UNIVERSITÀ DI PISA
FACOLTÀ DI INGEGNERIA

Corso di Laurea Specialistica in Ingegneria Elettronica
Nuovo Ordinamento
Anno Accademico 2010–2011

Elaborato finale

STATIC ANALYSIS OF CIRCUITS FOR SECURITY

Candidato: Riccardo Bresciani

Relatore: Prof. Luca Fanucci

A tutti quelli che mi hanno dato un motivo per arrivarci in fondo.

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots.
So far, the Universe is winning.

Rick Cook

Abstract The purpose of the present work is to define a methodology to analyze a system description given in VHDL code and test its security properties. In particular the analysis is aimed at ensuring that a malicious user cannot make a circuit output the secret data it contains. ❖

❖ This master thesis is based on work done during my stage at LSV, ENS Cachan in spring/summer 2008, under the supervision of Jean Goubalt-Larrecq (LSV, ENS Cachan & CNRS & INRIA, France — goubalt@lsv.ens-cachan.fr), in collaboration with David Lubicz (IRMAR, Université de Rennes 1 & DGA, France — david.lubicz@univ-rennes1.fr) and Nicolas Guillermin (DGA, France — nicolas.guillermin@dga.defense.gouv.fr), with the financial support of ENS Cachan and Scuola Superiore Sant'Anna, Pisa.

Contents

Chapter 1

Introduction

Timeo Danaos et dona ferentes.

Publius Vergilius Maro

Nowadays a lot of tasks requiring high-speed processing capabilities are not performed via software anymore, but through hardware: an *ad hoc* component can be highly optimized for a specific task, possibly with a high degree of parallelism. For this reason *ASICs*[1] and *FPGAs*[2] represent a good solution to address this need.
Speed is not the only reason to perform some operation via hardware, for example cost savings may be achieved by producing a component with specific (but limited) capabilities: such a component, expressely tailored on the task, can be much cheaper than a general purpose processor.
Last but not least, sometimes the need of using hardware derives from security issues as a hardware component is harder to crack or to clone, as doing so requires a sophisticated equipment; it is also an excellent way to store cryptographic keys and provide personal identification.

Hardware programming is largely made through hardware description languages such as VHDL or Verilog: these two languages are equivalent in terms of expressiveness, therefore they share the same verification needs.

The particular need addressed in the present work is the one of ensuring secrecy of confidential data, so that we can be sure that data stored in a circuit will not flow outside: the challenge is to verify these secrecy properties not on the circuit but directly on VHDL code, formally.
We are therefore applying formal methods at the code level, as we are aiming at verifying what is actually going to be turned into hardware.
We propose two different methods: one is based on the black-box verification principles, the other one represents the white-box alternative — whenever applicable, as in some cases we may not have the inner details of how a component is made, but only its specifications.

The remainder of this section is dedicated to outlining the structure of this thesis.

1. **Introduction** It is not possible to understand the verification approach to be performed on hardware without first addressing both the concept of modelling digital circuits and the topic of formal methods from a broader perspective.

[1] ASIC stands for *Application-Specific Circuit*.
[2] FPGA stands for *Field-Programmable Gate-Array*.

For this reason the present chapter is divided into three sections: the first one discusses the general ideas of modelling digital circuits, the second one aims at giving a reasonably complete overview of the state of the art in formal methods, with special care towards what is relevant to verification of hardware and software, and finally the third one outlines the approach that has been followed to apply this wealth of techniques to VHDL.

2. **Circuit Synthesis through HDL** In this chapter we aim at giving a general overview of VHDL and its features, in particular those which are relevant to this work. This will lead to the definition of a subset of the language we will be referring to.

3. **Security and Protocol Verification** Security is an area which comprises many different topics: in chapter 3 we will try to go quickly through the most important aspects of security, and focus our attention on areas that are most relevant to the scope of this thesis, in particular what concerns protocol verification.

   The model, which underlies the analysis techniques we propose, is in fact the one of protocol verification, as an electronic device may be seen as the implementation of a communication protocol, where the agents can send and receive "messages" by means of the accessible terminals; as we are interested in security properties, protocol verification is a mature area from which we can draw ideas and tools. Protocol verifiers will also be presented, with special care to ProVerif.

4. **Process Calculi** In order to use protocol verification techniques, it is useful to give an overview of process calculi, as protocols are usually modelled as communicating processes. Likewise, we will be modelling electronic components as processes.

5. **Logic** An understanding of logic is also needed (in particular for what concerns *Horn clauses*), as we will verify some properties through some tools that process logic clauses as input, such as theorem provers and model finders.

6. **Black-box Verification Model** As we have already hinted, we will be proposing two different verification methods. The first one sees some subcomponents as black boxes, assuming they are working according to their specifications, and aims at verifying that there is no weakness introduced by the way we have interconnected these subcomponents.

7. **Looking into the Black Box** The second verification method is applicable if the description of a subcomponent is available: in addition to what we were able to prove with the black-box verification model, we can prove whether this component implements its specification.

8. **From VHDL to π-calculus** To put all of these ideas into practice we need to translate the VHDL code into a format, which is understandable by the tools we are using. We will be using a protocol verifier — ProVerif — for the black-box verification model, therefore we have to provide it with a π-calculus model of the hardware we aim at analysing. This analysis tells us whether the model we have provided leaks sensitive data.

9. **From VHDL to MACE4** For the white-box verification model, we will be using a model finder — MACE4 — that inputs logic clauses. The result of this analysis shows which terminals or wires must not be accessible to a malicious user: accessing these terminals or wires would result in compromising the confidentiality of the data we want to keep secret.

10. **Results and Conclusion** Finally we conclude by summarizing briefly the work done and the results obtained.

1.1 Modelling Digital Systems

The complexity of digital system has been constantly increasing in the last decades and nowadays it is not viable to understand (or project) such systems without a fair level of abstraction.
This has changed deeply the way electronic systems are designed, as a *bottom-up* methodology is not feasible, and this older paradigm has been replaced by a *top-down* methodology: if we start with a requirement document for the system, we can then design a high-level abstraction of the system that complies with the requirements; this abstract structure can then be decomposed into interacting subcomponents and this can be repeated iteratively, until we eventually get to the very bottom-level of transistor design — although normally the process descends down to the level of ready-made primitive components, which are made available to designers through specific component libraries.
Each subsystem can be designed independently of others, and this gives benefits in terms of workload sharing within a team and allows a designer to focus on one part of the system at a time.

The designer can think in terms of a model, abstracting away from tedious and irrelevant details: this implies that a system can be modelled in different ways, depending on what the details are relevant in a given context.
There are a number of good reasons for having models of a system [Ash08]:

- the use of a formal model to *communicate requirements* makes sure that *all* requirements are precisely identifiable, in a way that does not leave room for disagreement between customers and developers. Besides, a model clearly states what is required and what can be developed in different ways — what is not in the model is left to the designer's freedom of choice;
- a formal model also helps a user to understand the *functions* of a system: as a designer cannot foresee all of the possible ways a system can be used, providing a model of the system itself enables a user to check if it is going to work in the context of interest;
- a model enables developers to test the system they are building through *simulation*. This helps making sure that the high-level abstraction of the system behaves in the same way as we descend in the design hierarchy, one level of abstraction after the other. These same tests may be used later on the physical circuit, to check that the results are comparable to the simulation;
- having a model of the system is what underlies *formal verification* of the correctness of a design. We will not get into too much detail here, as we will tbe talking extensively of formal verification techniques in the following section 1.2;
- there are tools that can sythesize circuits automatically starting from a model of the system to be developed: an engineer can focus on developing a good model, the tools will take care of turning it into a manufacturable layout.

System models can be categorized into three domains: the *functional domain* is concerned with the operations a system can perform and can be seen as the most abstract one, as it does not give any indication on how a function is implemented; the *structural domain* is concerned with the actual composition of the system, in terms of interconnected components; finally the *geometric domain* deals with the physical system layout.
Each of these domains is divided into different levels of abstractions — see figure 1.1.

We will be talking more precisely about these topics in chapter 2, with the perspective of hardware description languages in mind.

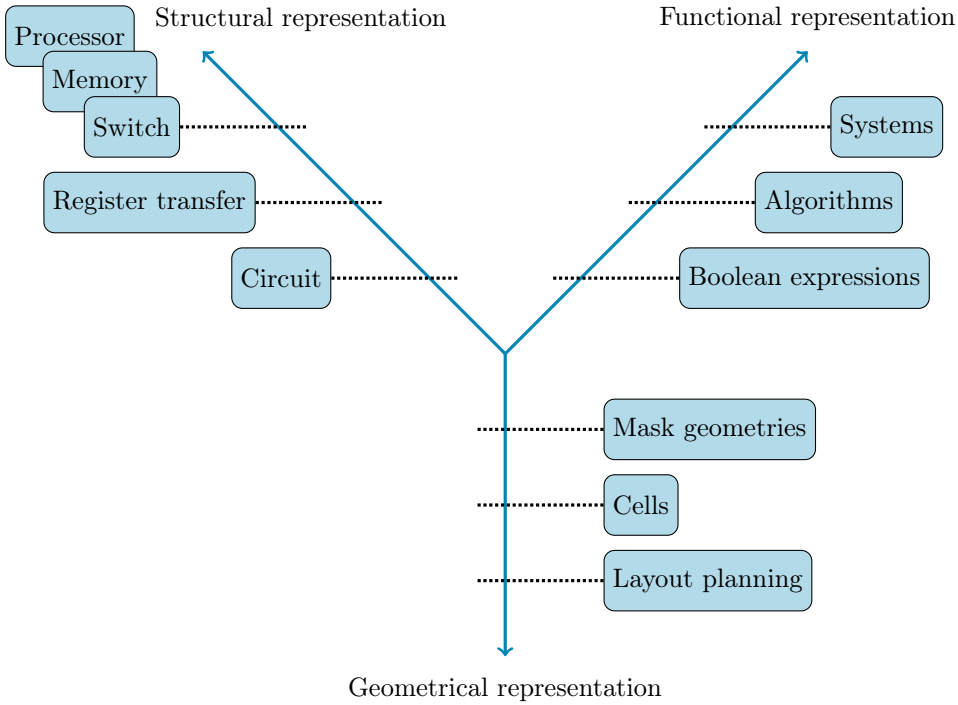


Figure 1.1: Design levels. [GK83]

1.2 Formal Methods

As computers are becoming essential to everyday's life, and also embedded systems are a constant presence, nowadays we rely heavily on the services provided by electronic devices. This has led to a strong increment in the importance of applying formal methods: the goal is to deliver dependable systems in an effective manner.
From a more general perspective we can see that there are a lot of complex application where some properties represent critical features, and therefore they have to be enforced by any means.

The very founders of computer science, people like Turing and von Neumann, had also some idea of software verification in mind, and this concept has been cropping up more and more often, until technical advances made it feasible: the computational power that has been gained over the years has made formal techniques available and usable for actual applications, overcoming all of those limitations that used to keep them relegated in a purely theoretical world, preventing them from being effectively used outside of it.
This breathed new life into the development of formal methods and gave an impulse that has brought to a whole new set of theories in the field. And all of this is still *in fieri*.

People have different misconceptions about formal methods. In the nineties Hall [Hal90] gave a list of 7 myths about formal methods, followed a few years later by another 7-myth list by Bowen and Hinchey [BH95a]: nowadays the situation is somehow better from this point of view, although some myths seem to last. [BH06]
For this reason we provide a visual synthesis of the aforementioned articles in figure 1.2.

When verifying a system, the properties that are to be sought vary depending on the purpose of the system. What we usually deal with is:

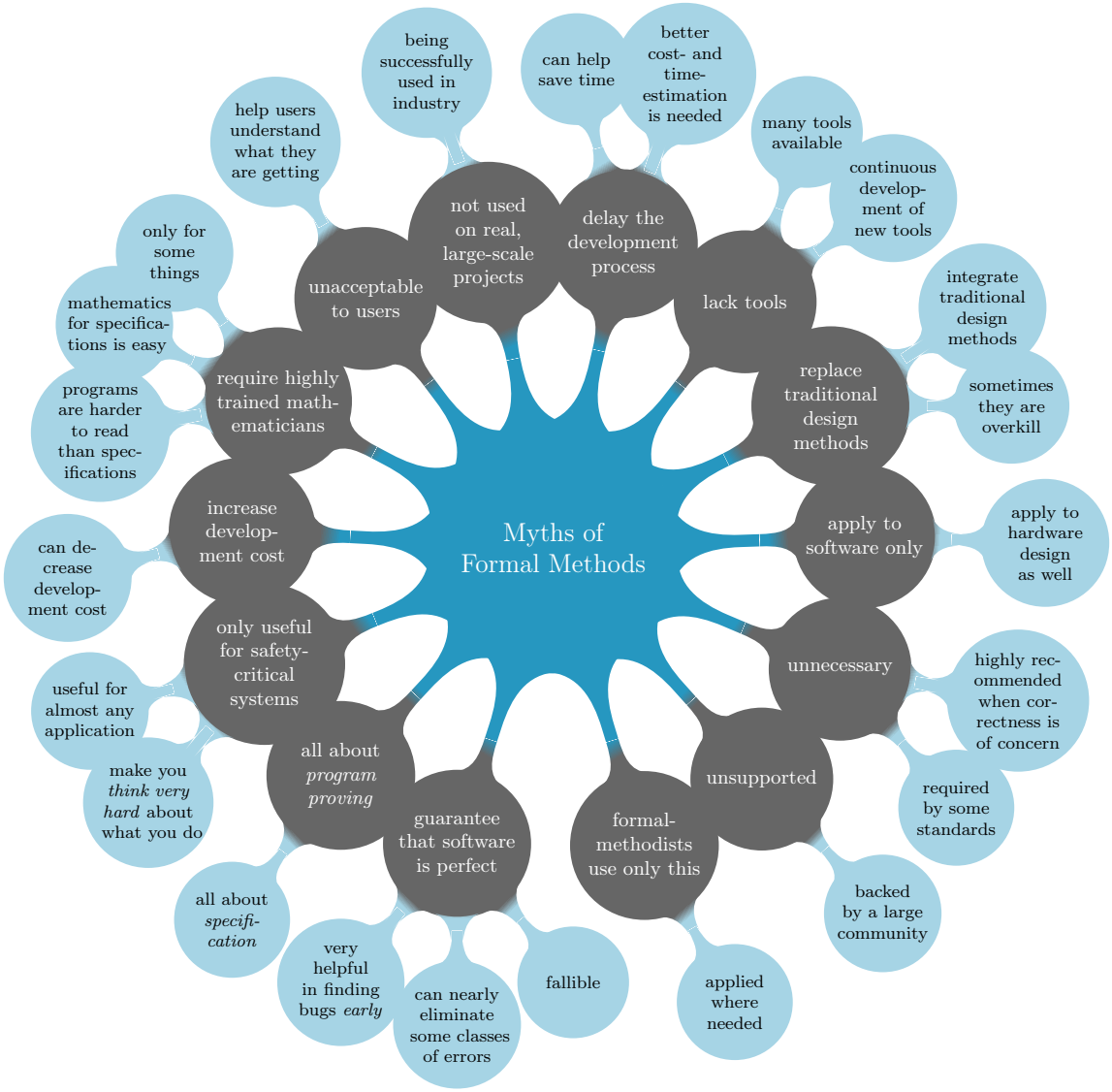


Figure 1.2: 14 myths of formal methods (on dark background) and corresponding facts (on light background). [Hal90; BH95a]

- *availability* — services provided by a system are accessible in time;
- *reliability* — a system is not subject to failures;
- *safety* — a system is not source of hazards, risks or damage to people, things or environment;
- *security* — a system is not vulnerable to intrusion or leaking sensitive information.

Failures in different kinds of systems lead to different consequences: for example *safety-critical systems* (or *life-critical systems*), such as flight control systems, must have no flaws, as this may lead to a plane crash.
A failure in a *mission-critical system*, such as a navigation control system or a flash memory in a core position, is less serious in the sense that it costs no lives, but it may cause a mission to abort.
Sometimes people talk about *business-critical systems* (a bank accounting platform is such as system): a failure in this case may lead to a considerable loss of money.

These are all settings where application of formal techniques is not only one desirable optional step in the development process, but a required guarantee that allows users to trust a system.

In the past we can find some bugs which led to epic failures, that would have been avoided if appropriate formal verification had been carried out on those projects.
Two really expensive bugs are worth mentioning:

- the Ariane 5 rocket exploded on its maiden launch after only 37 seconds: the reason was a 64-bit floating-point value being converted to a 16-bit signed integer. The estimated loss is around half a billion dollars [Nus97];
- the Pentium processor, released in 1994, had a bug in the floating-point unit and was returning wrong results for some computations. It turned out that five entries were missing from the look-up table that was needed by the algorithm: Intel was prepared to face a cost of 475 million dollars to recall and substitute the bugged chips [Cip95; Wil97].

Software and hardware development can benefit from extensive use of formal methods, as they improve the quality of the software being developed[3]: what we can obtain is a product which is *correct by design* or at least *proven to be correct*, rather than *tested* or *simulated to be correct*.
The difference is crucial, as testing can never make sure that the result obtained at the end of the development process complies with what was originally asked by the client, but can only show that in some chosen settings the given requirements are satisfied: if a bug is found we can only state that the system has a problem to be fixed, nothing more; in E.W.Dijkstra's words, *"testing shows the presence, not the absence of bugs"*[4]. [BR70; Dij72]
Conversely formal methods can prove correctness by analysing the *specification* of a product, determining which properties it satisfies and what other properties it does not, thus deducing whether in some setting some of the requirements cannot be met: either we find a bug or we state that the system is not affected by any bug *with respect to that specification*.
Sometimes the analysis performed via formal methods is referred to as *static analysis*, as it does not require the execution of a piece of code or producing a working prototype. Conversely testing is sometimes referred to as *dynamic analysis*.
An approach to formal analysis is to build a system, which implements a specification satisfying the requirements we are interested in, and then proving that the resulting *implementation* is a *refinement* of the initial specification. The compliance of an implementation with the corresponding specification is usually shown via a so-called *refinement calculus*. Still we must be aware that a formally verified system is just as good as its specification, so special care is required when defining a specification.
An alternative approach, that can be applied to systems built without using formal methods, is to verify properties on a *model* of the real system, that we will treat as its specification: again, the results are just as good as this specification.

The fact that the results are weighted against the appropriateness of the specification is one of the reason why in any case formal development, though overcoming the limitations of testing, still requires it[5]: errors may derive from a wrong or poor specifications.

[3] As in any other human — and thus error-prone — activity, absolute perfection is not something realistically achievable. Nonetheless formal methods help filling the gap: we could say that *"they work largely by making you think very hard about the system you propose to build"*. [Hal90]

[4] Incidentally, as an indication on the historical interest on formal methods, it is worth noticing that this statement dates back to 1969 and was said during a conference on software engineering techniques sponsored by the NATO Science Committee: *software specification* and *correctness* were among the topics it addressed. [BR70]

[5] *"Thou shalt test, test, and test again."* [BH95b; BH06]

Moreover, sometimes a human error can affect the verification procedure itself, possibly producing wrong results: again, formal methods represent a more rigorous technique than the traditional programming paradigms — therefore they can lead to increasing confidence in system integrity and performance *when applied correctly*, as their rigour helps reducing (though not eliminating) human error[6] — and are particularly suitable for *some*[7] applications, but that does not mean that they guarantee the development of bug-free products.

Besides improving overall product quality, the use of formal methods may be more cost-effective than traditional methods. One reason is because it helps spotting bugs in earlier stages of the development process (when they are cheaper to correct): the use of formal methods in industry is entering the development routine, and this allows some flaws to be uncovered well before a product undergoes testing (some examples of application of formal methods in industry may be found in a recent survey by Woodcock et al. [Woo+09]).
Another cost-saving feature that derives from the use of formal methods is that components can be reused: it is really easy to reuse formally developed components as they have a formal specification that can be quickly integrated as a *black box* into a new specification; besides, it is not simply a black box, it is a *verified* black box. A special kind of reuse is when a system is *ported* to a different architecture.
From a maintenance perspective, a formally developed system generally features a cleaner architecture, therefore maintenance can be more efficient.

Formal methods can be used in any and all of the steps of the development process, so they can be effective to help maintaining high-quality standards throughout the evolution of a product.
They also provide guidelines about how things should be done, sometimes drawing a line between what is considered to be good development practice (*e.g.* code that can be verified[8], that makes extensive use of assertions[9,10] and of any other device conceived to enforce correctness, that is available in the programming language being used) and what is to be avoided.

Formal methods is a mature area, but nonetheless it is still very attractive and vital for research, in terms of number of researchers, publications and funding. [Woo+09]

1.2.1 Product Development

The development of a product begins with the specification of the requirements it must comply with: these are properties which abstract from the actual implementation and the product can be engineered as preferred, as long as it meets all of the requirements.
It is crucial though that requirements are expressed in a complete and unambiguous way, so that both the designer and the user have the same understanding of what requirements the product is expected to comply with.
Requirement engineering comprises all of the steps that concur to the definition of the specification. According to Nuseibeh and Easterbrook [NE00] these steps are the following:

[6] Formal methods are no *panacea*, the more realistic goal is expressed in the manifesto of the *Verified Software Initiative*: *"We envision a world in which computer programmers make no more mistakes than other professionals."* [Hoa+09]

[7] Traditional development methods may turn out to be more cost-effective for some non-critical applications, sometimes it is the other way around.

[8] This is actually no big limitation to a programmer's freedom, as programming theory now covers many aspects of modern programming languages.

[9] In Microsoft Office there is an assertion about every tenth line of code (dropping to one every hundredth line in Windows). Nonetheless their existence is mostly due to testing purpose, rather than for proving correctness or the verifying if the program implementation refines its specification — but this programming practice is propedeutical to this goal: assertion may enable a verifying compiler to verify assertion at compile time, rather than at runtime. [Hoa02; BH06]

[10] Assertions have become part of the VHDL language after the 2008 standard has been approved. [Ash08; Iee]

- eliciting requirements;
- modelling and analysing requirements;
- communicating requirements;
- agreeing requirements;
- evolving requirements.

Having a high-level specification to comply with simplifies the task of organising the architectural structure of their components.
The next step is the implementation level and we can use formal methods to verify the correctness of the code that has been written: formal methods aim at proving that, withstanding certain conditions, a verified implementation satisfies the requirements outlined in the specification.
Developers can also avail of code generators that provide verified code, which is directly derived from formal models.
Once we have a verified product, we can use formal methods for its maintenance and evolution to keep everything working at the verified level.

1.2.2 Tool Support

Tool support is fundamental for the successful application of formal method techniques.

Some 20 years ago verification was performed through syntax and type-checkers (this is for example the case of the *IBM CICS transaction processing system* [HK91], first major technical achievement in this area), or also done by hand (yet in a successful way, as testified by the *Mondex project*, featuring a 200-page proof [Woo+08]).

Nowadays we have to face larger projects and we need (and we can avail of) more powerful tools, either completely automated or requiring human interaction.
Thanks to advances in the theory, we can count on *automated theorem provers* (in particular for first-order logic), *proof assistants*, *SAT*[11] and *SMT*[12] *solvers*, *model finders*, *model checkers*, *protocol verifiers* and so on, each tool with its own advantages and limitations.
The research community keeps on developing new tools — as well as maintainig and improving the existent ones — and some commercial tools are also available (a well-known example is *SCADE* by Esterel Technologies [Ber08], used for example by Airbus in the last decade), nevertheless some more work is still required in order to improve usability of these tools and foster industrial application to embrace a larger number of projects.

In the present work only some of the tools mentioned above have been used, and their features will be presented later on.

1.3 Static Analysis of Circuits

The previous section was meant to make the point, that from a general perspective there are several advantages of static analysis compared to dynamic analysis.

[11] In complexity theory the *satisfiability problem* is usually referred to as *SAT*.
[12] *Satisfiability Modulo Theories* — this is an extension of the SAT problem.

Let us now think of what this means in the hardware world: most of the above considerations still apply (with hardware programming the boundary between electronics and computer science is blurred), but there are some concepts which have particularly relevant implications for hardware designers.

If there is a software bug usually a patch is released, and the product is fixed after the patch has been applied (in the best case, *i.e.* unless the product has been critically compromised because of that bug). Hardware components are not always fixable. For example let us imagine we want to implement in hardware a function and we design an *ad hoc* integrated circuit: what is the cost of discovering a bug after all the masks have been produced? Or even worse, what is the price of having to replace a component, after it has already been produced and, possibly, deployed? It is hard to quantify, but a quite accurate answer may be "too high" — let us think again of the infamous Pentium FDIV bug. [Cip95; Wil97]
The advantage of static analysis is that it does not require that a component is actually implemented to analyse it: this is potentially a great cost-saving feature, as bugs can be detected at the very first stages of development, so this is particularly important if the component is hardware.

