre Rebert et al. "Optimizing seed selection for fuzzing". In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 861–875.

[100] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)". In: *2010 IEEE symposium on Security and privacy*. IEEE. 2010, pp. 317–331.

[101] Yan Shoshitaishvili et al. "Sok:(state of) the art of war: Offensive techniques in binary analysis". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 138–157.

[102] Thanassis Avgerinos et al. "Enhancing symbolic execution with veritesting". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 1083–1094.

[103] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A platform for in-vivo multi-path analysis of software systems". In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 1. ACM. 2011, pp. 265–278.

[104] Jeffrey Wilhelm and Tzi-cker Chiueh. "A forced sampled execution approach to kernel rootkit identification". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2007, pp. 219–235.

[105] Madjid Maidi. "The common fragment of CTL and LTL". In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE. 2000, pp. 643–652.

[106] Ziyad Hanna. "Jasper case study on formally verifying secure on-chip datapaths". In: (2013).

[107] Gianpiero Cabodi et al. "Embedded systems secure path verification at the hardware/software interface". In: *IEEE Design & Test* 34.5 (2017), pp. 38–46.

[108] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. "FANCI: identification of stealthy malicious logic using boolean functional analysis". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 697–708.

[109] Yier Jin and Yiorgos Makris. "Hardware Trojan detection using path delay fingerprint". In: *2008 IEEE International workshop on hardware-oriented security and trust*. IEEE. 2008, pp. 51–57.

[110] Yier Jin. "Design-for-security vs. design-for-testability: A case study on dft chain in cryptographic circuits". In: *2014 IEEE Computer Society Annual Symposium on VLSI*. IEEE. 2014, pp. 19–24.

[111] Jeyavijayan Rajendran, Vivekananda Vedula, and Ramesh Karri. "Detecting malicious modifications of data in third-party intellectual property cores". In: *Proceedings of the 52nd Annual Design Automation Conference*. ACM. 2015, p. 112.

[112] Mainak Banga and Michael S Hsiao. "Trusted RTL: Trojan detection methodology in pre-silicon designs". In: *2010 IEEE international symposium on hardware-oriented security and trust (HOST)*. IEEE. 2010, pp. 56–59.

[113] Flavio M De Paula et al. "Backspace: Formal analysis for post-silicon debug". In: *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. IEEE Press. 2008, p. 5.

[114] Xuehui Zhang and Mohammad Tehranipoor. "Case study: Detecting hardware Trojans in third-party digital IP cores". In: *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE. 2011, pp. 67–70.

[115] Xiaolong Guo et al. "Pre-silicon security verification and validation: A formal perspective". In: *Proceedings of the 52nd Annual Design Automation Conference*. ACM. 2015, p. 145.

[116] Yier Jin, Bo Yang, and Yiorgos Makris. "Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing". In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE. 2013, pp. 99–106.

[117] Eric Love, Yier Jin, and Yiorgos Makris. "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition". In: *IEEE Transactions on Information Forensics and Security* 7.1 (2011), pp. 25–40.

[118] Stephanie Drzevitzky. "Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration". In: *2010 International Conference on Field Programmable Logic and Applications*. IEEE. 2010, pp. 255–258.

[119] Ranjit Jhala and Rupak Majumdar. "Software model checking". In: *ACM Computing Surveys (CSUR)* 41.4 (2009), pp. 1–54.

[120] Bingchang Liu et al. "Software vulnerability discovery techniques: A survey". In: *2012 fourth international conference on multimedia information networking and security*. IEEE. 2012, pp. 152–156.

[121] Stephen Chong et al. "Report on the NSF workshop on formal methods for security". In: *arXiv preprint arXiv:1608.00678* (2016).

[122] Geoffrey Smith et al. "A New Type System for Secure Information Flow." In: *CSFW*. Vol. 1. Citeseer. 2001, p. 4.

[123] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. "Tracking information flow in dynamic tree structures". In: *European Symposium on Research in Computer Security*. Springer. 2009, pp. 86–103.

[124] Konstantin Knorr. "Dynamic access control through Petri net workflows". In: *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. IEEE. 2000, pp. 159–167.

[125] Konstantin Knorr. "Multilevel security and information flow in Petri net workflows". In: *Conference on Telecommunication Systems*. Citeseer. 2001, pp. 613–615.

[126] Paul Watson. "A multi-level security model for partitioning workflows over federated clouds". In: *Journal of Cloud Computing: Advances, Systems and Applications* 1.1 (2012), p. 15.

[127] Wen Zeng, Maciej Koutny, and Paul Watson. "Verifying secure information flow in federated clouds". In: *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. IEEE. 2014, pp. 78–85.

[128] Wen Zeng et al. "Formal verification of secure information flow in cloud computing". In: *Journal of Information Security and Applications* 27 (2016), pp. 103–116.