Though formal methods cannot eliminate hardware failures, they become important in verifying that error-handling procedures are implemented correctly.
Besides this, they can also help preventing these failures: a reasonable specification may require that the normal functioning of a system does not wear out a component, and we can prove whether the way we have implemented that system satisfies this crucial requirement. An example may be a flash memory controller, that should ideally implement some wear-levelling algorithm, so that the memory areas are all evenly used.

Another feature that becomes particularly important when referred to hardware is the improvement towards component reusability: the process of creating masks for the mass production of ASICs is an expensive one, so it is a good thing being able to reuse a well-designed component. The advantage is that once we interconnect verified components, we do not have to worry about verifying their inner behaviour, but just make sure that we have connected them in a sound way.

A subtle property that requires a formal analysis to be proven is security. we can never be sure that there exists no input (or sequence of inputs) to a system that causes an undesirable behaviour (such as leaking data, that was not to be disclosed) unless we prove this formally.

1.3.1 Static Analysis of Circuits for Security

There are many factors that concur to the concept of *security*, and their relevance changes depending on the area of interest. Just to mention a few examples, security can mean *anonimity* in some applications while in some other can be *non-repudation*, sometimes security is *secrecy*, sometimes *integrity*.
This looks like a list of buzz-words — well, actually it kind of is — but it is just a short way to make the point, that we keep quite a lot of different things under the big hat of security: we will be talking more in detail about concepts relating to security in chapter 3.

For these reasons there are different security-related properties that we may want to prove on a security-oriented circuit, and obviously we are not claiming we are addressing all of them: the aim of this work is rather specific, *i.e.* tackling the problem of proving secrecy properties on circuits, that are described by VHDL code.
More specifically our goal is to develop a methodology to ensure that a circuit will not reveal some confidential data it contains, regardless of the way it is used — *i.e.* even if given to a malicious user, who

can send it whatever input he wants, there is no way to make the circuit output confidential data.
The challenge is to find a way to prove this directly from the VHDL code that describes it; there is no literature that addresses specifically this task, but something has been written on other topics that are somehow related to our goal, in particular:

- *type systems to analyse information flows*: type systems are a powerful tool to ensure that a system is treating data appropriately, as they aim at verifying that coherence is maintained throughout the system: if a system typechecks, it cannot be the case that a piece of data of a certain type is in a position where data of that type is not supposed to be. In particular type systems can be used to control information flows: there is some work oriented towards security issues in data flows [VIS96; LV05];

- *protocol verification*: this topic focusses on security protocols, an area where the verification approach can use typing rules to prove secrecy properties on protocols [Aba99]. These rules are to be applied on a model of the protocol expressed as concurrent processes: if it typechecks we can be confident that no confidential information will be disclosed because of a flaw in the protocol. In general all the wealth of work done in the area of protocol verification is an important resource to reach the goal we have in mind, and will be discussed extensively in chapter 3;

- *information flow analysis on VHDL code*: this topic addresses some of the issues we are interested in, as information flow analysis can be used to track the path of data within a circuit, and in particular the path of confidential data. This can be done by analysing the VHDL that describes a system: in [TNN05] we can find an analysis technique of this kind, that results in a transitive non-directed graph from which we can derive pretty accurate results on data flows that interest the circuit. In addition to "simply" tracking the flow of data, we want a technique that can be aware of the subtle mathematical properties of the functions that are used for security applications (for example the `xor` function): results in this area offer a good starting point to move towards our goal;

- *checking safety properties on VHDL code*: safety properties and security properties have quite a lot in common from a verification perspective, as they both require an effective way of modelling the system to be analysed. We can draw interesting ideas from the work done in this area [Hym02; Hym04], as the way VHDL code is modelled to prove safety properties can be readapted to serve our purpose, in particular for what concerns giving a semantics to VHDL — without forgetting some older work, not safety-related, that is specifically addressing the task of giving an operational semantics to VHDL [Goo95; TE01].

The remainder of this section is dedicated to giving a general understanding of the problem we are addressing, in terms of how the circuits we are dealing with are made and of how they can be modelled. The technical details are postponed to the relevant chapters.

Working Hypotheses

The working hypotheses underlying this work are the following:

- the only accessible terminals of the circuit are declared *a priori*, a malicious user cannot access any other part of it;

- no physical security issue is taken into account: physical behaviour (*e.g.* power consumption, voltage

surges, etc.) can help a malicious user break the system[13], but we assume that no attack exploiting such weaknesses can be successfully performed on the system;

- there may be some blocks which are assumed to work perfectly (*i.e.* their real implementation corresponds to their specification, so they behave *exactly* how they are supposed to): these are the black boxes in the black-box verification model.

It is worth spending a few words discussing the last hypothesis: in real world implementations circuits are very often implemented by assembling IP blocks[14]: as the VHDL code that describes these blocks is usually unknown (IP blocks are normally offered as netlists) the designer has to trust the vendor that the provided block has been verified and works according to its specification.
Thus an IP block is nothing but a black box to the designer: he only knows what output corresponds to a given input.

This development methodology, though avoiding errors regarding what is implemented in IP blocks (in line of principle), is still prone to design errors when assembling these blocks: a final verification step can make sure that this has not happened.

Data, Channels and Functions

Within a circuit we can distinguish between data that can be publicly known and data that need to be strictly confidential: a circuit leaking this second type of data has an unacceptable flaw.

During the normal operation of the circuit, we have flows of both types of data and we have to make sure that confidential data is never treated as public data — though we do not mind if public data is treated as confidential data.
Attention must be paid to what kind of data is input to the circuit and, especially, to what kind of data is output by the circuit: we must be able to ensure that secret data is adequately protected and prevented from being diffused outside the circuit.
This task is not an easy one, as we need to track each piece of data and follow accurately its path and its interactions with other pieces of data, as well as taking into account what the effects of function applications are.

In a circuit data are transported by means of wires, and can be input or output to terminals: we will treat all these in the same way, and to us they will be simply *channels*.
We divide the possible channels through which data is flowing into two kinds: some channels that are suitable to transport both types of data and some other channels that must not transport secret data, as they do not preserve the secrecy of data flowing in.
A malicious user has access to these latter channels.

We can further distinguish channels into three types: input channels, output channels, internal channels:

- the malicious user can write on input channels;
- the malicious user can read from output channels;
- the malicious user cannot do anything on internal channels, as he cannot access them.

[13] Attacks of this kind are called side-channel attacks — for more information on these attacks one can refer to [Sta10] and [Sal10].
[14] *Intellectual Property* blocks are reusable components — logic, cell or chip layout design — that are used increasingly often to speed up the development process of ASICs and FPGAs.

We can imagine that these channels are interconnected through particular gates: these gates are the functions that are applied to data.
Depending on the function, the type of a piece of data can eventually be changed, or different pieces of data are combined into a single piece (so the data type can potentially change in this case as well).

After merging all of these notions together we can build a model of the system we want to verify: a suitable way to prove the desired properties in this model can be a type system — if a system can be proved to be well-typed, we have the formal proof that it fulfils the security requirements it was demanded to.

Chapter 2

Circuit Synthesis through HDL

> A good scientist is a person with original ideas. A good engineer is a person who makes a design that works with as few original ideas as possible. There are no prima donnas in engineering.
>
> *Freeman Dyson*

Technology advancements have influenced the way hardware is designed: enough computational power has become available since the eighties and this has led to the implementation of new software tools to speed up (and improve) the task of hardware design.

Another reason why such new tools were needed was the crisis of VLSI design at the beginning of the eighties: circuits were becoming smaller and smaller and at the same time they were made of an increasing number of subcomponents (nowadays it is no wonder having chip with millions of transistors, but back in the eighties it was quite a great achievement), thus designing such complex systems by hand was becoming an increasingly challenging task, both in terms of avoiding errors due the large scale of the systems and for what concerns meeting the deadlines[1]. As a result of the adoption of these new tools, the time to market has reduced dramatically in comparison with what was achievable with the techniques based on designing schematics used till that moment, as the process was essentially a graphic one (the only tools available were essentially geometric software, and placement and routing tools started to be developed since the mid-seventies).

EDA[2] tools have been the result of this combination of technology push and market pull.
They have the ability to synthesize circuits automatically, starting from a description in a suitable format: in the early eighties *hardware description languages* (HDL) made their first appearance on the electronic-design scene, and were meant to provide a precise description of the hardware to be implemented.

HDLs are general purpose and can be used to describe very complex systems as well as simple circuits made of just a few gates.
The description of a system can be on different levels of detail (see figure 1.1) — here is an example where a binary adder is seen at some of the different possible levels [Rot98]:

[1] In a highly competitive environment as the electronic market, products have to be built as quickly as possible and in a cost-effective way — time is a factor just as important as cost.
[2] EDA stands for *Electronic Design Automation*.

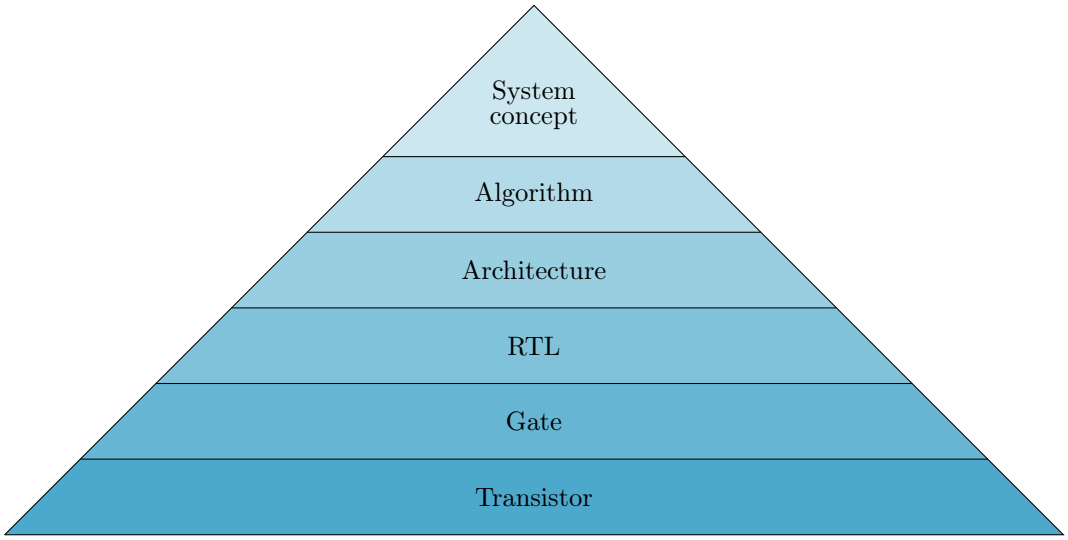


Figure 2.1: Top-down design. [Smi98]

- at the *behavioural level* (on the functional representation axis) it is described by a function that adds to binary numbers (no implemenation is given);
- at the *data flow level* (still on the functional representation axis, but on a lower level) the description is given by the logic equations for the adder;
- at the *structural level* (on the structural representation axis) the adder is specified by the gates it is made of and their interconnections.

Top-down design methodologies can be effectively adopted thanks to these tools, and allow designers to work at higher levels, focussing on the functionalities they want to implement, rather than on the logic- and transistor-level design (see figure 2.1): this is a way to meet the market needs, demanding shorter development cycles and increasingly complex capabilities.
This is also the only reasonable way to design *SoCs*[3], where the number of gates involved is commonly over one million gates — this has made possible for very complex systems to be synthesized on a chip, but also means that a schematic entry tool, besides being extremely error-prone for large-scale projects, is not a viable design method, as it would not be cost-effective, and extremely tedious and time-consuming.

The full description of a system in HDL results in a design that is portable among different EDA tools and independent of any particular silicon vendor's manufacturing technology.

The synthesis process comprises alternating steps of translation and optimization[4], that descend one abstraction level after the other until the very bottom level: this procedure extracts a netlist from a high-level description.

The behaviour of a component can be simulated and the result does not depend on the level of abstraction by which it is modeled — the simulation can be based on its HDL description as well as on its gate-level representation: this yields the same results.

[3] SoC stands for *System on a Chip*.
[4] There may be different kind of optimization that lead to different results: for example optimizing in order to achieve minimal area will deliver a different netlist then the case when the optimization criterion is maximal speed.

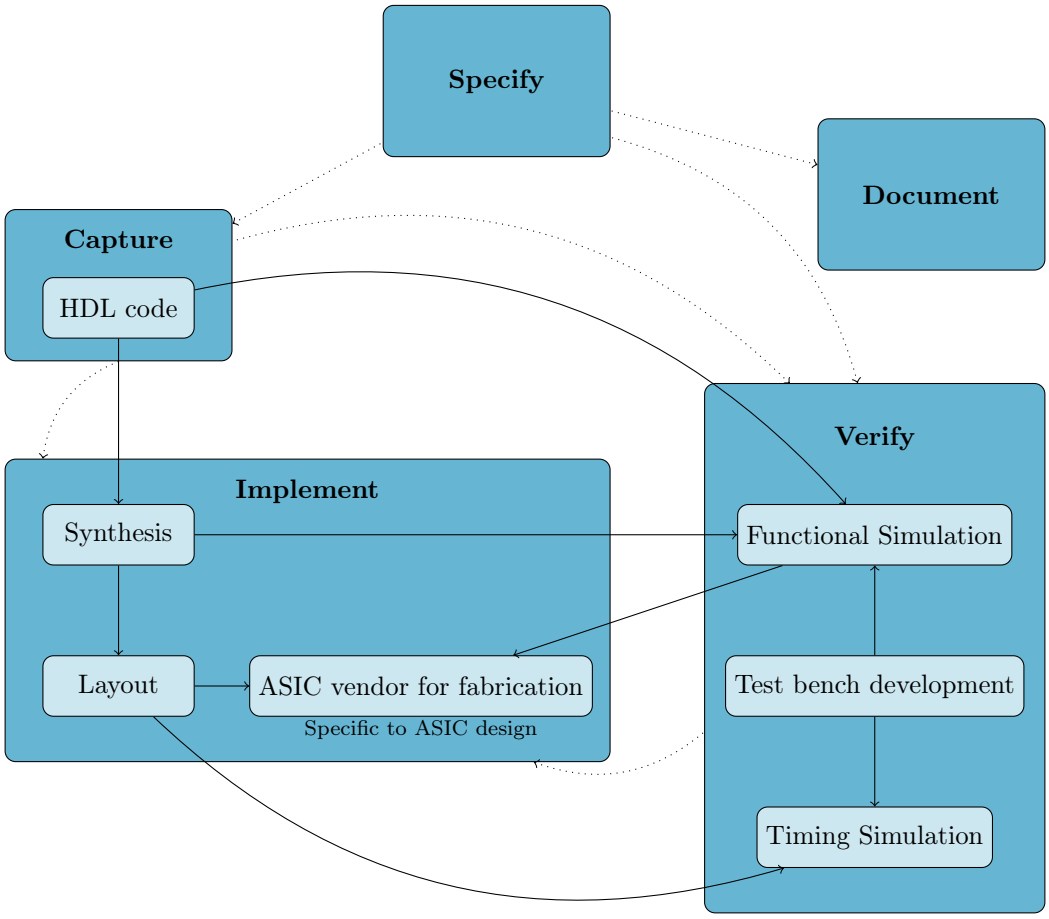


Figure 2.2: Design process. [PT97; Cha99]

Being able to simulate hardware has been a major achievement: the first, obvious reason is that it allows to detect bugs at early stages, rather than having to wait for the first prototype to be tested.
A second reason is that different design alternatives can be compared quickly by simulation.
A third reason is that it enables the designer to perform the so-called *fault simulation*, where typical manufacturing faults are injected into the model: when the final design is delivered to the manufacturer, the designer provides a set of test vectors which are to be used to test the final product, and the fault simulation aims at making sure that these test vectors are effective in detecting the most common manufacturing faults.

The design process can be divided into 5 macrosteps (see figure 2.2): the first one is the *specification* of the system that has to be designed, when the performance and interface requirements are formalized, without going down to implementation details; the second step is *capturing* the design, when the details of the system and its components are expressed through a computer-based design system, both as schematics and as HDL descriptions.
The following steps are interconnected: the *implementation* step starts with the synthesis from the HDL description, and this allows the *verification* of the design to start — *test benches*[5] are applied to the captured design, through functional simulation. Once this has been done, it is possible to proceed with

[5] A test bench is a performance specification for the circuit, that have to be developed during the design specification phase, and comprises descriptions of test vectors and corresponding outputs

the implementation and extract the layout of the synthesized system: after this the designer has some more verification to do, as it is possible to do the timing simulation.
If everything works as expected, *i.e.* the system complies with its specification, the layout can enter the manufacturing phase.
A *documentation* can be extracted from the specification of the circuit (usually before the end of the design process); being able to document adequately electronic system has been the reason pushing the project that led to VHDL (see the following section 2.1).

Summarizing, the advantages of this design approach are [Smi98]:

- increased productivity yields shorter development cycles with more product features and reduced time to market;
- reduced non-recurring engineering costs;
- possibility to reuse existing designs;
- increased flexibility to design changes;
- faster exploration of alternative architectures;
- faster exploration of alternative technology libraries;
- use of synthesis to rapidly sweep the design space of area and timing, and to automatically generate testable circuits;
- better and easier design auditing and verification.

There are two accepted industry standard HDL, namely VHDL and Verilog.
Although VHDL became a standard before Verilog did, they have the same expressive power[6], so which one is to be used is actually a matter of a designer's taste (or training): as we have already stated in the very beginning, we will be considering only VHDL, but whatever applies to VHDL is going to be applicable also to Verilog.

2.1 VHDL

VHDL stands for *VHSIC*[7] *Hardware Description Language* [Iee], and was a result of the VHSIC program, started in 1980 by the American Department of Defence: the purpose of this program was to make circuit design self-documenting, and it soon became clear that a standard HDL had to be provided to the subcontractors if this goal was to be achieved — this HDL had to be used to describe the structure and function of the designed systems and their components[8].
The development of VHDL began in 1983 with a joint effort of IBM, Texas Instruments and Intermetrics, under contract with the Department of Defence: this led to the definition of VHDL, that was later accepted from IEEE in 1987 as *IEEE Standard 1076*.

The 1987 standard is known as VHDL-87, later superseded by the 1993 and 2002 versions, known respectively as VHDL-93 and VHDL-2002, and by the current version that has been approved in 2008, known as VDHL-2008.

[6] Some tools translating from VHDL to Verilog are also available, such as Synapticad V2V.
[7] VHSIC stands for *Very High Speed Integrated Circuit*.
[8] The design of the F-22 tactical fighter, started in 1986, was one of the first major government project, where it was mandatory for subcontractors to use VHDL descriptions for all electronic subsystems.

Thus VHDL is a language that has been designed and optimised to describe digital systems and as such combines features of [PT97]:

- a *simulation modelling language*: the behaviour of electronic components can be modelled to a very precise level of detail through VHDL, and this can be used for systems of any size, ranging to simple circuits with only a few logic gates to complex ASICs and processors. Aspects that can be modelled include electrical aspects (rise and fall times, delays, and functional operation), so that it is possible to simulate the system to a high degree of precision;
- a *design entry language*: complex behavioural specification can be captured by VHDL, mixing low-level statements inherent to hardware with higher-level blocks, that remind conventional programming languages (in comparison with most programming languages, there is the added benefit of the possibility to describe concurrent statements — this is one big difference between hardware and software programming);
- a *verification language*: VHDL allows to capture test benches, and thus allows the designer to verify that the circuit behaviour complies with the requirements, both functional and temporal;
- a *netlist language*: VHDL can describe a circuit both at a high-level and at a lower level, and for this latter capability it can be used as a netlist language — for example it may be needed by tools that need to communicate at a low-level.

Hardware is represented in VHDL by means of a composition of building blocks, which are referred to as *design entities*: a design entity is a portion of the design, that has well-defined input and outputs, and that performs a precise operation.
It is defined by an *entity declaration* and a corresponding *architectural body*: the former defines the interface between the entity itself and the surrounding environment, the latter gives the actual description of what is contained in the entity, in particular it defines what relationship links inputs and outputs.
For each entity it is possible to have different architectural bodies, that represent the possible alternative implementations to instantiate that entity.

Configurations define the way entities are composed together into a design, by means of a *configuration declaration*.

An entity can be seen as being hierarchally organised into *blocks*, whereas the entity itself is the *top-level block*, which is made out of *internal blocks* (nothing but blocks of statements): sometimes the entity is referred to as *external block*, as it can be collected in a library and be used in different other designs.

A block of statements contains statements describing the internal organization and/or the operation of the block; the statements in a block execute concurrently and asynchronously.

Architecture bodies can contain also *processes*, which are a collection of actions that are to be executed sequentially (*sequential statements*).

In VHDL we have the familiar constructs of function and procedure declaration: as usual, a function is an expression that returns a value (there is the distinction between *pure* functions, that return a value which is deterministically dependent on the function arguments, and *impure* functions, that may return different values everytime they are called, even if they are passed the same arguments), whereas a procedure call is a statement.

Besides this behavioural description, it is possible to provide a structural description of an entity, by giving the port mappings that describe the connections among its subcomponents.
It is customary to use a mix of structural and behavioural descriptions, that results in a hybrid model.

VHDL is a large language and we do not need to take into account all of the features it offers, so — as customary in formal work regarding VHDL [Goo95; Hym02; Hym04; TNN05] — we will restrict ourselves to a subset of the available language constructs (which is also roughly the subset commonly used in real-world applications).

We will dedicate the remainder of this section to present the features of VHDL in finer detail, *i.e.* to the extent of what is necessary to develop an utility that parses actual VHDL[9].
We will be using the following notational conventions:

- reserved words are in typewriter font: `reserved_word`;
- syntactic categories are in italic sans serif: *syntactic_category*;
- an optional item is enclosed within square brackets: [*optional_element*];
- a comma-separated list of items, all of the kind *kind_of_item*, is written by enclosing such kind within angle brackets: ⟨*kind_of_item*⟩.

2.1.1 Lexical Elements

VHDL description are expressed as text files, that can contain any character from the ISO-8859-1 character set[10].

Identifiers A valid VHDL *identifier* must respect the following rules[11]:

- it must only contain alphabetic letters (without diacritics, *i.e.* from `A` to `Z` and from `a` to `z`), decimal digits and underscores;
- it must start with an alphabetic letter;
- it must not end with an underscore;
- it must not contain two successive underscores.

Identifiers are not case-sensitive.

Values Values that can be used in a VHDL description are:

- *numbers*: expressible in decimal format (also in exponential notation) and in an arbitrary base (with an eventual exponent)[12];
- *characters*: a character has to be enclosed in single quotation marks;
- *strings*: a string has to be enclosed in double quotation marks;

[9] This section is not meant to be a manual, some of the constructs may be simplified by omitting some optional parts. For more accurate detail on syntax, refer to the IEEE 1076 standard [Iee] or to one among the following references: [PT97; Rot98; Smi98; Cha99; Coh99; Ash08; Mäd09].
[10] Only ASCII character were supported in the early days of VHDL-87
[11] These are actually VHDL *basic identifiers*: since VHDL-93 *extended identifiers* are supported and can be any sequence of characters, enclosed between \ characters.
[12] In this case the number is in the format *base*`#`*number*`#[E`*exponent*`]`.

- *bit strings*: a bit string is a string containing only binary, octal or hexadecimal digits, which are to be expressed by putting respectively `B`, `O` or `X` before the opening double quotation mark[13].

Reserved Words We are not giving an exhaustive list of reserved words, as those which are relevant to the constructs within the scope of this thesis will be presented when each construct is introduced.

Operators VHDL operators include (but are not limited to): `not`, `and`, `or`[14], `+`, `-`, `<` and `=`:

op ::= `not` | `and` | `or` | `lnot` | `land` | `lxor` | `+` | `-` | `<` | `=` | `...`

Comments Single-line comments are introduced by a double hyphen (`--`) and comment out the text from there to the end of the line[15].

2.1.2 Data Types

The type of data defines the set of values that constants, variables, signals and files[16].
VHDL is a strongly typed language and offers many different data types (see figure 2.3): in the present work we will be using only a few of them, described in the remainder of this subsection.

Discrete Types

VHDL offers different discrete types, the ones we will be needing are *integers* and *booleans*:

- the integer type corresponds to the range $[-2\,147\,483\,647 .. +2\,147\,483\,647]$; it is possible to declare a custom type which is a subset of this range, via the reserved word `range`:

integer_type_definition ::= `type` *type_name* `is range` *lower_bound* `to` *upper_bound*
| `type` *type_name* `is range` *upper_bound* `downto` *lower_bound*

- the boolean type is a standard type comprising only two values, `true` and `false`. It is customary to use an enumerated type instead of booleans, namely the `std_logic` type, that has 9 possible different logical values and complies with the definitions in the IEEE standard 1164:

  `0` — strong drive, logic zero;

  `1` — strong drive, logic one;

  `X` — strong drive, unknown logic value;

  `L` — weak drive, logic zero;

  `H` — weak drive, logic one;

  `W` — weak drive, unknown logic value; ;

  `Z` — high impedance;

  `U` — uninitialized;

  `-` — don't care;

[13] VHDL-2008 has also other ways to express bit strings.
[14] All of these three are overloaded and can intended both as bitwise and as logical operators: in the definition we will split them, by putting a letter `l` before the logical operators.
[15] VHDL-2008 supports multiline comments, enclosed within `/*` and `*/`.
[16] These are the different kinds of object that are available in VHDL.

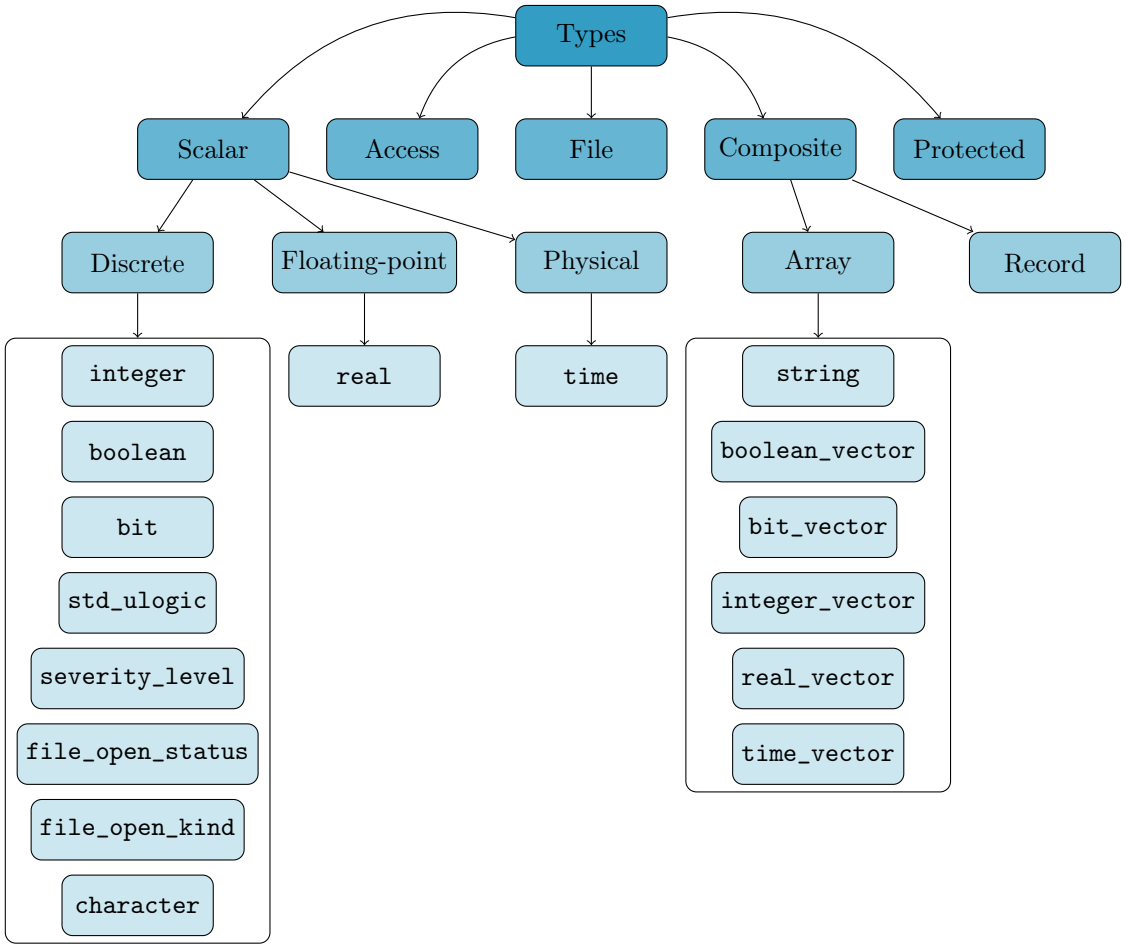


Figure 2.3: VHDL type classification. [Ash08]

Arrays

An array is a collection of values of the same type; each element in an array is identified by its *index*[17], which is a scalar value.
Therefore an array is a composite data type, and has to be declared via the reserved word `array`:

array_type_definition ::= `type` *type_name* `is array` *discrete_range* `of` *type*

Individual elements of an array are accessible by postponing `(`*index*`)` to the array name, where *index* identifies the position of the desired element in the array.

2.1.3 Expressions, Operations and Assignments

An *expression* is a combination of constant values, variables and signals, combined by operators. It can be *evaluated*, by replacing each variable by its current content and doing the maths.

[17] One-dimensional arrays have their elements identified by a single index, but in general it is possible to have n-dimensional arrays, where n different indices are necessary.

expression ::= *constant* | *variable* | *signal*
| *op expression* | *expression op expression*

Objects like constants, variables and signals can be assigned an expression or a value.

For constants and variables we have the following declarations:

constant_declaration ::= `constant` *constant_name* `:` *type* `:=` *expression*
variable_declaration ::= `variable` *variable_name* `:` *type* [`:=` *expression*]

Constant and variable declarations look very similar. The big hidden difference is that constants are declared in the very beginning and are not allowed to change their contents during the program execution: this difference is actually crucial, as constants are only a conceptual device for the designer, and will disappear at compile-time, when they will be replaced by the value they stand for.

Through an assignment, that is executed at run-time, we can change the contents of a variable:

variable_assignment ::= *variable_name* `:=` *expression*

For the purpose of this work we can think of a signal as the electrical equivalent of a variable, with the difference that the value[18] it bears is physically present on a wire or on a pin.
The signal assignment statement is very similar to variable assignment[19]:

signal_assignment ::= *signal_name* `<=` *expression*

2.1.4 Conditional Statements

VHDL has conditional statements, such as the conventional *if statement*[20], that executes a sequential statement if the condition it depends on is true.
Optionally it can offer one or more alternative codes to be executed if this condition does not hold:

if_statement ::= `if` *condition* `then` *sequential_statement*
[`elsif` *condition* `then` *sequential_statement*]
[`else` *sequential_statement*]
`endif`

We will give a definition of *sequential statement* in subsection 2.1.6.

[18] More generally a signal can bear a *waveform*, but for the application we are examining we are always dealing with a waveform that has constant value between signal assignments — that is the reason why in this casse there is a strong similarity with variables.

[19] To be precise, coherently with what we say in the preceding footnote, the proper definition should be *signal_assignment* ::= *signal_name* `<=` *waveform* [`after` *time_expression*], where the expression is substituted by a waveform and a time delay can be specified.

[20] It also has the *case statement* and, in VHDL-2008, also the *conditional assignment* and the *selected variable assignment*: we will consider none of the three for the purpose of this work, as all of them can be rendered through different if statements.

2.1.5 Loop Statements

When a sequence of actions has to be repeated, we can avail of loops, such as the *while loop*[21]: if a condition holds at the beginning of the loop, than the sequential statement it contains is executed, and this happens as long as the condition holds.
Obviously, if the condition does not hold when the program is supposed to enter the loop, the loop is skipped:

while_loop ::= `while` *condition* `loop` *sequential_statement* `endif`

2.1.6 Sequential and Concurrent Statements

In VHDL we distinguish between *sequential* and *concurrent statements*.
Sequential statements are either the constructs that have been presented so far or any *sequential composition* of sequential statements: the sequential composition operator is the colon `;` — this is a recursive definition:

sequential_statement ::= *variable_assignment*
| *signal_assignment*
| *if_statement*
| *while_loop*
| ...
| *sequential_statement* `;` *sequential_statement*

Concurrent statements are executed in parallel and are instantiated by processes in the entity body:

process_body ::= `process` [`(`*sensitivity_list*`)`] [`is`]
process_declarative_item
`begin` *concurrent_statement*
`end process;`

sensitivity_list ::= ⟨*signal_name*⟩ | `all`

We may give the following recursive definition for a concurrent statement:

concurrent_statement ::= *process_body* [*concurrent_statement*]

2.1.7 Entity Declarations, Architecture Bodies

The first step towards the design of an entity is its declaration, that describes the interface it presents to the system:

entity_declaration ::= `entity` *entity_identifier* `is`
`port` `(`*port_interface_list*`)`
entity_declarative_item
`end` [`entity`] `;`

[21] There are other different kinds of loop that are available in VHDL: we chose to give a definition of the while loop only, as every other loop can be seen as a particular while loop.

port_interface_list ::= ⟨*port_identifier* : [*port_mode*] *subtype_indication* [:= *expression*]⟩

port_mode ::= `in` | `out` | `buffer` | `inout`

The *entity_declarative_item* contains all type and constant declarations that are to be used in the entity.

To provide a code for an entity, we have to write its architecture body

architecture_body ::= `architecture` *architecture_identifier* `of` *entity_identifier* `is`
block_declarative_item
`begin` *concurrent_statement*
`end` [`architecture`] `;`

The *block_declarative_item* contains all of the declarations needed by the architecture body.

2.1.8 Structural Descriptions

Once we have coded all of the entities, they have to be instantiated to build a system.
The structure of such a system can be described in terms of subsystems, that are interconnected by signals (and this can go on recursively, referring to the structure of these subsystems). These interconnections are described via the port map statement when the component is instantiated:

component_instantiation ::= `entity` *entity_name* [`(`*architecture_identifier*`)`]
`port map` `(`*port_association_list*`)` `;`
port_association_list ::= ⟨[*port_name* `=>`] *signal_name* | *expression* | `open`⟩

2.2 *m*VHDL

*m*VHDL is the subset of VHDL we will be using: its syntax comprises only the construct that we have mentioned explicitly in the previous section[22].

The data types we will be considering are integers, booleans for truth values and a subset of `std_logic` — *i.e.* {`0`, `1`, `U`} — for logical values.

For convenience the syntax needed for concurrent statements in *m*VHDL is presented in figure 2.4.

[22] This is very similar to Hymans's MiniVHDL. [Hym04]

```
concurrent_statement ::= process_body [concurrent_statement]

process_body ::= process [(sensitivity_list)] [is]
                   process_declarative_item
                   begin concurrent_statement
                   end process;
sensitivity_list ::= ⟨signal_name⟩ | all

sequential_statement ::= variable_assignment
                       | signal_assignment
                       | if_statement
                       | while_loop
                       | sequential_statement ; sequential_statement

variable_assignment ::= variable_name := expression

signal_assignment ::= signal_name <= expression

if_statement ::= if condition then sequential_statement
                   [elsif condition then sequential_statement]
                   [else sequential_statement]
                   endif

while_loop ::= while condition loop sequential_statement endif

expression ::= constant | variable | signal
             | op expression | expression op expression

op ::= not | and | or | lnot | land | lxor | + | - | < | =
```

Figure 2.4: *m*VHDL sintax.

Chapter 3

Security and Protocol Verification

> Security, like correctness, is not an add-on feature.
>
> *Andrew S. Tanenbaum*

Information security is a problem that is always an actual one, and technological advancements have only changed the nature of the means to be used to secure information: having to deal with electronic information makes it necessary to provide the same tools that were available for pen-and-paper communications, that allowed to authenticate a message (a signature), to keep it confidential (an envelope), to ensure that it was not altered during the delivery process (a seal), and so on.

Such tools are necessary, because communications take place in a hostile environment, where we have *adversaries* (or *attackers*) besides legitimate *agents* and we have to deal with them: in the best-case scenario we are in the presence of a *passive adversary*, which cannot do much but eavesdrop from an unsecured communication medium; a more worrying scenario is when information security is challenged by an *active adversary*, who can also inject, modify or delete information from an unsecured communication medium.

Some of the necessary support to achieve the goals set by information security issues can be provided by cryptography, in the form of digital signatures to substitute hand-written signatures and authenticate a message, encrypting algorithms to make up for envelopes and provide confidentiality, techniques to ensure data integrity to replace seals, and so on.

This can be used in an electronic communication protocol, in order to preserve security properties of data being transferred: the scenario we are dealing with is a communication where two entities interact through a *channel*, which can be either *secure* or *unsecured*, depending on what an adversary can do with it.
The communicating parties are the entities that access the communication channel, and the legitimate agents are usually referred to as the *sender* and the *receiver*, depending on the direction of the information flow.

3.1 Cryptography

Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, authentication, and non repudiation:

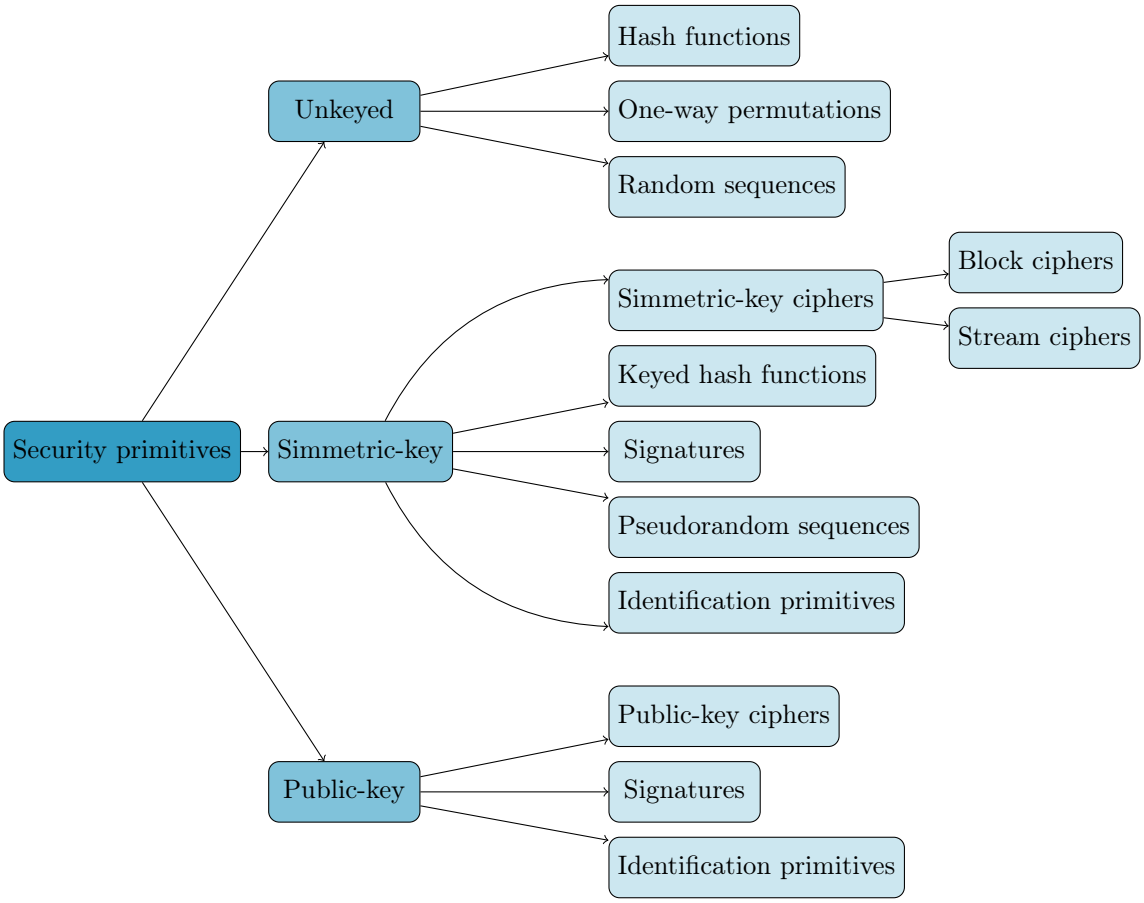


Figure 3.1: Security primitives. [MVO96]

- the aim of *confidentiality* is to maintain secrecy of data, so that it can be disclosed only to those who are entitled to access it;
- we are interested in making sure that transmitted/received data is not subject to any kind of alteration from a third party: *data integrity* is the service that we require to take care of this;
- electronic information can be duplicated easily and effortlessly: for this reason we need a way to identify data (*data origin authentication*) and those who manipulate it (*entity authentication*);
- finally we are interested in *non-repudiation*: this means that an entity cannot deny an action or a commitment.

The building blocks of cryptography are sometimes called primitives, and are shown in figure 3.1.

3.1.1 One-way Functions

Not all invertible functions are equal from a cryptographic perspective: an important role in cryptography is played by those which are straight-forward to compute, but for which it is computationally infeasible to compute the inverse function.
Such functions are usually referred to as *one-way functions* — one of the best known examples is the exponentiation over integers: there exists no efficient algorithm to compute discrete logarithm (yet).

A particular class of one-way functions which is often used in cryptography is that of *hash functions*: a hash function maps a string of arbitrary length to fixed-length binary string (*hash*), which has the property that it is infeasible to find two different inputs that have the same hash (*colliding* inputs), besides being efficient from a computational perspective.

Trapdoor one-way functions constitute a particular class of one-way functions, that have the property of being invertible in a computationally feasible way if some extra information (called *trapdoor information*) is available.

Finally another kind of functions we will be talking about are *involutions*, which are function which enjoy the property of being their own inverses.

3.1.2 Encryption Schemes

With an *encryption scheme* we aim at creating bijections between the *message space* $\mathcal{M}$, containing *plaintexts*, and the *cyphertext space* $\mathcal{C}$, containing *cyphertexts*: they both contain strings, formed of symbols from an alphabet $\mathcal{A}_\mathcal{M}$ or $\mathcal{A}_\mathcal{C}$ respectively — genereally speaking they can be different, but usually they are simply the alphabet $\mathcal{A} = \{0, 1\}$.

Encryption and decryption transformations (D_d and E_e) are bijections which are uniquely determined by keys d, e taken from the *key space* $\mathcal{K}$, and they relate elements from the spaces $\mathcal{M}$ and $\mathcal{C}$.
The key space has to be large enough to prevent an adversary to find the keys that have been used, thus identifying the encryption and decryption transformations, through exhaustive search.

An encryption scheme consists of two corresponding sets of encryption and decryption transformations, such that for every encryption key e there exists a decryption key d that allows to recover any message m that has been encrypted:

$$\forall m, e\, \exists d \bullet D_d\big(E_e(m)\big) = m$$

Kerckhoffs's desiderata date back to 1883, but are still valid today (with minor changes) and set requirements that should be satisfied by an encryption scheme:

1. the system should be, if not theoretically unbreakable[1], unbreakable in practice;
2. compromise of the system details should not inconvenience the correspondents;
3. the key should be rememberable without notes and easily changed;
4. the cryptogram should be transmissible by telegraph;
5. the encryption apparatus should be portable and operable by a single person;
6. the system should be easy, requiring neither the knowledge of a long list of rules nor mental strain.

Here is a non-exhaustive list of the most common attacks that can be mounted against an encryption scheme:

- a *ciphertext-only attack* is one where the adversary tries to deduce plaintext (and/or the decryption key as well) by only observing ciphertext;

[1] A system is said to be *breakable* if plaintexts can be systematically recovered from corresponding cyphertexts without any knowledge of the key pair (e, d) in a reasonable time.

- a *known-plaintext attack* can be mounted if an adversary has a fair amount of plaintext and corresponding ciphertext;
- a *chosen-plaintext attack* is one where the adversary can be given ciphertext that corresponds to some plaintext of his choice, and this information is used to decrypt other ciphertext;
- an *adaptive chosen-plaintext attack* is a chosen-plaintext attack, where the attacker has a strategy that guides him in choosing the plaintext, based on the previous requests;
- a *chosen-ciphertext attack* is one where the adversary selects the ciphertext and is then given the corresponding plaintext, and this information is used to decrypt other ciphertext by other means;
- an *adaptive chosen-ciphertext attack* is a chosen-ciphertext attack, where the attacker has a strategy that guides him in choosing the plaintext, based on the previous requests.

Symmetric-key Encryption

Symmetric-key schemes are those for which the encryption key e is equal to the decryption key d — we usually call this simply *secret key*. For extension this may designate also schemes where one key is easily derivable from knowledge of the other key, *i.e.* it is possible to derive the message $m \in \mathcal{M}$ that corresponds to a given cipehertext $c \in \mathcal{C}$ only by knowing the encryption key e.

Symmetric-key schemes are usually divided into two classes: *block ciphers* and *stream ciphers* (although the latter can be seen as a special case of block ciphers).

A block cipher splits a plaintext into several blocks of fixed length and encrypts them one at a time — most well-known symmetric-key encryption techniques are block ciphers.

Stream ciphers can be seen as a special case of block cyphers, as the message is encrypted one symbol after the other: it is like having blocks of unitary length.
Usually there is a different encryption key for every symbol — an example for this is the famous *Vernam cipher*.

Public-key Encryption

In public-key cryptography for every pair of encryption/decription trasformations (E_e, D_d) it is not possible to determine the decryption key d with the knowledge of the corresponding encryption key e, *i.e.* it is not possible to derive the message $m \in \mathcal{M}$ that corresponds to a given cipehertext $c \in \mathcal{C}$ even if the encryption key e is public domain.
The encryption transformation E_e is thus a trapdoor one-way function, and the decryption key d is the trapdoor information.

Usually the encryption key e is made publicly available (and for this reason is usually referred to as *public key*), so that anyone can send an encrypted message to the recipient holding the decryption key d (or *private key*).

Public key cryptosystems that are used in practice are *reversible* ones, *i.e.* those where $\mathcal{M} = \mathcal{C}$ and which have the following extra property:

$$\forall m, (e, d) \bullet D_d\big(E_e(m)\big) = E_e\big(D_d(m)\big) = m$$

Public-key cryptography is less efficient than simmetric-key encryption in terms of data throughput, as the algorithms it uses are more complex and have to deal with longer keys; nevertheless it has the great advantage that only private keys have to be kept secret and are not shared among different entities — for this reason keys need not be changed so often.
In some applications we take the best of both worlds: when two entities want to communicate, they may use public-key cryptography to agree on a symmetric-key, that they will use for a limited time to communicate more efficiently.

Last but not least, public-key cryptography is much younger than symmetric-key cryptography: an example of this is given by the fact that there is no public-key scheme that has been proven to be secure, and its security is presumed on the basis of the complexity of a problem from number theory.

3.1.3 Digital Signatures

A digital signature scheme aims at binding the identity of an entity to a piece of information.

The *signing transformation* S_A maps a message $m \in \mathcal{M}$ to a signature s in the signature space $\mathcal{S}$ — this transformation is to be kept secret by entity A.

The *verification transformation* V_A returns a boolean value when applied to a couple (m, s): this transformation is public domain and allows anyone to verify whether s is A's signature for message m, *i.e.* if $V_A(m, s) = \mathit{True}$.

To be protected against forgery, it must be infeasible to compute s for any message m, for anybody that does not know the signing transformation S_A.

An adversary may attack a signature scheme in order to forge signatures in a way similar to the possibilities described for encryption schemes.

3.1.4 Authentication

When talking about authentication (or *identification* as well) we distinguish between *entity authentication* and *data origin authentication*.

Entity authentication aims at assuring an entity (the *verifier*) of the identity of a second entity involved (the *claimant*), and that this second entity was active at the time the authentication happened.

Data origin authentication (or *message authentication*) identifies the entity that has originated the message — it must be noticed that this kind of authentication provides no timeliness guarantee, whereas entity authentication does.

An adversary may attack authentication procedures in order to forge authentication data in a way similar to the possibilities described for encryption schemes.

3.1.5 Protocols

A (cryptographic) *protocol* allows two or more entities to achieve specific security objectives through a precise sequence of steps and interactions, that involve exchanging messages having definite formats and behaving according to certain rules — this is usually described in a *protocol standard*[2].

[2]The main and most widely used internet protocol standards are developed and maintained by working groups from the *Internet Engineering Task Force* (IETF).

Here is a non-exhaustive list of the most common attacks that can be mounted against a protocol:

- a *known-key attack* enables an adversary to determine some keys once he has obtained some other keys, which have been used in a different context. A particular kind of attack targets current keys in order to compromise past keys: a protocol that is immune to such an attack offers *perfect forward secrecy*;
- a *replay attack* can be mounted if an adversary can avail of a record of a past communication session and replays a part of it during the attack;
- an *impersonation attack* happens when an adversary pretends to be a legitimate entity and takes part in the communication;
- a *dictionary attack* is an attack that tries to guess a password by trying all of the possible ones.