[129] Mohit Tiwari et al. "Complete information flow tracking from the gates up". In: *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 2009, pp. 109–120.

[130] G Edward Suh et al. "Secure program execution via dynamic information flow tracking". In: *ACM Sigplan Notices*. Vol. 39. 11. ACM. 2004, pp. 85–96.

[131] Steven Arzt et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps". In: *Acm Sigplan Notices* 49.6 (2014), pp. 259–269.

[132] William Enck et al. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones". In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), pp. 1–29.

[133] Feng Qin et al. "Lift: A low-overhead practical information flow tracking system for detecting security attacks". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE. 2006, pp. 135–148.

[134] Thoshitha T Gamage, Bruce M McMillin, and Thomas P Roth. "Enforcing information flow security properties in cyber-physical systems: A generalized framework based on compensation". In: *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*. IEEE. 2010, pp. 158–163.

[135] David W Palmer and Parbati Kumar Manna. "An efficient algorithm for identifying security relevant logic and vulnerabilities in RTL designs". In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE. 2013, pp. 61–66.

[136] Pramod Subramanyan et al. "Verifying information flow properties of firmware using symbolic execution". In: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2016, pp. 337–342.

[137] Frederick K Frantz. "A taxonomy of model abstraction techniques". In: *Proceedings of the 27th conference on Winter simulation*. 1995, pp. 1413–1420.

[138] David E Long et al. *Model checking, abstraction, and compositional verification*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1993.

[139] Tachio Terauchi and Alex Aiken. "Secure information flow as a safety problem". In: *International Static Analysis Symposium*. Springer. 2005, pp. 352–367.

[140] Gianpiero Cabodi et al. “Model Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification”. In: *International Conference on Codes, Cryptology, and Information Security*. Springer. 2019, pp. 462–479.

[141] Gianpiero Cabodi et al. “Model Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification”. In: *Electronics* 8.9 (2019), p. 1057.

[142] Yuval Yarom and Katrina Falkner. “FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack”. In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 719–732.

[143] Bo Yang, Kaijie Wu, and Ramesh Karri. “Scan based side channel attack on dedicated hardware implementations of data encryption standard”. In: *2004 International Conferce on Test*. IEEE. 2004, pp. 339–344.

[144] Lang Lin et al. “Trojan side-channels: lightweight hardware trojans through side-channel engineering”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2009, pp. 382–395.

[145] Mohammad Tehranipoor and Cliff Wang. *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[146] Jason Lowe-Power et al. “A case for exposing extra-architectural state in the ISA: position paper”. In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM. 2018, p. 8.

[147] G Joy Persial, M Prabhu, and R Shanmugalakshmi. “Side channel attack-survey”. In: *International Journal of Advanced Scientific Research and Review* 1.4 (2011), pp. 54–57.

[148] Junfeng Fan et al. “State-of-the-art of secure ECC implementations: a survey on known side-channel attacks and countermeasures”. In: *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 76–87.

[149] YongBin Zhou and DengGuo Feng. "Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing." In: *IACR Cryptology ePrint Archive* 2005 (2005), p. 388.

[150] Rich Bennett et al. "How to live in a post-meltdown and-spectre world". In: *Communications of the ACM* 61.12 (2018), pp. 40–44.

[151] Mohd Fadzil Abdul Kadir et al. "Retpoline Technique for Mitigating Spectre Attack". In: *2019 6th International Conference on Electrical and Electronics Engineering (ICEEE)*. IEEE. 2019, pp. 96–101.

[152] Daniel Gruss et al. "Kaslr is dead: long live kaslr". In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2017, pp. 161–176.

[153] Andrew Prout et al. "Measuring the Impact of Spectre and Meltdown". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE. 2018, pp. 1–5.

[154] David A Patterson, John L Hennessy, and David Goldberg. *Computer architecture: a quantitative approach*. Vol. 2. Morgan Kaufmann San Mateo, CA, 1990.

[155] Robert M Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units". In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[156] Butler W Lampson. "A note on the confinement problem". In: *Communications of the ACM* 16.10 (1973), pp. 613–615.

[157] Paul C Kocher. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". In: *Annual International Cryptology Conference*. Springer. 1996, pp. 104–113.

[158] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA key extraction via low-bandwidth acoustic cryptanalysis". In: *Annual Cryptology Conference*. Springer. 2014, pp. 444–461.

[159] Mordechai Guri et al. "PowerHammer: Exfiltrating data from air-gapped computers through power lines". In: *IEEE Transactions on Information Forensics and Security* (2019).

[160] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential power analysis". In: *Annual International Cryptology Conference*. Springer. 1999, pp. 388–397.

[161] Dakshi Agrawal et al. "The EM side—channel (s)". In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2002, pp. 29–45.

[162] Elad Carmon, Jean-Pierre Seifert, and Avishai Wool. "Photonic side channel attacks against RSA". In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2017, pp. 74–78.