3.1.6 Models for Security Evaluation

There is not a single concept of security and this is reflected in different models, that can be used to evaluate security according to the different conceptions:

- we talk about *unconditional security* when a system cannot be broken by no means from a theoretical perspective, *i.e.* even an attacker with unlimited computational power cannot do anything to mount a successful attack to the system, as there is not enough information that can be exploited.
- in the case where we face an adversary with computational power that is polynomial both in time and space, we use a model for *complexity-theoretic security*: this model will then consider both worst-case and asymptotic analysis in this setting, in order to find if any attack are feasible in this model — these attacks may be computationally feasible also in the real world;
- for *provable security* we are required to show by a formal proof of equivalence that the difficulty of breaking a system is essentially equivalent to the difficulty to solve a well-known and supposedly difficult problem — this approach is the one used in practice whenever possible;
- a generalization of provable security is given by *computational security*: this provides a measure in terms of the computational effort that is required to break a system, when using the best-known algorithms, thus we can say that a system is safe when this effort is not sustainable by the computational power of the attackers the system is going to face;
- the (weaker) approach of *ad hoc security* consists in a bit of hand-waving, to show that every successful attack that could be mounted against a system would require a resource level greater than the fixed resources of the expected adversary. We also talk about *heuristic security* in this case.

3.2 Protocol Verification

In this section we will be addressing protocol verification in finer detail: we will need to do information flow analysis to achieve the goal set for this thesis and we will be using techniques from the domain of protocol verification.

In the last years research has strongly focused on proofs of security. The verification step to ensure that a computer program or a protocol have certain requested properties is a crucial one, and this task has to

be done preferentially by formal reasoning rather than by tests and simulations, as the latter approach is not as exhaustive as the formal one.

For a quick overview on the state of the art, one can refer to the recent survey by Abadi, Blanchet, and Comon-Lundh [ABCL09].

There are two possible approaches to protocol verification: the formal model and the computational model.
In the first model, we are in a highly idealized setting, whose properties can be expressed through logic and manipulated with formal techniques (for example rewriting rules or theorem proving), and therefore this can be effectively implemented in fully-automated protocol verifiers.
This is the so-called Dolev-Yao model [DY83], presented in a paper dating back to 1983, which assumes that:

- the net is under the intruder's control (see figure 3.2): messages can be intercepted and altered. New messages can be injected to the net;
- the cryptographic primitives are perfect;
- the protocol admits any number or participants and any number of parallel sessions;
- the protocol messages can be of any size.

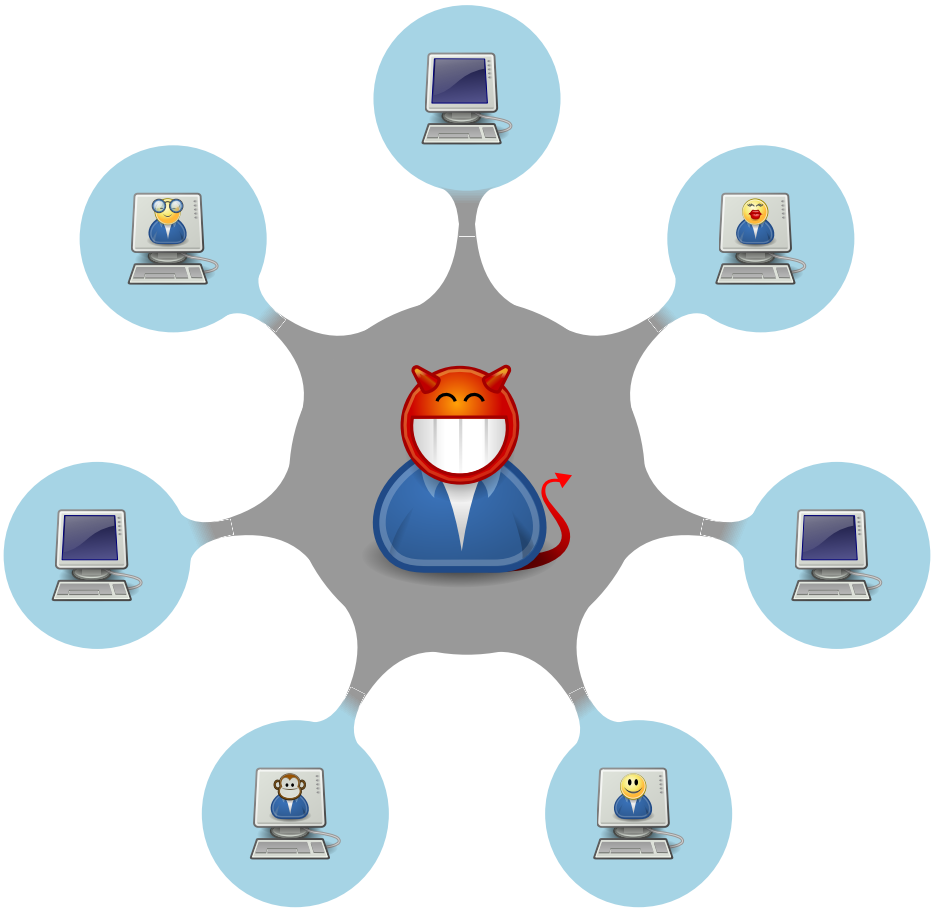


Figure 3.2: The Dolev-Yao model.

In this model we can reason about an idealized version of the protocol, so we can abstract from the implementation issues: for example a flaw in an implementation of a protocol due to overflow will not be detected in the formal model, but a flaw due to misconception of the protocol will be found by a protocol verifier.

The second approach adopts a computational perspective, focusing on the actual computations underlying a protocol, and borrows ideas from complexity theory. It requires much more human intervention in proofs, and is only recently being automated.
Bridging the gap between proofs in the formal model and in the computational one is one of the current issues in protocol verification[3], but we will not focus on this second approach any further, as we will be using a purely formal approach.

These verification techniques allow us to uncover design faults that may remain hidden for years. There are a lot of examples that can be recalled on this topic, for example a famous successful application of verification in the formal model can be found in a work by Lowe [Low95; Low96]: the popular Needham-Schroeder protocol dates back to 1978, but it was just in 1995 that he found that a *Man-in-the-Middle* attack can effectively be mounted against this protocol and proposed a modification. To achieve this goal Gavin Lowe used the FDR tool , which is a model checker for CSP.

Besides generic model checkers, there are tools which have been conceived specifically with communication protocols in mind. An example is one of the tool we will be using, *i.e.* Bruno Blanchet's ProVerif : if the original Needham-Schroeder protocol is analysed with this tool, this same security flaw can be uncovered and a trace of the attack given.

A technique to model cryptographic protocols sees them as interacting processes, and we use *ad hoc* languages to describe this model: in the next chapters we will focus on the applied π-calculus and on logic clauses, as means to describe such processes: this is going to be our approach as well.

Besides this there are other interesting options, but we will not go into details as they are outside the scope of this thesis; nevertheless we will mention quickly two current research subjects that have to be kept in mind, in case of future extensions of this work:

- a probabilistic calculus for cryptographic protocols would add probability and statistics to the picture, which are factors that have to be taken into account to drive security analysis closer to real-world issues;
- lately part of the research community has started to look at secure refinement calculus as a new technique to be applied to security problems and this could extend the range of security applications that can be verified.

[3] Abadi and Rogaway [AR02] present a computational-soundness theorem, that relates the two views: through this theorem it is possible to relate formally equivalent terms with computationally indistinguishable terms.

Chapter 4

Process Calculi

This chapter has the potential of being very technical, nevertheless we will try to keep technicalities down to a minimum level, stripping the theory to the bone and simplifying things as much as possible, in order to present the general ideas and give the foundations to understand the work done, while avoiding all of the harsh details — for the interested reader, there is a great wealth of literature in this domain, and for example one may refer to *Communicating and mobile systems: the π-calculus* by Milner [Mil99], where the author provides an overview on the subject and introduces the π-calculus, or to *Reactive Systems: Modelling, Specification and Verification* by Aceto et al. [Ace+07] as well, which is more focused on process calculi and reactive systems but leaves out the π-calculus.

4.1 Automatas and Labelled Transition Systems

Every electronic engineer has dealt with *finite automatas* at some stage, perhaps without knowing it: he is usually fed something that goes more often under the name of *finite state machine* (FSM) — this is for sure a more familiar name[1].

Now that we have made clear what we are talking about, if we see things from a more formal perspective

[1] At least this was my personal experience!

we can say that an automaton A over a set of *actions* $\mathtt{Act}$ is:

$$A \triangleq (\mathcal{Q}, q_0, \mathcal{F}, \mathcal{T})$$

where

- $\mathcal{Q}$ is the finite[2] set of *states*;
- $q_0 \in \mathcal{Q}$ is the *start state*;
- $\mathcal{F} \subseteq Q$ is a set of *accepting states*;
- $\mathcal{T} \subseteq \mathcal{Q} \times \mathtt{Act} \times \mathcal{Q}$ is a set of *transitions* — a transition (q, a, q') can be read as "updating" the current state q to the state q', after performing the action a.

We are in the case of a *deterministic* automaton if for every pair $(q, a) \in \mathcal{Q} \times \mathtt{Act}$ there is exactly one state q' such that $(q, a, q') \in \mathcal{T}$, *i.e.* q' is univocally determined by (q, a) — this is the case we will be dealing with, so we will not address all issues that concern non-determinism.

A common representation for an automaton is a *transition graph*: an example is given in figure 4.1:

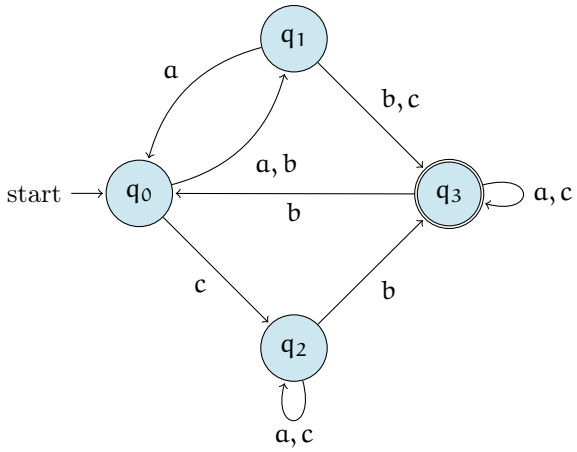


Figure 4.1: A finite-state automaton $A = (\mathcal{Q}, q_0, \mathcal{F}, \mathcal{T})$ over $\mathtt{Act} = \{a, b, c\}$, where $\mathcal{Q} = \{q_0, q_1, q_2, q_3\}$ and $\mathcal{F} = \{q_3\}$ — $\mathcal{T}$ can be inferred trivially from the graph.

There may be different sequence of actions (*strings*) that can lead to an accepting state: the set of all such strings is called the *language* of the automaton.
If we go back to think of an automaton as a FSM, the language of an automaton is the collection of all possible behaviours of the FSM.

By generalizing the concept of automaton to the case where we remove the notions of start and accepting state, we have the definition of a *labelled transition system* (LTS):

$$S \triangleq (\mathcal{Q}, \mathcal{T})$$

From a LTS we can extract a different automaton by choosing a starting state in $\mathcal{Q}$, where every state is an accepting state.

[2] In general we can also have the case where $\mathcal{Q}$ is not finite.

4.2 Sequential and Concurrent Processes

When we select a starting state in a LTS we have a *sequential process*: it can perform a sequence of actions (*trace*), and the structure of the process accounts for all of the possible execution traces.

Processes can also run concurrently and interact: this implies that a process can perform an *internal action* (which cannot be observed from outside the process) or react to an *external action*.

Along this distinction among actions, we introduce also the concept of *complementary actions*: for example for each write operation we can have a read operation which "complements" it, similarly for each send operation we can have a receive operation.
Such actions are very important, as concurrent processes synchronize on such actions (*handshake*): one process cannot write if another process is not willing to read, similarly a process cannot send something if there is no process receiving that message — these are *blocking actions*, as a process cannot continue running if any such action has not been completed.
If we look at two concurrent processes as to a single larger process, a handshake is an internal action and it cannot be seen from the "outside world": for this reason we sometimes talk also of *unobservable actions* in this case.

There are different concepts of equivalence that can be used to compare processes: we are not going to present all of the different concepts that can be found in the literature, we will rather restrict ourselves to the cases of interest to the present work.
Such cases are (from the most restrictive criterion to the most general):

- *strong bisimulation*, where two processes P and Q are said to be equivalent ($P \sim Q$) if their states can be related in a way that each couple of related states can perform the same actions, which lead to states that are related;
- *weak bisimulation* is a generalization of this concept, where we talk about *weak actions* instead of usual actions ($P \approx Q$) — a weak action is a (possibly empty) sequence of internal actions with at most one interleaved external action;
- *trace equivalence* is an even weaker criterion, as the only requirement for two processes to be regarded as equivalent is to be able to perform the same sequence of (weak) actions.

We are not going into any deeper detail about processes: this very brief presentation should be enough to get intuitively the basic ideas, although a deeper understanding of this subject is needed to appreciate the technicalities underlying the present work — please refer to the books suggested in the introduction to this chapter.

4.3 The π-calculus

The π-*calculus* is a kind of process calculus that focuses on the concept of *mobility*, intended as a process capability to create and change *links*: this feature overcomes some of the limitations of previous process calculi, and makes it an interesting language to model complex behaviours — including (but not limited to) those of computer networks and mobile phone networks, just to mention a couple.

The syntax, also shown in figure 4.2 for convenience, describes the way a process is built:

$\overline{a}\langle b\rangle.P$: a process can *output* the content of b on a channel a, for some other process to input for that channel, and then behaves like the process P;

P,Q ::=		Processes
|	$\overline{a}\langle b\rangle.P$	Output
|	$a(b).P$	Input
|	$\tau.P$	Unobservable action
|	0	Nil process
|	$P\|Q$	Parallel composition
|	$!P$	Replication
|	$(\nu a).P$	Restriction

Figure 4.2: π-calculus sintax, where a,b and c are *names*.

$a(b).P$: a process can *input* some content from a channel a and bind it to b, and then behaves like the process P, which can obviously avail of the content of b;

$\tau.P$: a process can take an *unobservable action*, and then behaves like the process P;

0 : the most basic process is the *nil process*, which simply does nothing;

$P|Q$: a process can be made of two processes P and Q running in parallel, which can eventually communicate and synchronize;

$!P$: a process can be made of infinite instances of the same process running in parallel;

$(\nu a).P$: a process can define a *new name* a and make it available exclusively to its continuation P.

4.4 The Applied π-calculus

The features of π-calculus have gathered attention from the community that is involved in security and protocol verification, so some extension of this language have started to appear on the scene.

The *spi-calculus* is formalised in a paper by Abadi and Gordon [AG97] as an extension of Milner's π-calculus, where the authors add cryptographic primitives. Nevertheless something is still missing to make it a satisfying tool for protocol verification. For this reason the spi-calculus can be regarded as a first step towards the *applied π-calculus* [AF01].

The applied π-calculus is the standard π-calculus with the addition of the capability to express functions and equations, as well as the possibility of sending more complicated terms through channels. For this reason the authors felt the need of adding also a way to declare a short name for a more complicate expression (a kind of substitution).
A crucial improvement with respect to the spi-calculus is the possibility to define custom cryptographic primitives, whereas in the spi-calculus they are fixed *a priori*.
Destructors and error handling have later been embedded in the applied π-calculus: the resulting syntax is shown in figure 4.3.

It is apparent from the syntax that constructors and destructors are two categories of function symbols: this is to stress the fact that there *may* be a function (the destructor) which in some sense undoes what another function (the constructor) has done — formally the application of a constructor returns a term, which may be manipulated in the process by a destructor, and this is the reason why the constructor application appears in the "Terms" part of figure 4.3 while the destructor application does not.

$M,N ::=$		Terms
	x,y,z	Variables
	a,b,c	Names
	$f(M_1,\ldots,M_n)$	Constructor application
$P,Q ::=$		Processes
	$\overline{M}\langle N\rangle.P$	Output
	$M(x).P$	Input
	$\text{let } x = g(M_1,\ldots,M_n) \text{ in } P \text{ else } Q$	Destructor application
	$\text{if } M = N \text{ then } P \text{ else } Q$	Conditional
	0	Nil process
	$P\|Q$	Parallel composition
	$!P$	Replication
	$(\nu a).P$	Restriction

Figure 4.3: Applied π-calculus sintax.

Moreover the possibility that the application of a destructor may fail is explicitly taken into account, as it is possible to specify the process that is executed in case of failure.

Such improvements make the applied π-calculus a versatile tool that can be used in a variety of cases and for a variety of different purposes: the next subsection will present Bruno Blanchet's *ProVerif* as an application using the applied π-calculus towards protocol verification — we are interested in this application as in chapter 6 we will describe a way to use the same approach to verify VHDL descriptions of electronic components.

4.4.1 ProVerif

ProVerif is a cryptographic protocol verifier, which is based on the Dolev-Yao model [DY83], described in section 3.2; the results this tool delivers provide information not only about secrecy properties, but also correspondence[3] or observational equivalence properties.

It has the capability to handle a non-limited number of sessions and a whole variety of cryptographic primitives, to be defined either through equations, either by means of rewriting rules.
It is useful to show some examples:

- in the case of *symmetric key encryption* if we encrypt the message m under the shared key k, we obtain the term $\texttt{encrypt}(m,k)$, where `encrypt` is the constructor; the destructor `decrypt` has the property that $\texttt{decrypt}(\texttt{encrypt}(m,k),k) = m$ — if an agent can encrypt a message, he is also able to decrypt it;

- in the case of *public key encryption* the situation is slightly more complicated than this: we have a pair of keys, one of which is to be kept private, k, and the other one is to be made publicly available, $\texttt{pk}(k)$ — `pk` is the constructor that builds a term that accounts for the public key that correspond to the private key k: we can model the non-reversibility of this function[4] by not providing any destructor to decompose terms built by `pk`. If we encrypt the message m under the public key $\texttt{pk}(k)$, we obtain the term $\texttt{encrypt}(m,\texttt{pk}(k))$; the destructor `decrypt` has the property that

[3] A *correspondence property* is a property like "if a certain event has happened, that it must be the case that some other events have happened".
[4] We remind that it is assumed not to be possible to recover a private key from the corresponding public key.

$\texttt{decrypt}(\texttt{encrypt}(\texttt{m}, \texttt{pk}(\texttt{k})), \texttt{k}) = \texttt{m}$ — the fact that an agent can encrypt a message does not yield that he is also able to decrypt it;

- in the case of a *signature* based on the public key scheme described above, we can build a signature for a message $\texttt{m}$ through the constructor $\texttt{sign}$ and have the term $\texttt{sign}(\texttt{m}, \texttt{k})$. The verification of the signature can be made with the public key $\texttt{pk}(\texttt{k})$: $\texttt{check}(\texttt{sign}(\texttt{m}, \texttt{k}), \texttt{pk}(\texttt{k}), \texttt{m}) = \mathit{True}$. It must be noted that no destructor is provided to recover neither $\texttt{m}$ nor $\texttt{k}$ from the signature;
- generally speaking, in the case of any one-way function, such as a *hashing function*, no destructor is to be provided: from the term $\texttt{oneway}(\texttt{x})$ it must not be possible to recover $\texttt{x}$.

The processes describing the agents interacting as part of the protocol have to be expressed either in the applied π-calculus or in the form of *Horn clauses*, which is a sort of lower level representation[5] as ProVerif translates internally an applied π-calculus description in this form before analysing the protocol.

Roughly speaking ProVerif keeps track of the knowledge of an attacker: at the very beginning his knowledge pool is made of all things that are public domain, such as the protocol description, the constants that are mentioned in such description, the functions that can be applied and so on.
As the protocol runs he can gather all of the information that flows through public channels, as well as all of the information that he can derive from that (such as a message encrypted with a key the attacker already has or as decrypting a previously sent message once the decryption key becomes available). Moreover the attacker can interact with the agents in any way that is allowed by the protocol and can eventually drive them to disclose confidential information.
When we want to verify whether a protocol preserves the secrecy of a term, we demand that it will not be part of the attacker's knowledge at any moment.

Through ProVerif we can verify more subtle properties: the present work is focused on proving secrecy properties starting from VHDL descriptions of electronic circuits, nevertheless our approach is a general one and, for the part that involves the use of ProVerif, can be extended to verify all properties that are verifiable through ProVerif.

[5] We will present them more accurately in section 5.1.1, where we will also show how ProVerif deals with them as an example.

Chapter 5

Logic

> Contrariwise, if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic.
>
> *Lewis Carroll*

Logic is something that everybody has always been dealing with, in situations that range from the non-formalised setting of everyday's life to those that see logic as a precise discipline, at different degrees of complexity since primary school through doctoral studies and research.
Our focus is *propositional logic* in particular and we assume that the reader is familiar with all related basic notions — we do not need much more than that, so will only define the more specific terminology we are going to use.

For a formal presentation of logic, one can refer to:

- Harry J. Gensler. *Introduction to Logic*. Routledge, 2010
- Warren Goldfarb. *Deductive Logic*. Hackett Publishing, 2003
- Stephen Cole Kleene. *Mathematical Logic*. Dover Publications, 1967
- Alfred Tarski. *Introduction to Logic and to the Methodology of the Deductive Sciences*. Oxford University Press, 1994

5.1 Logic Clauses

A *predicate*, which is a statement that can be either true or false, is the atomic component of a logic clause and it is itself the simplest logic clause.

If we negate a logic clause with the negation operator $\neg$, we obtain another logic clause.

We can build more complex logic clauses by using one of the following *logical connectives* to join any two clauses: *conjunction* ($\wedge$), *disjunction* ($\vee$), *implication* ($\Rightarrow$), and *double implication* ($\Leftrightarrow$).
We are using the standard definitions for these operators.

5.1.1 Horn Clauses and ProVerif

Horn clauses are a subset of logic clauses, which are characterised by the following shape:

$$\bigwedge_{i=1}^{N} P_i \Rightarrow Q$$

From a computer science perspective they are very interesting, as there are resolution algorithms that are applicable to a set of Horn clauses — and ProVerif is an example of this: we will now give a brief overview of the way ProVerif operates, as our approach is quite similar.

ProVerif uses Horn clauses which are built on two predicates:

- the predicate $\texttt{attacker}(x)$ states that x is the knowledge pool of the attacker;
- the predicate $\texttt{message}(m, c)$ states that the message m is on a channel c.

By using these predicates, we can build clauses that account for the attacker's behaviour, and in particular for:

- the attacker knowing all of the protocol constants, so for each constant a we have a corresponding clause $\texttt{attacker}(a)$;
- the attacker being able to create new names, so we are entitled to add a clause $\texttt{attacker}(b)$ for any name that is not mentioned anywhere else in the processes representing the protocol;
- the attacker being able to apply any n-ary constructor f, which means that if he has n terms he can use them as arguments for f — $\texttt{attacker}(x_1) \wedge \texttt{attacker}(x_2) \wedge \ldots \wedge \texttt{attacker}(x_n) \Rightarrow \texttt{attacker}\big(f(x_1, x_2, \ldots, x_n)\big)$;
- being able to apply an n-ary destructor g (whenever possible), which means that if he has an n-uple of terms M_i that belong to the domain of g he is able to apply it and obtain $M = g(M_1, M_2, \ldots, M_n)$ — $\texttt{attacker}(M_1) \wedge \texttt{attacker}(M_2) \wedge \ldots \wedge \texttt{attacker}(M_n) \Rightarrow \texttt{attacker}(M)$;
- the case when a message m flows through a channel c which the attacker has access to, then the attacker can read the message — $\texttt{message}(m, c) \wedge \texttt{attacker}(c) \Rightarrow \texttt{attacker}(m)$;
- the attacker being able to send a message m on a channel he can access — $\texttt{attacker}(m) \wedge \texttt{attacker}(c) \Rightarrow \texttt{message}(m, c)$.

There are also clauses that account for the protocol description, which are far less intuitive; we are not going to present them, as this would not add any benefit in terms of giving a more complete intuition of how Horn clauses can be used for our goals, but it would merely be tedious and time-wasting — nevertheless the interested reader can check out the work by Bruno Blanchet, in particular the report "Vérification automatique de protocoles cryptographiques: modèle formel et modèle calculatoire" [Bla08] for a quick overview.

5.2 Theorem Provers and Model Finders

Once we have a consistent (*i.e.* non-contradicting) set of clauses, we can derive new clauses: each new clause is a new theorem of our system.

Viceversa, given a clause we are interested in deciding whether it is a theorem derivable from a given set of clauses.
There are two ways to answer this questions:

- we derive the theorem, and this yields a positive answer;
- we find a counterexample, and this yields a negative answer.

This can be done by hand, although this is not always advisable, as this requires time and it can be error-prone.

We can avail of a variety of tools to overcome this: an example is given by proof assistants, which do not take care of all the work that needs to be done, but they help the people doing it by suggesting the possible proof steps.

In some settings it is possible to apply *automated proving techniques*, so there are also tools that prove theorems (or find counter-examples as well) without requiring any interaction with the user at runtime — one such setting is that of *first-order logic* (also with equations), which is the setting we will be working in.

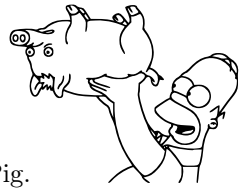
Nevertheless each tool has its points of strength and weakness, depending on the algorithms behind each tool that may suit effectively the intended application or not: we have tried different tools and we found out that William McCune's suite (the theorem prover *Prover9* and the model finder *MACE4*) was able to address appropriately the issues we were dealing with — we will discuss this trial-and-error path that led us to the choice of Prover9/MACE4 in chapter 7.1.

Differently from the case of ProVerif, when we needed a deeper understanding of the way it works, we will skip the description of the theory underlying the Prover9/MACE4 suite.

Chapter 6

Black-box Verification Model

Spider Pig, Spider Pig
Does whatever a Spider Pig does.
Can he swing
from a web?
No, he can't,
he's a pig.
Look out,
he's a Spider Pig.



Homer J. Simpson

In the present work we are proposing two different techniques, that aim at the verification of electronic circuits through the verification of the *m*VHDL code that describes them.

The first approach sees the device under test as a number of interconnected functional blocks, and each block is seen as a black-box, whose behaviour is specified by an input-output relation, rather than by a lower-level electronic description.

We are interested in this kind of approach for a variety of different reasons:

- it is easier to understand as it avoids working on the gory details of lower-level description;
- it needs less computational power to carry out the verification process, as there are much less variables that have to be used;
- whenever it is reasonable (or needed) to go to a lower-level with the analysis, it makes sense to check first that everything works properly at a higher-level (*e.g.* the terminals of all functional blocks are properly interconnected);
- one can build a repository of verified blocks, that need not be verified again at the lower-level
- besides proprietary blocks, an electronic designer can avail of a constantly increasing number of IP blocks, which are usually provided in register transfer language or as netlists, in order to be reasonably protected against reverse-engineering: as a result these blocks have to be modelled as a black-box.

This chapter aims at providing a detailed description of how it is possible to model a device in a way that is suitable to verify security properties, in the case when we are using some functional blocks to account for higher-level operations.

6.1 The Device

We can think of a device as a set of different blocks which are connected by wires: this topology can be rendered by processes representing the different blocks, that can communicate by means of message-passing through channels.

We can distinguish different kinds of channels:

- an *input channel* is a channel that represents a wire connected to an input terminal of the device;
- an *output channel*, conversely, represents a wire connected to an output terminal of the device;
- all other channel are not available to somebody using the device (we are assuming that there is some sort of physical protection that prevents an user from directly accessing to the inside of the device): for this reason we refer to them as *internal channels*.

These channels will be an abstraction of the physical wires present in the device: we assume that they can bear signals of any size. This is a standard assumption in formal verification, and the reason why it is a common practice is that this allows us to disregard all information about unnecessary parameters that add nothing to the functional analysis, and draw attention away from proving the correctness of a design.

We are going to address the problem of proving a design correct from a security perspective, and for this reason we have to distinguish channels that can be accessible by a user from channels that are out of reach: the latter channels are only the internal ones, and therefore they are, in general, private channels; accessible channels are public by definition, generally speaking, and are all input and output channels. These are general guidelines, as we may have a verification scenario where we want to alter these "default settings", and have a private input/output channel (for example if we want to model the case of an input/output terminal that is somehow protected from intrusion) or a public internal channel (it is the case that some internal wire can be reached, so actually we are simply seeing it as an input/output terminal, which is by default represented by a public channel).

As the reader may have noticed, we are recreating a sort of Dolev-Yao model presented in section 3.2 to model the device environment: we keep on moving along this line and postulate that all operations provided by a block, including cryptographic primitives, are perfect and immune from any flaws that can be exploited to cause a security breach.
This is a strong assumption, as probably there exist no such thing as a "flawless device", nevertheless an electronic designer who is using an IP block has no other choice but to trust the specifications of the component. The ultimate meaning of this assumption is that the whole point of this analysis is to make sure that the electronic designer has not assembled (allegedly) flawless components into a flawed design.

Once we have completed the analysis with this assumption and we obtain the formal proof that we have a sound design, we can generalise this approach by allowing probabilistically flawed components, and instead of assuming that all operations provided by a block are perfect, we could assume that they deliver a wrong result with a certain probability and/or we could assume that cryptographic functions

can be broken with given probability: these are all interesting possibilities that would add a quantitative perspective to this analysis, which are beyond the scope of this thesis but that may be subject of future developments of this work.

6.2 The Malicious User

An user can interact with the device through all accessible terminals (which are usually all unprotected input and output terminals, *i.e.* those represented by public channels), and is not able to gather access to any other part of the device (*i.e.* internal channels are not accessible).
It is important to make this point clear, as if the device ends up in the wrong hands, we want to be sure that a malicious user cannot extract confidential data from it: he can access the same terminals as a legitimate user, as we assume that also for him it is not possible to open the device and access to any other part thereof.
A malicious user starts interacting with a device with a certain knowledge pool: his goal is to widen his knowledge in order to compromise the security goals set for the device.

The usual assumption is that a malicious user has an unlimited amount of computational power: in our setting this means that he is able to interact with the device by sending arbitrary signals to input terminals (*i.e.* he can forge any message and send it through an input channel).
Thanks to this amount of computational power, a malicious user is able to process all signals that can be read from the output terminals, in order to increase his knowledge.

We add one final assumption, concerning the number of devices a malicious user can avail of: he has access to many identical copies of the device, so that he can work at once with all of them, and eventually interconnect them together as he wants.

This completes the creation of a setting in the style of the Dolev-Yao model: for this reason we will interpret a *m*VHDL description of a device as a protocol, that describes the interactions among the different blocks of the device.
Once we have given this interpretation and we have translated it into a suitable format, we can use the tool ProVerif to perform the analysis of the device: we will require that confidential data remains secret, so that a malicious user cannot gather access to it by any means.

6.3 From *m*VHDL to the Applied π-calculus

*m*VHDL and the applied π-calculus are two languages that have been conceived with different requirements in mind and, as a result, they differ for a number of features, so it is evident that it is not possible to make a 1:1 translation from *m*VHDL to the applied π-calculus.
Nevertheless this is not a big limitation in view of the goal of verifying security properties on *m*VHDL description, as what we need to capture and focus on is all of the communication events that happen among the different blocks which the device is made of. In particular we intend to be able to track what flows through all channels, with special care devoted to confidential data — this kind of data must be transmitted in clear only on private channels.

As we hinted in the previous §6.1, we can model an electronic device as a set of processes running in parallel, which exchange messages to communicate: the processes we will be using will have a certain degree of nesting, as we have processes (in *m*VHDL sense) that contain subprocesses and entities, thus

the nested structure of the original *m*VHDL code is reflected into the applied π-calculus model.

The rest of this section will present how we are going to render the different *m*VHDL elements in the corresponding applied π-calculus model.

We will present an example of the application of this translation procedure in §A.1.

6.3.1 Processes

In *m*VHDL we can describe a system as a parallel composition of several processes. Each process in turn can be either a sequence of commands or a parallel composition of different subprocesses.

The applied π-calculus provides the process replication operator `!`: if we put this operator before a process, it generates an unbounded number of replications of the process. We can use this operator in front of the complete description of the system under verification, in order to model the malicious user's capability of using multiple copies of the device in parallel (in some sense he is establishing multiple parallel sessions).

Processes are made quite in the same way both in *m*VHDL and in the applied π-calculus, so we would expect that the task of expressing a *m*VHDL process in the applied π-calculus is a straightforward one: this is quite corresponding to reality, although we must pay attention to the fact that it is not possible to single out a monolithic fragment of *m*VHDL code for each process, as in *m*VHDL it often happens that a prototype of a part of the design is given, and followed only later on by the corresponding implementation (this is for example the case of an entity declaration and the corresponding architecture).

Entities

An entity is instantiated through the keyword `entity`: this provides a prototype describing the ports available for interactions with other entities and components of the design, but there is no actual implementation in this declaration.

The corresponding *m*VHDL code is the following:

```
entity entity is
generic (
  [generic]
)
port (
  [ports]
)
end entity;
```

This fragment will be rendered in the applied π-calculus by declaring a process with the same identifier of the entity — we leave the description of this task to the moment when we process the corresponding architecture declaration.
It is worth spending a few words on the keyword `generic`, which is used to introduce parameters of the entity: the appropriate values of each parameter will have to be communicated by the environment, and we have two possible ways of doing this:

- in case the parameter is not crucial for the analysis, it can be substituted by a dummy constant: an

example is given by the case when the parameter is the width of a channel — we have assumed that messages of arbitrary width can flow through every channel, so it is no use having this parameter;

- on the contrary, we can declare a private channel with the same name of the parameter, that will input the appropriate value at the very beginning — this value can eventually be a public one. There is a small technicality hidden in this statement, as we are using a *private* channel to input a (potentially) *public* value: the reason is that the channel being private rules out the possibility for the malicious user to provide a different value, although he is entitled to know the value which has been set for that parameter, and is therefore a public one.

The entity declaration is ended by a list of ports: each port has to be declared as a channel, the context will determine if it is private or public — the general criterion is the one stated above, *i.e.* the device terminals are normally public channels and all others are private channels.

```
[private] free port_entity.
```

Architectures

After saying that there will be a process corresponding to each entity, which also shares the same name as this entity, it is now time to actually consider what should be in the process body.

For every entity at least one possible implementation has to be provided, introduced by the keyword `architecture`.
Here we restrict ourselves to the case when for each entity there is a single possible implementation: we do so for the sake of simplicity, but this does not cause any loss of generality, as we can address the general case of multiple possible implementation by repeating the analysis for each possibility — we must make sure that the analysis we perform holds for all possible architectures.

The *m*VHDL code to declare an architecture is the following:

```
architecture architecture of entity is
  [signals]
  [component declarations]
begin
  [processes and components]
end architecture;
```

Before the architecture body, enclosed between the keywords `begin` and `end`, we can see that there are some other declarations, and therefore we are going to have some corresponding code among the initial declarations in the applied π-calculus code.
This part comprises both signal and component declarations: we will address components later on, for the moment let us just consider signals.
Each signal corresponds to a (potentially private) channel in the applied π-calculus, so we are going to have several declarations with the following form:

```
[private] free signal_entity.
```

In the implementation part of the architecture we have both processes and components, that are composed to perform a task: as a result the code for the process ENTITY is a composition of the processes accounting for this parts, which are:

- a sequence of commands that are to be executed (see §6.3.2): this constitutes a sequential process;
- the processes describing components;
- the processes describing the port map for components;
- the processes (in applied π-calculus sense) describing the processes (in *m*VHDL sense, introduced by the keyword `process`.

Here is the resulting code for the process ENTITY:

```
let entity =
  (
    [commands]
  )
  | [components]
  | [port maps]
  | [processes].
```

We are now going to provide additional details on each item of the above list.

Components and port maps

The difference between a component and an entity is more a hierarchical one, rather than a conceptual one: in fact a component is nothing but an entity which has been defined elsewhere, with its own architecture, and as such it is going to be rendered in the applied π-calculus in a similar way.
For this reason the translation procedure is a recursive one, as each component may be made of other subcomponents, mentioned in its architecture: the recursive procedure does not go any further when it finds a component which is "atomic", in the sense that its architecture holds all of the *m*VHDL code that is needed to describe it.

An implementation is able to use a component if the information about its interface is provided, *i.e.* if it is known what ports are available — here is the component declaration:

```
component component.
port (
  [ports]
)
end component;
```

This declaration has to be completed with the actual instantiation of the component, which is made in the following way:

```
component instance : component
port map (
  port=>signal
)
end component;
```

This instantiates a component and connects its port to other signals and ports, according to the indications of the port map.

In the applied π-calculus code the port map can be written as a process that relays every message to and from connected ports; an alternative is using the same name for connected ports.

Processes

Each process is made of a sequence of commands and has an optional sensitivity list, which defines which signals will cause the process to update the outputs: we can safely ignore this feature, as we treat processes as being reactive to every signal.

Here is the *m*VHDL code that corresponds to a process declaration:

```
process : process ([sensitivity list])
  [variables]
begin
  [commands]
end process process;
```

A process can have local variables, and they are declared before the reserved word `begin`: variable declarations do not add information, as we can infer what variables a process uses directly when translating the commands in the process body.
Thus we can translate directly the process body into a corresponding π-calculus process as follows:

```
let process =
  [inputs]
  [variable assignments and other commands].
```

6.3.2 Commands

We are now going to present how to render *m*VHDL commands in the applied π-calculus.
The commands are:

- variable assignment;
- signal assignment;
- conditional statements;
- loop constructs.

Variable assignments

In *m*VHDL variable assignment is made through the operator `:=`, which takes an expression as a right-hand operand, evaluates it in the current state and assigns the resulting value to the variable provided as left-hand operand.

```
variable := expression;
[process]
```

In the applied π-calculus we can bind the expression to a name, which will later be used by the rest of the process:

```
let variable = expression in
  [process]
```

Signal assignments

Signal assignment is made through the operator `<=`, and is quite similar to variable assignment, as the evaluation of the expression on the right-hand side is assigned to the signal on the left-hand side, which will have a transition.

```
signal <= expression;
[process]
```

In spite of the similarity between signal and variable assignment, the conceptual difference between a signal and a variable causes the translation to be completely different in the two cases: in fact we can see a signal assignment as the operation of outputting a value on a channel with the signal name:

```
out(signal,expression);
[process]
```

Conditional statements

*m*VHDL provides two different constructs to code for conditional statements: `if` statements and `case` statements.
There is no need to provide an *ad hoc* translation for `case` statements, as they are nothing but syntactic sugar for a nested `if` statement.

A *m*VHDL fragment using an `if` statement has the following shape:

```
if condition then
  [if process]
else
  [else process].
```

The applied π-calculus provides an identical construct, so the translation is extremely straight-forward:

```
if condition then
  [if process]
else
  [else process].
```

Iterations

*m*VHDL allows the use of `while` and `for` loops. The first one is a guarded recursion, that executes the loop body if a condition (guard) is true and skips it otherwise; the second one is the repetition of the loop body for a given number of times.
It is clear that a `for` loop can be seen as an instance of guarded recursion, where there is an implicit instruction to increment the value of a counter until it reaches the limit fixed by the guard. It is obvious that `for` loops can be expressed as `while` loops: for this reason there is no need to address the case of `for` loops, but we will provide a way of translating only the more general case of `while` loops.

The *m*VHDL code for a `while` loop is:

```
while expression do
  [process]
[after_loop_process]
```

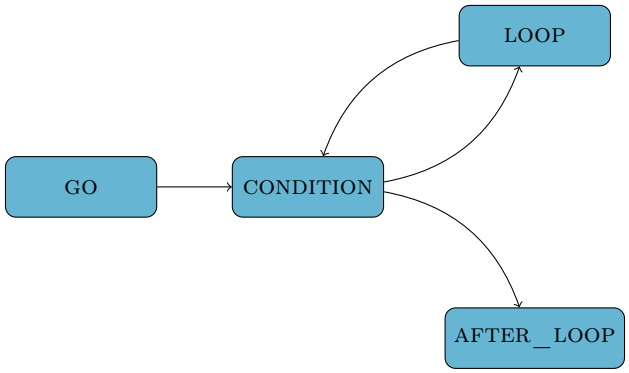


Figure 6.1: The GO process starts the CONDITION process, which decides whether to trigger the LOOP (which will return to CONDITION after its run) or to pass over to what follows the iteration..

The applied π-calculus is recursion-free and this makes it absolutely non-trivial to come up with a way to render loops in the applied π-calculus.
The solution we have decided to use is presented in figure 6.1.
We are going to use a combination of four processes:

- the loop body is contained in the process LOOP, which is infinitely replicated;
- the guard for the process LOOP is checked in the process CONDITION, which is infinitely replicated as well;
- the process GO is the beginning of the loop, and is responsible of starting the whole thing;
- the process AFTER_LOOP accounts for the code to be executed after the loop has finished.

Therefore the corresponding code in the applied π-calculus is:

```
 !(
    in(go,go_value);
    if go_value=TRUE then
    (
      in([terms in expression],[terms in expression]_loc);
      if expression then
        out(loop,TRUE);
    )
  )
|!(
    in(loop,loop_value);
    if loop_value=TRUE then
    (
      [process]
    )
  )
|(
  in(end,end_value);
  if end_value=TRUE then
```

```
    [after_loop_process]
 )
| out(go,TRUE);
```

6.3.3 Variables, signals and constants

We now take a closer look at variables, signals and constants, as well as to operators that are applied to them.
For what concerns variables, we are always dealing with *local* variables:

```
variable variable : type [: value];
```

In the applied π-calculus they are bound names, as their scope is limited to the process they reside in.

We have already discussed how signals are something different from variables:

```
signal signal : type [: value];
```

We have shown that a signal assignment can be rendered as a communication over a channel with the same of the signal: we have implicitly stated that signals are free names names in π-calculus.

*m*VHDL uses some constants, which can be divided into three groups: logical values, truth values and integers.

expression	::=	`'0'` | `'1'` | `'U'`	(Logical values)
	|	`true` | `false`	(Truth values)
	|	$n \in \mathbb{Z}$	(Integers)

It is important to have a clear picture of what constants are used by *m*VHDL, as they have to be declared one by one in the applied π-calculus:

```
data constant/0.
```

This declaration contains information about the arity of a constant, which is 0: the applied π-calculus sees a constant as a function taking no arguments.

When a custom data type is used, we have to add a new constant declaration for each of the possible values.

A set of standard constant declarations is the following:

```
data UNDEFINED/0.
data ZERO/0.
data ONE/0.
data POSEDGE/0.
data NEGEDGE/0.
data TRUE/0.
data FALSE/0.
data N_1/0.
data N_2/0.
...
data N_m/0.
```

Signal transitions

In *m*VHDL we distinguish when a signal has a stable value from the moment when there is a transition from one value to another, distinguishing also if there is a rising or a falling edge.

A possible way to model this is to declare `POSEDGE` and `NEGEDGE` as possible logical values of a signal (it is like having a custom data type): each signal whose dynamic behaviour matters cannot pass directly from `0` to `1` (and viceversa) without passing through `POSEDGE` (or `NEGEDGE`): this situation is depicted in figure 6.2.

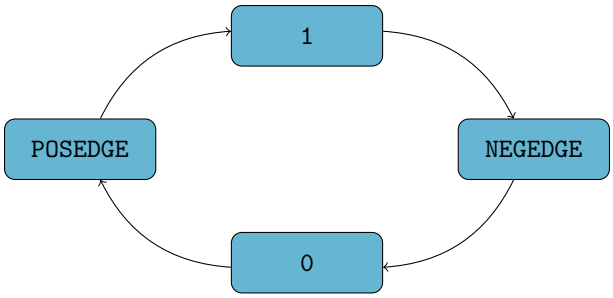


Figure 6.2: The transitions from `0` to `1` (and viceversa) must pass through `POSEDGE` (or `NEGEDGE`).

6.3.4 Operators

If we see an operator as a function *operator*`(`*[arguments]*`)`, we can see the execution of an operation as the application of the corresponding function to the operands, followed by the assignment of the resulting value to a variable:

```
variable := operator([arguments]);
[process]
```

What above obviously holds also for signals — we just have to use `<=` instead of `:=`.

In the applied π-calculus we can declare a function implementing the operation, and use it to build an expression.
In case of a variable assignment we have:

```
let variable = operator([arguments]) in
  [process]
```

If there is a signal assignment instead of a variable assignment, we have to render it as an assignment to a temporary variable followed by the assignment of that variable to a signal, therefore a `let` statement followed by an `out` statement:

```
let temp_var = operator([arguments]) in
  out(signal,temp_var);
  [process]
```

*m*VHDL provides these unary and binary operators:

op ::= `not` | `and` | `or` | `lnot` | `land` | `lxor` | `+` | `-` | `<` | `=` (Operators)

For each one of these operators we need to have a corresponding function declaration in the applied π-calculus:

```
fun unary_operator_name/1.
fun binary_operator_name/2.
```

Again, we need to state explicitly the arity of each function.

A set of standard function declarations is the following:

```
fun lnot/1.
fun land/2.
fun lor/2.
fun xor/2.
fun plus/2.
fun minus/2.
fun lessthan/2.
fun equals/2.
fun concat/2.
```

After these initial declarations we can add equational theories if appropriate.

6.3.5 Clock

We add an extra process, which is not explicitly accounted for in the *m*VHDL code, but which is an implicit presence in case of synchronous systems: the clock.
One possible process CLOCK, in case there is only one top entity that has to synchronize with it, is the following:

```
let clock =
  in(clk_clock,prev);
  if prev=ZERO then out(clk_top,POSEDGE)
  else if prev=POSEDGE then out(clk_top,ONE)
  else if prev=ONE then out(clk_top,NEGEDGE)
  else if prev=NEGEDGE then out(clk_top,ZERO).
```

This process inputs a value containing the previous clock value and binds it to `prev`: depending on the received value, it decides which value should be output and then terminates — for this reason it has to be infinitely replicated via the `!` operator.

In the case of more processes needing to synchronize with this clock, we can add an input statement for each extra process: it will have to collects a token from each concurrent process, before allowing them to proceed.
Obviously doing so requires that also the output part of the process has to be modified accordingly.

6.3.6 The translation procedure

We now describe a procedure to render a piece of *m*VHDL code in the applied π-calculus: this procedure has to be implementable in a program, so that it requires (almost) no human intervention.

We require the file to be annotated, in the sense of having some *ad hoc* comments with the purpose to give directives to the translation procedure — the will usually look like the following:

```
--<directive>
```

The *m*VHDL code has to be parsed hierarchically, and for this reason we need to identify the top entity. It is possible to infer this information by analysing all of it and building a dependency tree, nevertheless it would speed things up if we could avoid this: for this reason it would be wise to annotate the code with `--<top>` just immediately before the `entity` declaration — the translation procedure could scan through the file looking for this directive, and fall back to the hierarchical analysis only if the top entity is not introduced in this way.

A device interface to the outside world is made of all the ports of the top entity, as they are the terminals available to the user for interaction with the device.
All ports of the top entity are therefore declared as public channels by default; we allow the use of the directive `--<private>` (to be used immediately before the port name it should refer to) to override the default setting.
All other signals are to be declared as private channels — the directive `--<public>` should be used to override this default setting.

Throughout the translation procedure, when we have to declare a channel name we do want to make sure that all names are unique: we comply with this requirement by making up a channel name starting from the name of the signal and the name of the process it belongs to — we concatenate them and add `_` for readability:

```
[private] free signal_process.
```

The body of the top entity is then parsed and all commands therein are translated according to the specifications given in §6.3.2.

Once this has been done, an analogous procedure is repeated for all dependencies of the top entity, and repeated again recursively until all of the necessary parts have been parsed and translated: we are going to obtain a structure where higher level processes instantiate lower level ones.
There has to be a slight difference, though: while all ports had to be considered public by default in the top entity, here everything is private by default — in fact there is no direct external access to a dependent component (even in the case of a direct connection to a public top entity port we can see this as a private channel being used for communication with the top entity, which relays the data received on a public channel).

Each applied π-calculus process has to begin with a sequence of `in` statement, whose purpose is to do a sort of initialization for the process: we have to read the content of each channel needed by the process and make this value available by binding it to a local variable *channel*`_loc`.
We see that this is the dual procedure of signal and non-local variable assignments, which are to be instantiated by means of `out` statements.

In some settings it may be required/desirable to be able to provide manually some code in the applied π-calculus to account for a fragment of *m*VHDL code: we can do so by adding the custom code to the original *m*VHDL code as comments with an appropriate format:

```
--<manual>
```

```
--<picalculus>
--  [pi calculus code]
--</picalculus>
  [manually translated mVHDL  code]
--</manual>
```

In this way it is possible to implement a translation procedure that will ignore whatever *m*VHDL code is contained between `--<manual>` and `--</manual>` and output the code contained between `--<picalculus>` and `--</picalculus>`.

We have to provide an accessory process INIT, which is responsible for the initialization of variables and ports: it will be a sequence of `out` statements, and the process terminates after that.

At this point we have all of the pieces we need to express the *m*VHDL code under analysis in the applied π calculus: we simply have to put the pieces together by instantiating the parallel composition of process INIT with the replication of process TOP (the replication operator `!` accounts for the malicious user availing of many copies of the device) and the replication of process CLOCK:

```
process init | !top | !clock
```

Chapter 7

Looking into the Black Box

> Outside of a dog, a book is man's best friend. Inside of a dog, it's too dark to read.
>
> *Groucho Marx*

In the previous chapter we have presented a possible verification approach, which analyses a device from a rather high level: in the present chapter we are going to analyse things in deeper detail in order to be able to verify a detailed model of a device, when available.

This approach can be used to build a library of verified components, which can be later combined with IP blocks into a verifiable design, where the components need not be verified for what concerns their inner functioning, but only the connections and the extra code have to — and this can be made either with the techniques presented in the previous chapter or with those presented in this one.