[163] Raphael Spreitzer. "Pin skimming: Exploiting the ambient-light sensor in mobile devices". In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. 2014, pp. 51–62.

[164] Julie Ferrigno and M Hlaváč. "When AES blinks: introducing optical side channel". In: *IET Information Security* 2.3 (2008), pp. 94–98.

[165] J Alex Halderman et al. "Lest we remember: cold-boot attacks on encryption keys". In: *Communications of the ACM* 52.5 (2009), pp. 91–98.

[166] Peter Pessl et al. "{DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks". In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 565–581.

[167] Daniel J Bernstein. "Cache-timing attacks on AES". In: (2005).

[168] Shaizeen Aga and Satish Narayanasamy. "Invisimem: Smart memory defenses for memory bus side channel". In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 94–106.

[169] Monodeep Kar et al. "Exploiting fully integrated inductive voltage regulators to improve side channel resistance of encryption engines". In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. 2016, pp. 130–135.

[170] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "S $ A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 591–604.

[171] Qian Ge et al. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *Journal of Cryptographic Engineering* 8.1 (2018), pp. 1–27.

[172] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache attacks and countermeasures: the case of AES”. In: *Cryptographers’ track at the RSA conference*. Springer. 2006, pp. 1–20.

[173] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache template attacks: Automating attacks on inclusive last-level caches”. In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 897–912.

[174] Ofir Weisse et al. “Nda: Preventing speculative execution attacks at their source”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 572–586.

[175] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), pp. 244–263.

[176] Werner Damm and Amir Pnueli. “Verifying out-of-order executions”. In: *Advances in Hardware Design and Verification*. Springer, 1997, pp. 23–47.

[177] Jun Sawada and Warren A Hunt. “Processor verification with precise exceptions and speculative execution”. In: *International Conference on Computer Aided Verification*. Springer. 1998, pp. 135–146.

[178] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. “Decomposing the proof of correctness of pipelined microprocessors”. In: *International Conference on Computer Aided Verification*. Springer. 1998, pp. 122–134.

[179] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. “Proof of correctness of a processor with reorder buffer using the completion functions approach”. In: *International Conference on Computer Aided Verification*. Springer. 1999, pp. 47–59.

[180] Jerry R Burch and David L Dill. "Automatic verification of pipelined microprocessor control". In: *International Conference on Computer Aided Verification*. Springer. 1994, pp. 68–80.

[181] Jens U Skakkebæk, Robert B Jones, and David L Dill. "Formal verification of out-of-order execution using incremental flushing". In: *International Conference on Computer Aided Verification*. Springer. 1998, pp. 98–109.

[182] Kenneth L McMillan. "Verification of an implementation of Tomasulo's algorithm by compositional model checking". In: *International Conference on Computer Aided Verification*. Springer. 1998, pp. 110–121.

[183] Anuj Goel et al. "BDD based procedures for a theory of equality with uninterpreted functions". In: *International Conference on Computer Aided Verification*. Springer. 1998, pp. 244–255.

[184] Sergey Berezin et al. "Verification of out-of-order processor designs using model checking and a light-weight completion function". In: *Formal Methods in System Design* 20.2 (2002), pp. 159–186.

[185] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[186] Patrice Godefroid, Michael Y Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing". In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[187] Armaiti Ardeshiricham et al. "Register transfer level information flow tracking for provably secure hardware design". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 1691–1696.

[188] David Zhu et al. "TaintEraser: Protecting sensitive data leaks using application-level taint tracking". In: *ACM SIGOPS Operating Systems Review* 45.1 (2011), pp. 142–154.

[189] Gilles Barthe, David Naumann, and Tamara Rezk. "Deriving an information flow checker and certifying compiler for java". In: *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE. 2006, 13–pp.

[190] Ranjit Jhala and Kenneth L McMillan. "Microarchitecture verification by compositional model checking". In: *International Conference on Computer Aided Verification*. Springer. 2001, pp. 396–410.

[191] Panagiotis Manolios and Sudarshan K Srinivasan. "A complete compositional reasoning framework for the efficient verification of pipelined machines". In: *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*. IEEE. 2005, pp. 863–870.

[192] Armin Biere, Keijo Heljanko, and Siert Wieringa. "AIGER 1.9 and beyond". In: *Available at fmv. jku. at/hwmcc11/beyond1. pdf* (2011). (Accessed on 18/09/2019).

[193] Gianpiero Cabodi et al. "Speeding up model checking by exploiting explicit and hidden verification constraints". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association. 2009, pp. 1686–1691.

[194] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. "Strengthening model checking techniques with inductive invariants". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.1 (2008), pp. 154–158.

[195] Gianpiero Cabodi and Sergio Nocco. "Optimized model checking of multiple properties". In: *2011 Design, Automation & Test in Europe*. IEEE. 2011, pp. 1–4.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.