The same considerations of §6.1 and §6.2 apply, so we are going to jump to the description of the verification technique without any further preamble.

7.1 From *m*VHDL to MACE4

We are going to express a fragment of *m*VHDL as a set of Horn clauses (see §5.1.1), and then we are going to use a model finder to analyse them: the model finder is delivering a model (*i.e.* a set of values for the set of variables used in the logic clauses), and for this result we present an interpretation showing which channels are to be private and which other channels can be public, in order to meet the security goal we set.
The methodology presented here has been tailored on the model finder MACE4, but it can work with any model finder simply by adjusting the syntax by which logic clauses are expressed.
The translation from *m*VHDL into Horn clauses can be divided into five parts:

- preamble (equational theories, function declarations, malicious-user capabilities);
- initialization;
- internal description (transition rules);
- goals;

- hints and requirements.

We will present an example of the application of this translation procedure in §A.2.

7.1.1 Why MACE4

The decision of using MACE4 to pursue our goal was not a painless one, as MACE4 was only one among several possible alternatives — but unluckily it was not the first one that has been tested.

The first step was to generalise the approach presented in the previous chapter and use a cryptographic protocol verifier: the choice was between ProVerif and AVISPA.
In different tests we did not manage to overcome the problems posed by the need of using equational theories (for the `xor` function, see §7.1.3), as ProVerif simply takes too long[1] to perform this analysis with equational theories, whereas AVISPA did not meet our needs — this tools has four different back-ends, and those supporting equational theories do not support custom data types, and viceversa.

Moving on from protocol verifiers, it was the time of testing theorem provers: they are powerful tools, which have the great advantage of being general-purpose, and we were considering three options, namely SPASS, Prover9 and H1.
Tests have been carried out with SPASS and Prover9, but both took too long; we did not expect the result of running the analysis with H1 to be different, so we moved further on.

Finally it was the time of tackling the problem from a different perspective, the one with model finders: they say "third time lucky", and MACE4 turned out to be a suitable approach to the problem, yielding the results we were looking for, in the shape of a type system for the device channels and data.
Without further ado, we are now going to describe this approach.

7.1.2 Predicates

We are using logic to reason about the system under verification, and therefore we need to express its properties by means of logic clauses.

When looking at electronic devices from a security perspective, we are interested in being able to express the malicious user's knowledge and to describe the data flow through the different channels; the first matter will be addressed by the predicate `mknows`, whereas the latter is addressed by the predicate `value`; these are the building blocks that will allow us to describe the behaviour of a system: we do so by combining them with the appropriate logic connectives into the logic clauses mentioned above, accounting for the system properties.

We are now going to discuss the predicates `mknows` and `value` in finer detail, introducing also the relevant inference rules.

The predicate `mknows`

The predicate `mknows(x)` has arity 1 and states that the malicious user knows the parameter `x`.
The `mknows` predicate can be used in several ways:

- we state the initial knowledge of the malicious user by having a predicate `mknows(x)` for each term `x` that is publicly known — it shall be noted that no premise is required to infer the predicate

[1] And by "too long" we mean that we killed the process after three days without getting any result.

`mknows(x)`:

$$\frac{}{\texttt{mknows(x)}}\ \text{(initial knowledge)}$$

Example: `mknows(0).  mknows(1).`

- we explicitly allow the malicious user to use a function φ, and this is done by allowing the malicious user to infer the result of the applying a function to terms in his knowledge:

$$\frac{\bigwedge \texttt{mknows}(x_i)}{\texttt{mknows}(\varphi(x_1, x_2, \ldots, x_n))}\ \text{(function } \varphi\text{)}$$

Example: `mknows(x) & mknows(y) -> mknows(xor(x,y)).`

- in general the malicious user acquires knowledge as the system runs, when some conditions are verified: each time this happens, the malicious user gets to know a new term x and we have a corresponding inference rule with the following shape:

$$\frac{\bigwedge \texttt{cond}}{\texttt{mknows(x)}}\ \text{(possibility to infer x)}$$

The predicate `value`

The predicate `value(Cxxx,x)` has arity 2 and states that the value on channel `Cxxx` is of the same type as x. Although it is not equivalent, it is helpful to think that the predicate asserts that the value x itself is on channel `Cxxx`: this helps to understand a system description in most cases, but it may be misleading in some settings — for example when we are testing for inequality, this will become clear in §7.1.4.
The `value` predicate is used in the following situations:

- whenever we intend to declare the initial values on channels, we are going to have a predicate $\texttt{value(Cxxx,}x_0\texttt{)}$ for each channel `Cxxx` which we wish to initialize to the value x_0:

$$\frac{}{\texttt{value(Cxxx,}x_0\texttt{)}}\ \text{(init)}$$

As above, it shall be noted that in this inference rule we have the void hypothesis as antecedent.

- we have inference rules to describe the transitions that can be triggered. We describe patterns as appropriate combinations of `value` predicates, and this can be used to describe the initial and final state (intended as a particular combination of the values of each relevant signal, variable and port) of a transition:

$$\frac{\bigwedge \texttt{value(Cxxx}_i\texttt{,}x_j\texttt{)}}{\bigwedge \texttt{value(Cxxx}_k\texttt{,}x_l\texttt{)}}\ \text{(transition)}$$

- when used in combination with the predicate `mknows`, we can express the malicious user's capability to read from a channel `Cxxx` (and we are therefore declaring this channel as a public one):

$$\frac{\texttt{value(Cxxx,x)}}{\texttt{mknows(x)}}\ \text{(read on channel Cxxx)}$$

Example: `value(Output,x) -> mknows(x)`

- the dual rule is the one expressing the malicious user's capability to write to a channel `Cxxx` (also a public one):

$$\frac{\texttt{mknows(x)}}{\texttt{value(Cxxx,x)}}\ (\text{write on channel } \texttt{Cxxx})$$

  Example: `mknows(x) -> value(Input,x)`

- we can use this predicate to constrain two channels to have the same value, and this in effect accounts for a port map:

$$\frac{\texttt{value(Cxxx}_1\texttt{,x)}}{\texttt{value(Cxxx}_2\texttt{,x)}}\ (\texttt{xxx}_1 \mapsto \texttt{xxx}_2)$$

$$\frac{\texttt{value(Cxxx}_2\texttt{,x)}}{\texttt{value(Cxxx}_1\texttt{,x)}}\ (\texttt{xxx}_2 \mapsto \texttt{xxx}_1)$$

7.1.3 Functions

Functions are used to manipulate terms: we are going to use only term functions on terms that perform internal operations, *i.e.* whose result is itself a term.
We are going to need only the following functions:

- exclusive or (`xor`);
- a generic invertible function (`invertible`);
- concatenate (`concat`).

An equational theory is needed for the `xor` function; for the `invertible` and `concat` functions we just need to give an invertibility rule.

Equational theory of the `xor` function

The `xor` function has had several applications in cryptography, and security of several algorithms rely on its properties. This is why we need to be able to take such properties into account and have an appropriate equational theory to reason about it, as pattern matching is inadequate[2].

The properties of the `xor` function are:

$$x \oplus y = y \oplus x$$
$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$
$$(x \oplus x) \oplus y = y$$

[2] In fact terms that match according to an equational theory fail to match if we are restricting ourselves to pattern matching.

The `invertible` function

We aim at building a type system to determine whether an electronic devices complies with requested security features: from this perspective it is clear that we do not need to address in detail how every function works. This is in fact the case of all those functions that are "invertible", in the sense that from their result it is possible to infer some knowledge concerning their argument — this is therefore a wider class, compared to that of functions that are mathematically invertible: if the malicious user knows the result of applying an "invertible" function to a term which is supposed to be secret, in our analysis we have to consider this term as compromised, as confidential information has leaked.

The following inference rules apply:

- if the malicious user knows the term $\mathtt{x}$, he can infer the term $\Psi\mathtt{x}$:

$$\frac{\mathtt{mknows(x)}}{\mathtt{mknows}(\Psi\mathtt{x})}\ \text{(application of } \Psi\text{)}$$

- if the malicious user knows the term $\Psi\mathtt{x}$, than we consider the term $\mathtt{x}$ as compromised, *i.e.* we assume he knows $\mathtt{x}$:

$$\frac{\mathtt{mknows}(\Psi\mathtt{x})}{\mathtt{mknows(x)}}\ \text{(application of } \Psi^{-1}\text{)}$$

The `concat` function

The purpose of this function is to merge two terms into a single term, without really adding anything: this function is needed whenever we have disjoint pieces of data and we want to apply a function to them.

The following inference rules apply:

- if the malicious user knows the terms $\mathtt{x}$ and $\mathtt{y}$, then he can infer their concatenation:

$$\frac{\mathtt{mknows(x)} \qquad \mathtt{mknows(y)}}{\mathtt{mknows(x|y)}}\ \text{(concatenation)}$$

- viceversa, if the malicious user knows the concatenation of two terms, he is able to extract the individual terms from it.

$$\frac{\mathtt{mknows(x|y)}}{\mathtt{mknows(x)} \wedge \mathtt{mknows(y)}}\ \text{(decomposition)}$$

It is immediate to see how we can render invertible functions of any arity as an appropriate combination of the functions `invertible` and `concat`.

7.1.4 A semantics for *m*VHDL

A necessary premise to the task of translating *m*VHDL into logic clauses is the formalization of its semantics, as this provides a complete and unambiguous specification of the language. In particular we need to define a translation function that maps each *m*VHDL fragment to a statement understandable by MACE4.
Within a statement in *m*VHDL we can recognize different parts, which have to be treated differently in the translation process.

Equality and inequality

Comparison[3] is one of the most common operations: it returns a boolean value *True* or *False* depending on the result obtained after testing for equality (or inequality) the content of a variable (or the value of a signal) against an expression[4] in an environment ρ.
We use $\mathcal{T}[\![c]\!]\rho$ ($\mathcal{T}$ as in *test*) to denote the translation of testing condition c in the environment ρ; when the condition being tested is equality or inequality, this specialises to:

$$\mathcal{T}[\![x = e]\!]\rho \triangleq \texttt{value}(c_x, \mathcal{E}[\![e]\!]\rho)$$
$$\mathcal{T}[\![x \neq e]\!]\rho \triangleq \mathcal{T}[\![x = e]\!]\rho$$

This translation may look a little surprising, because testing a condition c or its opposite $\neg c$ translates to the same predicate. Nevertheless we have to keep in mind that the `value` predicate focuses on the data type, rather than on the actual value itself: this yields that the translation function must return the same translation in both cases, as the data type is the same whether we are testing for equality or inequality, the translation function must return the same translation.
Another implication of this feature of the translation function is that in an if-statement both of the *if-branch* and the *else-branch* will be explored — and this is exactly what we are looking for, as verification has to explore both branches, so that all potential flaws can be uncovered.

Assertion

An assertion states that a channel contains a certain value, and we use $\mathcal{A}[\![a]\!]\rho$ ($\mathcal{A}$ as in *assert*) to denote the translation of an assertion a in the environment ρ:

$$\mathcal{A}[\![x = e]\!]\rho \triangleq \texttt{value}(c_x, \mathcal{E}[\![e]\!]\rho)$$

We can see that there is no apparent different in the translation of an assertion and that of a test: once again the reason is that we are only interested in the data type.
Moreover as assertion and test are two different concepts, it would make no sense to merge these notions (this would be quite misleading instead): this will become evident later on, when we start giving semantics to the different constructs.

Semantics

Now that we have defined the translation functions for tests and assertion, we are ready to move on and actually start to give semantics to complete *m*VHDL statements.

For a *m*VHDL statement s we note its translation as $\mathcal{S}[\![s]\!]\rho\gamma$ ($\mathcal{S}$ as in *semantics*): this translation function returns an implication, where all tests are in the premise and lead to an assertion[5].

We can notice that in $\mathcal{S}[\![s]\!]\rho\gamma$ we have appended an additional term γ after the environment ρ: this accounts for the guards that have to be true for a statement to be executed (γ can therefore be seen as

[3] We restrict ourselves to the case when a variable (or a signal) is compared against an expression, as it is more straightforward to implement — and clearly this is no limitation, as we can still compare two expressions by adding an assignment of the value of either expression to a temporary variable.

[4] If we want to be pedantic, instead of talking of an expression e *tout court* we should rather talk about its *evaluation* $\mathcal{E}[\![e]\!]\rho$ of e in the environment ρ.

[5] We have that $A \Rightarrow B \equiv A \Rightarrow (A \wedge B)$: when writing the translations of the different statement we will implicitly apply this rule, as we want to emphasize that what was true before the statement s has to be true also afterwards, *i.e.* types of channels cannot change.

nothing but a conjunction of tests).

Variable and signal assignment When assigning an expression e to a variable x we are taking the contents of all variables mentioned in e (its free variables, which make the set $fv(e)$) and use these values to evaluate e.
In some sense what we need to do in order to do this is to test that the variable has been initialized, *i.e.* that there is a corresponding `value` predicate being true, and this is the reason why the (implicit) premise of a variable assignment is given by a conjunction of tests on all variables in $fv(e)$; provided that this premise holds, we can then assert that the new value of x is the evaluation of e:

$$\mathcal{S}[\![x := e]\!]\rho\gamma \triangleq \gamma \wedge \bigwedge_{y \in fv(e)} \mathcal{T}[\![c_y = e_y]\!]\rho \Rightarrow \gamma \wedge \bigwedge_{y \in fv(e)} \mathcal{T}[\![c_y = e_y]\!]\rho \wedge \mathcal{A}[\![x = e]\!]\rho$$

We can apply the translation functions $\mathcal{T}$ and $\mathcal{A}$ to see the translation in terms of `value` predicates:

$$\mathcal{S}[\![x := e]\!]\rho\gamma = \gamma \wedge \bigwedge_{y \in fv(e)} \texttt{value}(c_y, \mathcal{E}[\![e_y]\!]\rho) \Rightarrow \gamma \wedge \bigwedge_{y \in fv(e)} \texttt{value}(c_y, \mathcal{E}[\![e_y]\!]\rho) \wedge \texttt{value}(c_x, \mathcal{E}[\![e]\!]\rho)$$

Once this definition has been given, things are easy when it comes to dealing with signal assignment, as we are treating signals and variables as the same thing, and therefore we want also signal assignment to have the same translation as variable assignment:

$$\mathcal{S}[\![x <= e]\!]\rho\gamma \triangleq \mathcal{S}[\![x := e]\!]\rho\gamma$$

Sequential composition In sequential composition we have a sequence of statement, that are to be executed one after the other. From a security perspective the fact that the statements are executed sequentially is not particularly relevant, as we are only interested in the data types involved (we can think of it as being simply interested in the fact that a statement does not cause any sensitive information to leak, no matter when it is executed).
Therefore sequential composition of different statements translates to the union of the clauses that translate each statement:

$$\mathcal{S}[\![s_1 \; ; \; s_2]\!]\rho\gamma \triangleq \mathcal{S}[\![s_1]\!]\rho\gamma \cup \mathcal{S}[\![s_2]\!]\rho\gamma$$

Parallel composition Similar considerations apply to the case of parallel composition, with the additional constraint that two concurrent statement must not interfere and therefore we require the sets of variables, on which they operate, to be disjoint:

$$\mathcal{S}[\![s_1 \mid s_2]\!]\rho\gamma \triangleq \mathcal{S}[\![s_1]\!]\rho\gamma \cup \mathcal{S}[\![s_2]\!]\rho\gamma \text{ where } fv(s_1) \cap fv(s_2) = \varnothing$$

Usually the side condition on the disjointness of $fv(s_1)$ and $fv(s_2)$ is verified as we are normally dealing with physical distinct modules (if this does not hold, there is a problem with parallel composition itself, so probably there's a design problem which we should worry about well before worrying about security). To avoid unintentional name capture we will always use alpha-renaming[6], so we need not worry about the same variable identifier being used in different modules.

[6] Alpha-renaming is a procedure that replaces any bound variable identifier in the program with an univocal identifier that is not used elsewhere to refer to different variables. This makes name resolution absolutely trivial, as there are no scoping rules involved.

Conditional construct The conditional construct `if-then-else` has the following semantics:

$$\mathcal{S}[\![\texttt{if}\ q\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\rho\gamma \triangleq \mathcal{S}[\![s_1]\!]\rho(\gamma \wedge \mathcal{T}[\![q]\!]\rho) \cup \mathcal{S}[\![s_2]\!]\rho(\gamma \wedge \mathcal{T}[\![\neg q]\!]\rho)$$

We can see that we are treating the condition q as a guard for the execution of the statement s_1, whereas $\neg q$ acts as the guard for the execution of the statement s_2.

We stress once again that as a consequence of $\mathcal{T}[\![q]\!]\rho$ having the same translation as $\mathcal{T}[\![\neg q]\!]\rho$, we are exploring both branches of the conditional construct.

Iteration The formal definition of a loop involves fixpoint theory, as can be seen from the following definition of the loop construct `while-do`:

$$\mathcal{S}[\![\texttt{while}\ q\ \texttt{do}\ s]\!]\rho\gamma \triangleq \mu x \bullet \mathcal{S}[\![s\ ;\ x]\!]\rho(\gamma \wedge \mathcal{T}[\![q]\!]\rho)$$

As for the conditional construct, q acts as a guard.
Handling this construct from a software point of view requires a fixpoint engine [Hym04], that will take care of unfolding the loop.
Implementing such an engine is necessary if we are looking for complete tool support (which is well beyond the scope of this thesis), but this is not required to reason at a higher level; in the example addressed in the present work we will manually unfold loops when necessary.

Translation assistant

The whole task of verifying the code from a *m*VHDL design can be automatised: MACE4 is able to perform the analysis and therefore we need not worry about this part, what still needs to be automated is the actual translation from *m*VHDL into a format understandable by MACE4.

We have developed a demonstrative tool to take care of this task, which goes under the name of TA:CON (Translation Assistant: Clauses over Nets).
It is a small utility, written entirely in C++, which helps translating the behaviour of a system, starting from its *m*VHDL description — this is the most delicate part of the translation from *m*VHDL into logic clauses, as this is the most error-prone task.
The program parses an input file containing *m*VHDL code and creates a table with all of the possible internal transitions: the file is parsed word by word, and the program works on a stack where it keeps track of the context where each word is being read.
The transition table ends up containing a scheme with all possible behaviours of the system, so that we will be able to generate logic clauses starting from this. In this preliminary version of TA:CON we did not implement any kind of expression parser, so each clause will need to be further refined at a later stage: each formula has to be expressed in terms of free variables — to be added in the clause — combined by the functions `invertible`, `xor` and `concat`.
The transition table is logically organised in couples of rows: the first row constitutes the premise in each clause, whereas the second one is the consequence; the organisation of data in each row is also arranged in couples of cells, the first one containing a channel name and the latter containing the correspondent value, so for each couple of cells we can write a `value` statement.

The translation has to be finalized with a simple *copy & paste* to merge together the different files (eventually with some word substitutions to link instances of a lower-level entity — instanced by a higher-level

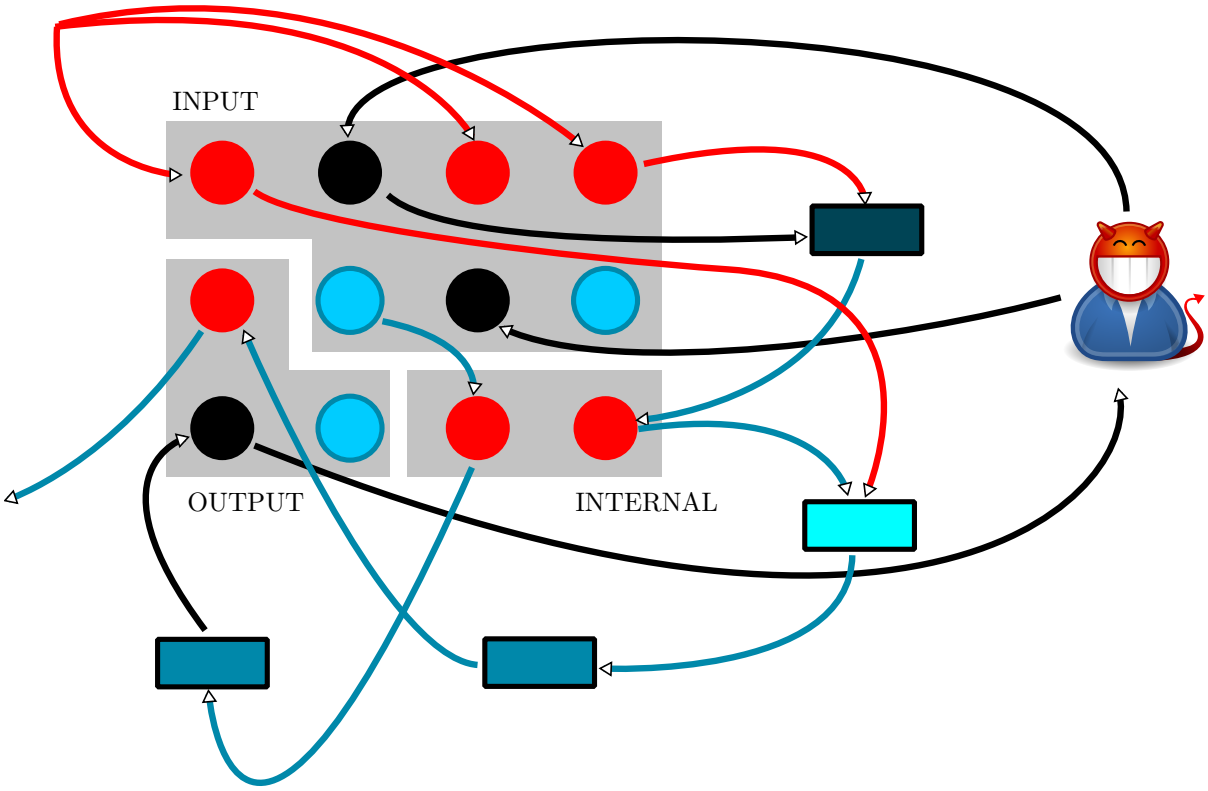


Figure 7.1: Channel and data types, where circles represent channels and the arrows represent the data flow — black channels and data are public, red ones are private, all others to be determined.

one — to their VHDL descriptions) and we finally have the complete translation of the system behaviour.

A few words on other current limitations: we must keep in mind that version 0.1 of TA:CoN is a demonstrative tool, whose goal is merely showing the feasibility of an automated translation from *m*VHDL to MACE4 input files. As such it is still rather limited, as it does not support the following constructs:

- loop unfolding: as we have already stated, this must be done manually;
- nested `if-then-else`: nesting is allowed only if there is no `else` instruction;
- inequality operators (<,>).

7.1.5 The type system

We can distinguish two different types of data within a circuit: the first type comprises all public[7] data, the second all private data.

Likewise we can also distinguish channels into two types: a public channel, which is readable by the malicious user, and a private channel, which does not leak any data to the malicious user.

Security of a device is compromised if the malicious user manages to gather access to confidential data, so it is clear that a secure design must prevent all confidential data from flowing into public channels.

[7] Public in the sense that it can be *made* public without compromising security of the device.

The analysis performed through MACE4 returns a 2-element model that can be interpreted as a type system[8]: we can use this information to understand whether the circuit leaks some confidential data, as well as to determine which part of the circuit require to be protected from external unauthorized access.

When we run the analysis in fact we set a security goal, which usually consists in requiring that a malicious user cannot get to know a piece of confidential data. For example if we require that the system preserves secrecy of the term `x`, we do so by adding the following clause:

```
¬ mknows(x)
```

We have the option of adding some extra clauses to the system description, in order to give some "hints" to MACE4: for some of the channels we know from the beginning whether they are private or public, so we can add some clauses to account for this:

```
value(Cxxx,0).    value(Cyyy,1).
```

which has the meaning that `Cxxx` is a public channel, whereas `Cyyy` is a private one.
This considerably speeds up the analysis in some settings.

We can use this option also to set some requirements as well, for example to constrain a given channel to be public — the analysis is going to tell us if these requirements are compatible with the security goals we have set.

Function tables

The analysis delivers also tables that describe what happens when we apply a function to data of the different types. In effect their application combines data having (possibly) different types into output data, whose type depends on that of the input data.

We can distinguish between *privacy-preserving functions* (*i.e.* `invertible` and `concat`) and *privacy-altering functions* (*i.e.* `xor`).

In the case of 1-ary privacy-preserving functions, the application of a function to an argument returns data of the same type, *i.e.* if data was public it remains public, if it was private it has to remain private. An example is the `invertible` function: having the result of this function reveals information on the data it was applied to, and therefore it preserves the type.

In the case of n-ary privacy-preserving functions, the result of its application is private data if at least one of its arguments was private, public otherwise.
The `concat` function is a 2-ary privacy-preserving function: if at least one of its arguments is private the output is private as well, viceversa the output remains public when both arguments are public.

Privacy-preserving functions are therefore all those functions which are "invertible" in the sense presented earlier, *i.e.* that knowing the result is equivalent to knowing the arguments passed to the function or, at least, it gives some information about them;

Privacy-altering functions return data of a different type than that of the arguments: for example when all arguments are private data, the output is public data.
The `xor` function is privacy-altering: when at least one argument is public the type of the output is the

[8] The element `0` refers to public channels and data, and the element `1` refers to private ones.

one of the second argument, when both arguments are private the output is public.

Predicate tables

The analysis also provides predicate tables, which define the truth value of a predicate, depending on the type(s) of its argument(s):

- the `mknows` predicate must return false whenever the malicious user gets to know a secret value: if such a thing occurs there can be no model for the system. Viceversa, the `mknows` predicate returns true for a public value, as anyone can get to know values of that type;
- the `value` predicate must return false whenever secret data flows on public channels, which implies that the malicious user can get to know that secret. We can see that when a `value` predicate is false for a certain piece of secret data, coherently a `mknows` predicate that has the same piece of data as argument is false. The `value` predicate returns true when data flows in appropriate channels, *i.e.* only public data on public channels and any type of data on private channels.

Chapter 8

Conclusion

In the end, everything is a gag.

Charlie Chaplin

The present work can be placed in the intersection between the domain of electronics and that of computer science: hardware programming in fact belongs to this area, where the boundary between electronics and computer science gets blurred.

Hardware programming is being used on a large scale to implement a variety of devices, whose complexity is constantly increasing: this sets non-trivial verification and testing challenges, as these techniques must be able to address a vary complex reality.

Our aim has been a security-oriented verification of the VHDL design of a device. The motivation for this research it that hardware implementation of computationally expensive algorithms (such as those used in cryptography), as well as development of hardware keys and all kind of security devices in general, requires new verification techniques to ensure that a given design complies with the security goals set in its specification. In particular we have aimed at formal verification, as this approach is much more desirable than testing, as static analysis of code delivers more accurate results.

Having adequate tool support is a key element for formal verification to be efficient, so we have tailored our approach in order to avail of existing tools, namely ProVerif and MACE4. We have also developed a demonstrative tool to show the feasibility of automating the part of the verification process that was not covered by existing tools.

Within a circuit we can categorize data and channels into two classes, distinguishing what can be made available to a malicious user and what must not leak: we propose two methodologies that allow us to prove that no confidential data can possibly be disclosed to a malicious user, regardless of the way the verified device is being used.

The first methodology recreates a scenario very similar to the Dolev-Yao model used in protocol verification: the reason for doing so is to abstract an electronic device and see it as a cryptographic protocol, so that we can use a cryptographic protocol verifier for the analysis of the device compliance with respect to a given security goal.
We have used ProVerif and this has required us to develop a procedure to model the device as an applied π-calculus process

The second methodology returns a type system indicating which data and channels are required to be private in order to achieve a given security goal.
The tool we used for this second methodology is the model finder MACE4 and this has required us to develop a procedure to derive logic clauses from the VHDL code: these clauses describe the system behaviour in terms of type of data and channels.

Future work will move in the direction of the development of an integrated tool, implementing both methodologies along with the translation procedures necessary to parse directly VHDL code and deliver directly the result of the analysis without further human intervention.

Appendix A

Examples

A.1 A first example: a synchronous LFSR scrambler

We take the case of a synchronous LFSR scrambler[1] to test the methodology described in §6.
Our aim is to prove that the malicious user cannot get to know any of the confidential data processed by the circuit, *i.e.* the input data (provided that this channel is protected) and the key.

A.1.1 The VHDL Code

```
------------------------------------------------------------------
FILE: top.vhdl
------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;

LIBRARY lfsr_lib;
USE lfsr_lib.ALL;

---------------------
--
-- The entity top is the whole system
--
-- PORTS (IN):
-- rst: asynchronous reset *B*
-- clk: clock *B*
-- IV: IV *B*
-- go: enables ciphering *B*
-- din: data in *R*
-- key: key *R*
--
-- PORTS (OUT):
-- dout: data out *B*
--
---------------------

--<top>
entity top is
  port (
     rst  : in std_logic;
     clk  : in std_logic;
```

[1] Code kindly written by Nicolas Guillermin.

```
     IV   : in std_logic_vector(3 downto 0);
     go   : in std_logic;
     --<private>
        din  : in  std_logic;
     dout : out std_logic;
     --<private>
        key  : in  std_logic_vector(3 downto 0));
end top;

---------------------
--
-- Architecture arch of top
--
-- SIGNALS:
-- go_cipher: resets the cipherbloc
-- mode: puts cipher.dout on cipher.din
-- FSM: status of finite state machine
--
-- INPUTS: rst,clk,IV,din,key,go_cipher,mode
-- OUTPUTS: dout
--
---------------------

architecture arch of top is
  signal go_cipher : std_logic;
  signal mode : std_logic;

  TYPE enc_state IS (idle,reset_cipher_bloc,cipher); -- custom data type
  signal FSM : enc_state := idle;

  component cipherbloc -- declaration of component cipherbloc
    generic (
      LFSR_depth : integer);
    port (
      rst : in std_logic;
      clk : in std_logic;
      start : in std_logic_vector(3 downto 0);
      dout : out std_logic;
      key : in  std_logic_vector(3 downto 0));
  end component;

begin -- arch

  fsm_proc: process (clk,rst)
  begin  -- process fsm_proc:  sequential process to control the main sequencer
    if rst = '0' then -- asynchronous reset (active low)
      FSM <= idle;
    elsif clk'event and clk = '1' then -- rising clock edge
      case (FSM) is
        when idle =>
           if go='1' then
             FSM<= reset_cipher_bloc;
           end if;
        when reset_cipher_bloc=>
             FSM<= cipher;
        when others =>
```

```
          if go='0' then
            FSM<= idle;
          end if ;
      end case;
    end if;
  end process fsm_proc;

  go_cipher<= '0' when (rst='0') else '0' when FSM= reset_cipher_bloc else '1';
  dout<= go_cipher and (mode xor din);

  cb : cipherbloc generic map (
    LFSR_depth => 4)
    port map (
    dout  => mode,
    clk   => clk,
    rst   => go_cipher,
    start => IV,
    key   => key);

end arch;

-----------------------------------------------------------------
FILE: cipherbloc.vhdl
-----------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;

LIBRARY lfsr_lib;
USE lfsr_lib.ALL;

---------------------
--
-- The entity cipherbloc computes a LFSR
--
-- LFSR_depth: size of the LFSR
--
-- PORTS (IN):
-- rst: asynchronous reset *B*
-- clk: clock *B*
-- start: IV *B*
-- key: key *R*
--
-- PORTS (OUT):
-- dout: data out *B*
--
---------------------

entity cipherbloc is
  generic (
    LFSR_depth : integer := 4);
  port (
    rst  : in std_logic;
    clk : in std_logic;
    start : in std_logic_vector(LFSR_depth-1 downto 0);
```

```
    dout : out std_logic;
    key  : in  std_logic_vector(LFSR_depth-1 downto 0));
end cipherbloc;

---------------------
--
-- Architecture arch of cipherbloc is a sequential process used to cipher
--
-- SIGNALS:
-- data: LFSR register
-- res
--
-- INPUTS: clk, rst, data
-- OUTPUTS: data
--
---------------------

architecture arch of cipherbloc is
  signal data : std_logic_vector (LFSR_depth-1 downto 0);
  signal res : std_logic;
begin -- architecture arch

  main: process (clk, rst)
  begin  -- process main: caculates next value of data
    if rst = '0' then -- asynchronous reset (active low)
      data <= start;
    elsif clk'event and clk = '1' then -- rising clock edge
      data<=data(LFSR_depth-2 downto 0)&res;
    end if;
  end process main;

  calc: process (data)
    variable i : integer;
    variable t : std_logic;
  begin  -- process calc: combinational process to compute tge result of LFSR

  --<manual>
  --<picalculus>
  --  in(key_cipherbloc,key_calc_in);
  --  let t=custom(key_calc_in,ZERO) in
  --</picalculus>
    t:='0';
    for i in 0 to LFSR_depth-1 loop
      t:=data(i) xor key(i);
    end loop;
  --</manual>

    res <= t;
  end process calc;

  dout<= res;
end arch;
```

A.1.2 π-calculus code

```
fun concat/2.
fun land/2.
fun lnot/1.
private fun custom/2.
data ZERO/0.
data ONE/0.
data POSEDGE/0.
data NEGEDGE/0.
data TRUE/0.
data FALSE/0.

private free KEY,DIN.
free RST,CLK,IV,GO.

free clk_clock.

free rst_top,clk_top,IV_top,go_top,dout_top.
private free key_top,din_top.

private free go_cipher_top,mode_top.

private free rst_cipherbloc,clk_cipherbloc,start_cipherbloc,dout_cipherbloc,key_cipherbloc.

private free data_cipherbloc,res_cipherbloc.

fun xor/2.

query attacker: KEY.
query attacker: DIN.


let main = (
  in (clk_cipherbloc,clk_main_loc);
  in (rst_cipherbloc,rst_main_loc);
  in (data_cipherbloc,data_main_loc);
  in (start_cipherbloc,start_main_loc);
  in (res_cipherbloc,res_main_loc);

  if rst_main_loc=ZERO then
  (
    out(data_cipherbloc,start_main_loc)
  )
    else
  (
    if clk_main_loc=POSEDGE then
      out(data_cipherbloc,concat(data_main_loc,res_main_loc))
  )
).

let calc = (

  in(key_cipherbloc,key_calc_loc);
  in(data_cipherbloc,data_calc_loc);
```

```
  while

).

private free go_1,loop_1.

let while = (
  !(
    in(go_1,go_1_value);
    if go_1_value=TRUE then
    out(loop_1,TRUE)
  )
  |!(
    in(loop_1,loop_1_value);
    if loop_1_value=TRUE then
    (
      let t=xor(key_calc_loc,data_calc_loc) in
      out(res_cipherbloc,t);
       out(go_1,FALSE)
    )
   )
  | out(go_1,TRUE)
).

let cipherbloc =
  main
  | calc
  | (
      in(res_cipherbloc,res_cipherbloc_loc);
      out(dout_cipherbloc,res_cipherbloc_loc)
    ).

  data idle/0.
  data reset_cipher_bloc/0.
  data cipher/0.

  private free FSM_top.

  private free clk_fsm_proc,rst_fsm_proc.

let fsm_proc=

  in(rst_fsm_proc,rst_fsm_proc_loc);
  in(clk_fsm_proc,clk_fsm_proc_loc);
  in(FSM_top,FSM_fsm_proc_loc);
  in(go_top,go_fsm_proc_loc);

    if rst_fsm_proc_loc = ZERO then
    (
      out(FSM_top,idle)
    )
  else
  (
    if clk_fsm_proc_loc = POSEDGE then
        (
        if FSM_fsm_proc_loc=idle then out(FSM_top,reset_cipher_bloc)
```

```
        else if FSM_fsm_proc_loc=reset_cipher_bloc then out(FSM_top,cipher)
        else if go_fsm_proc_loc=ZERO then out(FSM_top,idle)
      )
  ).

let topseq =
  (
  in(FSM_top,FSM_top_loc);
  in(go_cipher_top,go_cipher_top_loc);
  in(mode_top,mode_top_loc);
  in(key_top,key_top_loc);
  in(IV_top,IV_top_loc);
  in(clk_top,clk_top_loc);
  in(din_top,din_top_loc);
  in(rst_top,rst_top_loc);

  if rst_top_loc=ZERO then out(go_cipher_top,ZERO)
  else if FSM_top_loc=reset_cipher_bloc then out(go_cipher_top,ZERO)
  else out(go_cipher_top,ONE);

  out(dout_top,(land(go_cipher_top_loc,xor(mode_top_loc,din_top_loc))));
  out(clk_clock,clk_top_loc)
  ).

let top =
topseq
 | ( in(dout_cipherbloc,dout_cipherbloc_tmp);
     out(mode_top,dout_cipherbloc_tmp) )
 | ( in(clk_top,clk_top_tmp);
     out(clk_cipherbloc,clk_top_tmp) )
 | ( in(go_cipher_top,go_cipher_top_tmp);
     out(rst_cipherbloc,go_cipher_top_tmp) )
 | ( in(IV_top,IV_top_tmp);
     out(start_cipherbloc,IV_top_tmp) )
 | ( in(key_top,key_top_tmp);
     out(key_cipherbloc,key_top_tmp) )
 | cipherbloc.

let init =
  out(FSM_top,idle);
  out(rst_top,RST);
  out(clk_clock,ZERO);
  out(IV_top,IV);
  out(go_top,GO);
  out(din_top,DIN);
  out(key_top,KEY).

let clock =
  in(clk_clock,prev);
  if prev=ZERO then out(clk_top,POSEDGE)
  else if prev=POSEDGE then out(clk_top,ONE)
  else if prev=ONE then out(clk_top,NEGEDGE)
  else if prev=NEGEDGE then out(clk_top,ZERO).

process init | !top | !clock
```

A.1.3 ProVerif analysis

```
-- Secrecy & events.
Starting rules:
Rule 0: equal:v_30,v_30
Rule 1: attacker:FALSE()
Rule 2: attacker:v_33 & attacker:v_32 -> attacker:land(v_33,v_32)
Rule 3: attacker:cipher()
Rule 4: attacker:v_35 & attacker:v_34 -> attacker:xor(v_35,v_34)
Rule 5: attacker:idle()
Rule 6: attacker:POSEDGE()
Rule 7: attacker:v_36 -> attacker:lnot(v_36)
Rule 8: attacker:ONE()
Rule 9: attacker:v_38 & attacker:v_37 -> attacker:concat(v_38,v_37)
Rule 10: attacker:TRUE()
Rule 11: attacker:NEGEDGE()
Rule 12: attacker:ZERO()
Rule 13: attacker:reset_cipher_bloc()
Rule 14: attacker:v_39 -> attacker:(v_39)
Rule 15: attacker:(v_40) -> attacker:v_40
Rule 16: mess:v_42,v_41 & attacker:v_42 -> attacker:v_41
Rule 17: attacker:v_44 & attacker:v_43 -> mess:v_44,v_43
Rule 18: attacker:dout_top[]
Rule 19: attacker:go_top[]
Rule 20: attacker:IV_top[]
Rule 21: attacker:clk_top[]
Rule 22: attacker:rst_top[]
Rule 23: attacker:clk_clock[]
Rule 24: attacker:GO[]
Rule 25: attacker:IV[]
Rule 26: attacker:CLK[]
Rule 27: attacker:RST[]
Rule 28: attacker:new_name[v_45]
Rule 29: mess:FSM_top[],idle()
Rule 30: attacker:RST[]
Rule 31: attacker:ZERO()
Rule 32: attacker:IV[]
Rule 33: attacker:GO[]
Rule 34: mess:din_top[],DIN[]
Rule 35: mess:key_top[],KEY[]
Rule 36: attacker:ZERO() & mess:din_top[],din_top_loc_55 & attacker:clk_top_loc_56 &
 attacker:IV_top_loc_57 & mess:key_top[],key_top_loc_58 &
 mess:mode_top[],mode_top_loc_59 & mess:go_cipher_top[],go_cipher_top_loc_60 &
 mess:FSM_top[],FSM_top_loc_61 -> mess:go_cipher_top[],ZERO()
Rule 37: rst_top_loc_63 <> ZERO() & attacker:rst_top_loc_63 &
 mess:din_top[],din_top_loc_64 & attacker:clk_top_loc_65 & attacker:IV_top_loc_66 &
 mess:key_top[],key_top_loc_67 & mess:mode_top[],mode_top_loc_68 &
 mess:go_cipher_top[],go_cipher_top_loc_69 &
 mess:FSM_top[],reset_cipher_bloc() -> mess:go_cipher_top[],ZERO()
Rule 38: FSM_top_loc_78 <> reset_cipher_bloc() & rst_top_loc_71 <> ZERO() &
 attacker:rst_top_loc_71 & mess:din_top[],din_top_loc_72 &
 attacker:clk_top_loc_73 & attacker:IV_top_loc_74 & mess:key_top[],key_top_loc_75 &
 mess:mode_top[],mode_top_loc_76 & mess:go_cipher_top[],go_cipher_top_loc_77 &
 mess:FSM_top[],FSM_top_loc_78 -> mess:go_cipher_top[],ONE()
Rule 39: FSM_top_loc_87 <> reset_cipher_bloc() & rst_top_loc_80 <> ZERO() &
```

```
 attacker:rst_top_loc_80 & mess:din_top[],din_top_loc_81 & attacker:clk_top_loc_82 &
 attacker:IV_top_loc_83 & mess:key_top[],key_top_loc_84 &
 mess:mode_top[],mode_top_loc_85 & mess:go_cipher_top[],go_cipher_top_loc_86 &
 mess:FSM_top[],FSM_top_loc_87 ->
  -> attacker:(land(go_cipher_top_loc_86,xor(mode_top_loc_85,din_top_loc_81)))
Rule 40: FSM_top_loc_96 <> reset_cipher_bloc() & rst_top_loc_89 <> ZERO() &
 attacker:rst_top_loc_89 & mess:din_top[],din_top_loc_90 & attacker:clk_top_loc_91 &
 attacker:IV_top_loc_92 & mess:key_top[],key_top_loc_93 &
 mess:mode_top[],mode_top_loc_94 & mess:go_cipher_top[],go_cipher_top_loc_95 &
 mess:FSM_top[],FSM_top_loc_96 -> attacker:clk_top_loc_91
Rule 41: mess:dout_cipherbloc[],dout_cipherbloc_tmp_99 ->
  -> mess:mode_top[],dout_cipherbloc_tmp_99
Rule 42: attacker:clk_top_tmp_102 -> mess:clk_cipherbloc[],clk_top_tmp_102
Rule 43: mess:go_cipher_top[],go_cipher_top_tmp_105 ->
  -> mess:rst_cipherbloc[],go_cipher_top_tmp_105
Rule 44: attacker:IV_top_tmp_108 -> mess:start_cipherbloc[],IV_top_tmp_108
Rule 45: mess:key_top[],key_top_tmp_111 -> mess:key_cipherbloc[],key_top_tmp_111
Rule 46: mess:res_cipherbloc[],res_main_loc_118 &
 mess:start_cipherbloc[],start_main_loc_119 &
 mess:data_cipherbloc[],data_main_loc_120 & mess:rst_cipherbloc[],ZERO() &
 mess:clk_cipherbloc[],clk_main_loc_121 -> mess:data_cipherbloc[],start_main_loc_119
Rule 47: rst_main_loc_126 <> ZERO() & mess:res_cipherbloc[],res_main_loc_123 &
 mess:start_cipherbloc[],start_main_loc_124 &
 mess:data_cipherbloc[],data_main_loc_125 & mess:rst_cipherbloc[],rst_main_loc_126 &
 mess:clk_cipherbloc[],POSEDGE() ->
  -> mess:data_cipherbloc[],concat(data_main_loc_125,res_main_loc_123)
Rule 48: mess:go_1[],TRUE() & mess:data_cipherbloc[],data_calc_loc_132 &
 mess:key_cipherbloc[],key_calc_loc_133 -> mess:loop_1[],TRUE()
Rule 49: mess:loop_1[],TRUE() & mess:data_cipherbloc[],data_calc_loc_138 &
 mess:key_cipherbloc[],key_calc_loc_139 ->
  -> mess:res_cipherbloc[],xor(key_calc_loc_139,data_calc_loc_138)
Rule 50: mess:loop_1[],TRUE() & mess:data_cipherbloc[],data_calc_loc_143 &
 mess:key_cipherbloc[],key_calc_loc_144 -> mess:go_1[],FALSE()
Rule 51: mess:data_cipherbloc[],data_calc_loc_148 &
 mess:key_cipherbloc[],key_calc_loc_149 -> mess:go_1[],TRUE()
Rule 52: mess:res_cipherbloc[],res_cipherbloc_loc_154 ->
  -> mess:dout_cipherbloc[],res_cipherbloc_loc_154
Rule 53: attacker:ZERO() -> attacker:POSEDGE()
Rule 54: attacker:POSEDGE() -> attacker:ONE()
Rule 55: attacker:ONE() -> attacker:NEGEDGE()
Rule 56: attacker:NEGEDGE() -> attacker:ZERO()
Completing...
Starting query not attacker:DIN[]
RESULT not attacker:DIN[] is true.

-- Secrecy & events.
Starting rules:
Rule 0: equal:v_271,v_271
Rule 1: attacker:FALSE()
Rule 2: attacker:v_274 & attacker:v_273 -> attacker:land(v_274,v_273)
Rule 3: attacker:cipher()
Rule 4: attacker:v_276 & attacker:v_275 -> attacker:xor(v_276,v_275)
Rule 5: attacker:idle()
Rule 6: attacker:POSEDGE()
Rule 7: attacker:v_277 -> attacker:lnot(v_277)
Rule 8: attacker:ONE()
```

```
Rule 9: attacker:v_279 & attacker:v_278 -> attacker:concat(v_279,v_278)
Rule 10: attacker:TRUE()
Rule 11: attacker:NEGEDGE()
Rule 12: attacker:ZERO()
Rule 13: attacker:reset_cipher_bloc()
Rule 14: attacker:v_280 -> attacker:(v_280)
Rule 15: attacker:(v_281) -> attacker:v_281
Rule 16: mess:v_283,v_282 & attacker:v_283 -> attacker:v_282
Rule 17: attacker:v_285 & attacker:v_284 -> mess:v_285,v_284
Rule 18: attacker:dout_top[]
Rule 19: attacker:go_top[]
Rule 20: attacker:IV_top[]
Rule 21: attacker:clk_top[]
Rule 22: attacker:rst_top[]
Rule 23: attacker:clk_clock[]
Rule 24: attacker:GO[]
Rule 25: attacker:IV[]
Rule 26: attacker:CLK[]
Rule 27: attacker:RST[]
Rule 28: attacker:new_name[v_286]
Rule 29: mess:FSM_top[],idle()
Rule 30: attacker:RST[]
Rule 31: attacker:ZERO()
Rule 32: attacker:IV[]
Rule 33: attacker:GO[]
Rule 34: mess:din_top[],DIN[]
Rule 35: mess:key_top[],KEY[]
Rule 36: attacker:ZERO() & mess:din_top[],din_top_loc_296 & attacker:clk_top_loc_297 &
 attacker:IV_top_loc_298 & mess:key_top[],key_top_loc_299 &
 mess:mode_top[],mode_top_loc_300 & mess:go_cipher_top[],go_cipher_top_loc_301 &
 mess:FSM_top[],FSM_top_loc_302 -> mess:go_cipher_top[],ZERO()
Rule 37: rst_top_loc_304 <> ZERO() & attacker:rst_top_loc_304 &
 mess:din_top[],din_top_loc_305 & attacker:clk_top_loc_306 & attacker:IV_top_loc_307 &
 mess:key_top[],key_top_loc_308 & mess:mode_top[],mode_top_loc_309 &
 mess:go_cipher_top[],go_cipher_top_loc_310 &
 mess:FSM_top[],reset_cipher_bloc() -> mess:go_cipher_top[],ZERO()
Rule 38: FSM_top_loc_319 <> reset_cipher_bloc() & rst_top_loc_312 <> ZERO() &
 attacker:rst_top_loc_312 & mess:din_top[],din_top_loc_313 & attacker:clk_top_loc_314 &
 attacker:IV_top_loc_315 & mess:key_top[],key_top_loc_316 &
 mess:mode_top[],mode_top_loc_317 & mess:go_cipher_top[],go_cipher_top_loc_318 &
 mess:FSM_top[],FSM_top_loc_319 -> mess:go_cipher_top[],ONE()
Rule 39: FSM_top_loc_328 <> reset_cipher_bloc() & rst_top_loc_321 <> ZERO() &
 attacker:rst_top_loc_321 & mess:din_top[],din_top_loc_322 & attacker:clk_top_loc_323 &
 attacker:IV_top_loc_324 & mess:key_top[],key_top_loc_325 &
 mess:mode_top[],mode_top_loc_326 & mess:go_cipher_top[],go_cipher_top_loc_327 &
 mess:FSM_top[],FSM_top_loc_328 ->
  -> attacker:(land(go_cipher_top_loc_327,xor(mode_top_loc_326,din_top_loc_322)))
Rule 40: FSM_top_loc_337 <> reset_cipher_bloc() & rst_top_loc_330 <> ZERO() &
 attacker:rst_top_loc_330 & mess:din_top[],din_top_loc_331 & attacker:clk_top_loc_332 &
 attacker:IV_top_loc_333 & mess:key_top[],key_top_loc_334 &
 mess:mode_top[],mode_top_loc_335 & mess:go_cipher_top[],go_cipher_top_loc_336 &
 mess:FSM_top[],FSM_top_loc_337 -> attacker:clk_top_loc_332
Rule 41: mess:dout_cipherbloc[],dout_cipherbloc_tmp_340 ->
  -> mess:mode_top[],dout_cipherbloc_tmp_340
Rule 42: attacker:clk_top_tmp_343 -> mess:clk_cipherbloc[],clk_top_tmp_343
Rule 43: mess:go_cipher_top[],go_cipher_top_tmp_346 ->
```

```
  -> mess:rst_cipherbloc[],go_cipher_top_tmp_346
Rule 44: attacker:IV_top_tmp_349 -> mess:start_cipherbloc[],IV_top_tmp_349
Rule 45: mess:key_top[],key_top_tmp_352 -> mess:key_cipherbloc[],key_top_tmp_352
Rule 46: mess:res_cipherbloc[],res_main_loc_359 &
 mess:start_cipherbloc[],start_main_loc_360 &
 mess:data_cipherbloc[],data_main_loc_361 & mess:rst_cipherbloc[],ZERO() &
 mess:clk_cipherbloc[],clk_main_loc_362 ->
  -> mess:data_cipherbloc[],start_main_loc_360
Rule 47: rst_main_loc_367 <> ZERO() & mess:res_cipherbloc[],res_main_loc_364 &
 mess:start_cipherbloc[],start_main_loc_365 &
 mess:data_cipherbloc[],data_main_loc_366 & mess:rst_cipherbloc[],rst_main_loc_367 &
 mess:clk_cipherbloc[],POSEDGE() ->
  -> mess:data_cipherbloc[],concat(data_main_loc_366,res_main_loc_364)
Rule 48: mess:go_1[],TRUE() & mess:data_cipherbloc[],data_calc_loc_373 &
 mess:key_cipherbloc[],key_calc_loc_374 -> mess:loop_1[],TRUE()
Rule 49: mess:loop_1[],TRUE() & mess:data_cipherbloc[],data_calc_loc_379 &
 mess:key_cipherbloc[],key_calc_loc_380 ->
  -> mess:res_cipherbloc[],xor(key_calc_loc_380,data_calc_loc_379)
Rule 50: mess:loop_1[],TRUE() & mess:data_cipherbloc[],data_calc_loc_384 &
 mess:key_cipherbloc[],key_calc_loc_385 -> mess:go_1[],FALSE()
Rule 51: mess:data_cipherbloc[],data_calc_loc_389 &
 mess:key_cipherbloc[],key_calc_loc_390 -> mess:go_1[],TRUE()
Rule 52: mess:res_cipherbloc[],res_cipherbloc_loc_395 ->
  -> mess:dout_cipherbloc[],res_cipherbloc_loc_395
Rule 53: attacker:ZERO() -> attacker:POSEDGE()
Rule 54: attacker:POSEDGE() -> attacker:ONE()
Rule 55: attacker:ONE() -> attacker:NEGEDGE()
Rule 56: attacker:NEGEDGE() -> attacker:ZERO()
Completing...
Starting query not attacker:KEY[]
RESULT not attacker:KEY[] is true.
```

A.2 An example on a 2-element model

We take the case of a CBC mode[2] to test the methodology described in §7.
This circuit is made of a top entity which embodies a lower-level entity that actually performs the ciphering operation needed for the whole unit to work properly.
Our aim is to prove through the type system that securing the input channels where we input the key to the circuit is enough to prevent the key from being diffused, *i.e.* there is no lack of information on other channels.

A.2.1 The *m*VHDL code of the CBC

```
------------------------------------------------------------------
FILE: mode_cbc.vhdl
------------------------------------------------------------------

[ ... ]

begin
  mode : process (clk, rst)
```

[2] Code kindly written by Nicolas Guillermin.

```
  begin
    if rst = '0' then
      FSM<="001";
      mode_register<=(others=>'0');
      prim_start<= '0';
    elsif clk'event and clk = '0' then
      if FSM="001" then
        if get_IV = '1' then
          mode_register<= IV_in;
          FSM<="010";
        end if;
      elsif FSM="010" then
        if read_dout='1' then
          FSM<="011";
        end if;
      elsif FSM = "011" then
        if get_din = '1' then
          mode_register<=mode_register xor din;
          prim_start<='1';
          FSM<="100";
        end if;
      elsif FSM = "100" then
        prim_start<='0';
        if prim_ready = '1' then
          mode_register<= prim_dout;
          FSM<="010";
        end if;
      end if;
    end if;
  end process;

  pr : primitive port map (
    clk    => clk,
    rst    => rst,
    start  => prim_start,
    input  => mode_register,
    key    => key,
    output => prim_dout,
    ready    => prim_ready);

  dout<= mode_register when FSM="010" else (others=>'0');
  need_IV<='1' when FSM = "001" else '0';
  need_din<='1' when FSM = "011" else '0';
  ready_dout<='1' when FSM = "010" else '0';
  busy<= not prim_ready;

[ ... ]

------------------------------------------------------------
FILE: primitive.vhdl
------------------------------------------------------------

[ ... ]

begin
  fsm_proc: process (clk, rst)
```

```
  begin
    if rst = '0' then
      FSM <="000000000000001";
      result<= (others=>'0');
      subkey<= (others=>'0');
    elsif clk'event and clk = '1' then
      case FSM is
        when "000000000000001" =>
          if start = '1' then
            result<= input xor key(127 downto 0);
            FSM<= FSM(13 downto 0) & FSM(14 downto 14);
            subkey<= key(255 downto 128);
          end if;
        when others =>
          FSM<= FSM(13 downto 0) & FSM(14 downto 14);
          result<= algo_turn xor subkey;
          subkey<= subkey (112 downto 0) & (subkey(127 downto 113) xor FSM );
      end case;
    end if;
  end process fsm_proc;

  algo_turn <= ((result(29 downto 0 ) & result(127 downto 30))) xor CST ;
  output <= result when FSM ="000000000000001"  else (others=>'0');
  ready<= '1' when FSM = "000000000000001" else '0';

[ ... ]
```

A.2.2 The MACE4 input file for the CBC

There is a loop in the *m*VHDL description of the CBC: this had to be unfolded in order to allow the analysis to proceed.

```
[ ... ]

% XOR equational theory
xor(x,y)=xor(y,x).
xor(x,xor(y,z))=xor(xor(x,y),z).
xor(xor(x,x),y)=y.

% Allowed functions
mknows(x) & mknows(y) -> mknows(xor(x,y)).
mknows(x) <-> mknows(not(x)).
mknows(x) & mknows(y) <-> mknows(concat(x,y)).
mknows(x) <-> mknows(invertible(x)).

%Input values
mknows(x) -> value(SrstCBC,x).
mknows(x) -> value(SclkCBC,x).
mknows(x) -> value(Sget_IVCBC,x).
mknows(x) -> value(Sget_dinCBC,x).
mknows(x) -> value(Sread_doutCBC,x).
mknows(x) -> value(SkeyCBC,x).
mknows(x) -> value(SIV_inCBC,x).
mknows(x) -> value(SdinCBC,x).

%Output values
```

```
value(Sneed_IVCBC,x) -> mknows(x).
value(Sneed_dinCBC,x) -> mknows(x).
value(SdoutCBC,x) -> mknows(x).
value(Sready_doutCBC,x) -> mknows(x).
value(SbusyCBC,x) -> mknows(x).

%Initial knowledge
mknows(B0).
mknows(B1).
mknows(CST).
mknows(cDin).
mknows(cIV).

% Initial state
value(SrstCBC,B0).
value(SclkCBC,B1).
value(SkeyCBC,cKey).
value(SIV_inCBC,cIV).
value(Sget_IVCBC,B1).
value(SdinCBC,cDin).
value(Sget_dinCBC,B1).
value(Sread_doutCBC,B0).

%Internal description - CBC
value(SrstCBC,B0) -> value(SFSMCBC,B001) & value(Smode_registerCBC,B0) &
value(Sprim_startCBC,B0).
value(SrstCBC,B1) & value(SclkCBC,B1) & value(SIV_inCBC,v1) &
value(Sget_IVCBC,B1) & value(SFSMCBC,B001) -> value(SrstCBC,B1) &
value(SclkCBC,B1) & value(SIV_inCBC,v1) & value(Sget_IVCBC,B1) &
value(SFSMCBC,B010) & value(Smode_registerCBC,v1).

[ ... ]

%Internal description - PR (primitive)
value(SrstPR,B0) -> value(SresultPR,B0) & value(SFSMPR,D1) &
value(SsubkeyPR,B0).
value(SstartPR,B1) & value(SclkPR,B1) & value(SkeyPR,v1) &
value(SinputPR,v2) & value(SFSMPR,D1) -> value(SstartPR,B1) &
value(SclkPR,B1) & value(SkeyPR,v1) & value(SinputPR,v2) &
value(SresultPR01,xor(invertible(v1),v2)) & value(SFSMPR,invertible(D1)) &
value(SsubkeyPR,invertible(v1)).
value(SclkPR,B1) & -value(SFSMPR,D1) & value(SFSMPR,v3) &
value(Salgo_turn01PR,v1) & value(SsubkeyPR,v2) -> value(SclkPR,B1) &
value(SresultPR02,xor(v1,v2)) & value(SFSMPR,invertible(v3)) &
value(Salgo_turn01PR,v1) &
value(SsubkeyPR,concat(invertible(v2),xor(invertible(v2),invertible(v3)))).

[ ... ]

%Internal description - Port maps
value(SclkCBC,v1) <-> value(SclkPR,v1).
value(SrstCBC,v1) <-> value(SrstPR,v1).
value(Sprim_startCBC,v1) <-> value(SstartPR,v1).
value(Smode_registerCBC,v1) <-> value(SinputPR,v1).
value(SkeyCBC,v1) <-> value(SkeyPR,v1).
value(Sprim_doutCBC,v1) <-> value(SoutputPR,v1).
```

```
value(Sprim_readyCBC,v1) <-> value(SreadyPR,v1).

%Secrecy goal
-mknows(cKey).

%Hints
Salgo_turn01PR=1.
Salgo_turn02PR=0.
Salgo_turn03PR=1.

[ ... ]
```

A.2.3 Results of the analysis

MACE4 succeeds in finding a model for the system:

```
For domain size 2.
Current CPU time: 0.00 seconds (total CPU time: 0.03 seconds).
Ground clauses: seen=547, kept=547.
Selections=18, assignments=18, propagations=67, current models=1.
Rewrite terms=4159, rewrite bools=662, indexes=1332.
Rules from neg clauses=31, cross offs=31.
```

The analysis outputs a 2-element model, that defines two types of data and two types of channels: for data this corresponds exactly to the distinction between secret and public, for channels to the distinction between private and public.
From this analysis we can draw the proof that the system does not lack the secret data (*i.e.* the key) in whatever way it is used if the malicious user cannot read data on the channels `SKeyCBC`: this channel is the red input in figure A.1.
In the remainder of this subsection, the details of the model.

Data types

As it can be seen in Table A.1, only `cKey` has to be secret data (no wonder, it was the security goal we set), all other data is public.

B0:	0
B001:	0
B010:	0
B011:	0
B1:	0
B100:	0
CST:	0
D1:	0
cDin:	0
cIV:	0
cKey:	1

Table A.1: Data types.

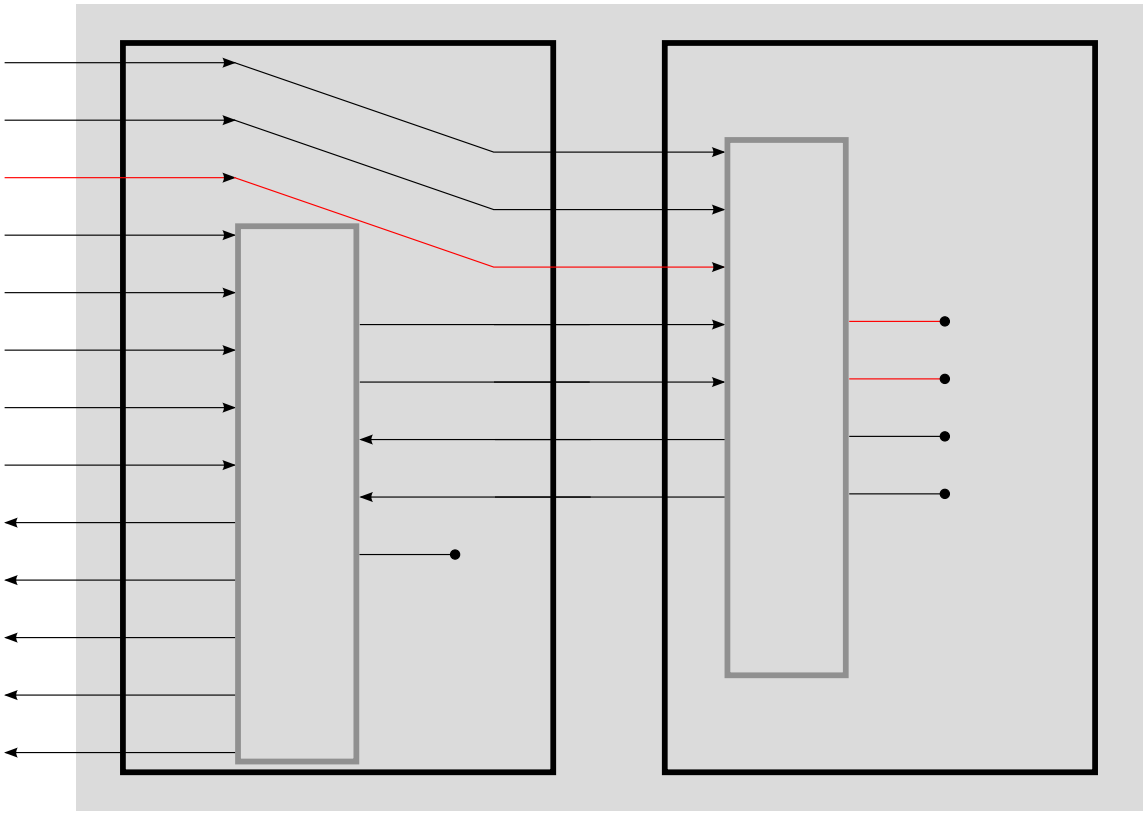


Figure A.1: Result of the analysis on the mode CBC: the only channels that need to be red are those ones where the key flows, all the others can be black.

Channel types

The channel types determined by MACE4 are shown in Table A.2.

Function tables

The function tables output by MACE4 are exactly the ones we were expecting: `linear`, `not` and `concat` are type-preserving functions, while `xor` is a type altering function.
The function tables are shown in Table A.3.

Predicate tables

Predicate tables are coherent, as with the truth tables found — that can be seen in Table A.4 — a contradiction can be derived only if the malicious user is able to derive a secret value.

SFSMCBC: 0	SFSMPR: 0
SIV_inCBC: 0	Salgo_turn01PR: 1
Salgo_turn02PR: 0	Salgo_turn03PR: 1
Salgo_turn04PR: 0	Salgo_turn05PR: 1
Salgo_turn06PR: 0	Salgo_turn07PR: 1
Salgo_turn08PR: 0	Salgo_turn09PR: 1
Salgo_turn10PR: 0	Salgo_turn11PR: 1
Salgo_turn12PR: 0	Salgo_turn13PR: 1
Salgo_turn14PR: 0	SbusyCBC: 0
SclkCBC: 0	SclkPR: 0
SdinCBC: 0	SdoutCBC: 0
Sget_IVCBC: 0	Sget_dinCBC: 0
SinputPR: 0	SkeyCBC: 1
SkeyPR: 1	Smode_registerCBC: 0
Sneed_IVCBC: 0	Sneed_dinCBC: 0
SoutputPR: 0	Sprim_doutCBC: 0
Sprim_readyCBC: 0	Sprim_startCBC: 0
Sread_doutCBC: 0	SreadyPR: 0
Sready_doutCBC: 0	SresultPR: 0
SresultPR01: 1	SresultPR02: 0
SresultPR03: 1	SresultPR04: 0
SresultPR05: 1	SresultPR06: 0
SresultPR07: 1	SresultPR08: 0
SresultPR09: 1	SresultPR10: 0
SresultPR11: 1	SresultPR12: 0
SresultPR13: 1	SresultPR14: 0
SrstCBC: 0	SrstPR: 0
SstartPR: 0	SsubkeyPR: 1

Table A.2: Channel types.

linear:	0	1
	0	1

concat:	0	1
0	0	1
1	1	1

xor:	0	1
0	0	1
1	1	0

Table A.3: Function tables.

mknows:	0	1
	1	0

value:	0	1
0	1	0
1	1	1

Table A.4: Predicate tables

Acknowledgements

Questa tesi rappresenta per me la fine di un percorso, una conclusione attesa per anni, una conclusione tuttavia che non ho mai cercato con troppa convinzione.

È indubbio che non abbia fatto nulla per affrettare questo viaggio (piuttosto il contrario); approdo alla meta ricco dei tesori accumulati per strada, senza aspettarmi ricchezze da questo traguardo: ingegneria elettronica mi ha dato il bel viaggio, senza di lei mai mi sarei messo sulla strada, ma adesso non mi aspetto nulla più.

Il valore vero sta nel viaggio e questa è la ragione di queste ultime pagine, alle quali ho dedicato particolare cura, in cui voglio ringraziare le persone che mi hanno accompagnato lungo il cammino.

This master thesis represents for me the end of a path, a long-awaited conclusion, nevertheless a conclusion which I have never sought with too much conviction.

There is no doubt that I didn't do anything to hurry the journey (quite the opposite); I reach my destination wealthy with all the treasures I've gained on the way, not expecting any richness from this goal: electronic engineering gave me the marvelous journey, without her I wouldn't have set out, but now I'm expecting nothing more.

The true value is in the journey and this is the reason of these last pages, to which I have devoted special care, where I want to thank the people who have accompanied me along the way.

Innanzitutto il primo ringraziamento va a Chiara Raccagni per la sua *terrolosità* (è stato necessario coniare un termine *ad hoc*): anche se ha vissuto con me solo gli strascichi della mia travagliata storia con Elettronica ed UniPi, il solo esserci stata in questi ultimi anni è stato fondamentale. Non le sarò mai abbastanza grato per essere la persona più splendida che abbia mai incontrato, per il sorriso che i suoi occhi tradiscono anche con il broncio, per la sua premurosa dolcezza.

Gianluca Meneghello e le sue idee folli sono stati una compagnia costante fin dai tempi del liceo, declinata in modo diverso a seconda del periodo storico — dall'adolescenziale *da sbrégo* all'*uffa* dell'età adulta.

Veronica Fichera è tra le persone che più mi sono mancate durante la mia lunga assenza da Verona, ma il suo supporto al di là della distanza è stato molto importante.

Sono ormai sette anni che conosco Serena Salehzadeh, la ringrazio per le pernacchie da mandare via mail, per la sua risata, per Mary Poppins come croce e delizia al tempo stesso, per esserci sempre.

Grazie ad Andrea Bedini, fidato compagno dai tempi del soggiorno parigino che ha portato a questa tesi: un brindisi all'*Hideout*, teatro di discussioni nerd/filosofiche e piani diabolici (soprattutto nella loro infruttuosità).

Ho un grosso debito di riconoscenza verso Caterina Sganga per essere stata amica e mentore, per tutti i buoni consigli dispensati da quando la conosco e per la sua esemplare determinazione.

Il tempo che ho passato a Scuola non avrebbe potuto essere lo stesso senza l'amicizia, la disponibilità ed il supporto di Maurizio Himmelmann.

Il ricordo delle merende *grasssse* con Leonardo Magneschi — che ha sempre saputo apprezzare l'essere *grassi nell'anima* — impone un momento di seria riflessione... Quando ripetiamo?!

Le poche occasioni in cui arrivavo fino a via Diotisalvi sarebbero state di ancora più rara occorrenza se non ci fosse stato Claudio Tagliabue.

Di sicuro non avrei potuto arrivare in fondo ad Ingegneria Elettronica senza Claudio Nani e Calogero Oddo, fidati pusher di appunti.

Tanti anni nello stesso collegio con Francesca Giraudo sono stati caffè, discussioni sul caffè, discussioni sulla gente, discussioni sulle trame di palazzo, discussioni sulle discussioni, scazzi, battute, frecciatine, corse (molto di rado in realtà), giri in bici, rimproveri, supporto, risate, divertimento, amicizia e molto molto altro... Grazie davvero di tutto!

Marco Bilanceri e Luca Martone hanno contribuito in maniera sostanziale all'instaurazione del nostro salotto caffé, un istituto che è durato per parecchio tempo e del quale se ne sente la mancanza.

Grazie ai (più o meno) nottambuli che hanno preso parte ad interminabili partite di *Age of Empires II*, più o meno continuativamente tra il 2004 ed il 2008, ed in particolare a Paolo Addis, Andrea e Luca Baù, Simone Brancatello, Stefano Burattini, Enrico Cecchi, Roberto Farolfi, Graziano "Sogliola" Giannecchini, Massimo Grava, Michele Magistrelli, Marino Pagan, Marco "Tarzan" Rocca e Michele Sciacca.

Sempre in relazione alla vita nerd del centro di calcolo, è stato un piacere poter inveire contro l'universo informatico in compagnia di Emiliano "Testina" Vavassori e Riccardo Vestrini, nonché con il mio tuttologo di fiducia Riccardo Brigo.

Un ringraziamento a Paolo "Dyno" Frumento e Graziella Donatelli per la loro amicizia, gentilezza e disponibilità.

Il mio cluster di donne preferito non mi ha mai fatto mancare il suo supporto: mille grazie ad Ilaria Sacco, Alice Sanna e Laura D'Angelo.

Tra le "nuove leve" (4 anni fa...) devo ringraziare Angela Cossu, Valentina da Prat, Margherita Notarnicola e Paola Sindaco, che mi hanno sempre fatto sentire a casa ogni volta che tornavo in una Scuola sempre più popolata da gente che non conoscevo.

Tornare a Pisa sempre per i soliti esami ha avuto i suoi lati positivi, perché mi ha dato modo di rivedere periodicamente (frequentemente?) amici come Andrea Azzini, Michele Basile, Marco Bonizzato, Marco Cempini, Federico Dragoni, Enrico Frumento, Giuseppe Genova, Matteo Gnocato, Esther Maragò, Gabriele Ricco e Marco "Belin" Rizzone.

La colonia londinese della SSSUP annovera tra i suoi Mattia Lazzerini, Francesco Podestà e Annalisa Tore: nella loro breve permanenza a Scuola per me hanno saputo contare davvero.

Grazie a Luca di Mauro, Thomas Mazzocco e Alberto Montagner, dispersi tra Parigi, Stirling e New York, in attesa del prossimo *catching up* in qualche dove.

Un ringraziamento a Gianni Tonioni per la fiducia che mi ha sempre accordato... anche su questioni accademiche, per le quali il mio parere non era considerato dai più come particolarmente "autorevole".

Mille grazie ad Ada Iovkova per tante cose. Un grazie in particolare per il paziente babysitting, che ha fatto sì io potessi arrivare a laurearmi in tempi decenti quantomeno alla triennale. Senza dimenticare un ringraziamento per avermi insegnato l'alfabeto cirillico, parole come *пиклив* o *лигав*, ma più in generale

per avermi dato ragioni per imparare il Bulgaro.

A questo proposito è d'obbligo ringraziare i miei insegnanti di Bulgaro ad UniPi, il prof. Giuseppe Dell'Agata e Ani Stancheva: senza dubbio si è trattato delle lezioni che ho seguito con maggior piacere durante la mia carriera universitaria (nonché in assoluto le più numerose!). E non va dimenticato che se ho rispettato (a filo) i requisiti minimi per restare in SSSUP dopo il quarto anno, sia in termini di media che di numero di esami, è stato solo grazie a quell'esame.

Българският език е причината, поради която се запознах със страхотни хора, като Alessandra Bonsignori, Lea Geringová, Антонина Иванова и Anna Kiššová.

През последните няколко години Цветелина Найденова и Надя Лазарова бяха не само мои приятелки, но и (много търпеливи) преподавателки по български, и бих искал да им благодаря за всичко

Благодаря на една от най-милите жени на света, Невена Елисеева, на която винаги мога да разчитам. И която винаги е готова да ме научи на малко *български*.

Un ringraziamento a Claudia Palla ed Enza Porpiglia, che non si sono limitate a foraggiare una banda di spocchiosi, ma lo hanno sempre fatto con un sorriso. Beh certo, il sorriso era *ad personam*, ma anche giustamente — certa gente *'un la si poteva vede'*...

Sono contento che Stefania Pizzini e Serena Segatori non siano state semplicemente "gente che lavora al Sant'Anna" e a loro devo davvero tanto. Senza dimenticare un grazie alla piccola Anna, che mi ha regalato un voto immeritatamente alto in uno degli ultimi esami.

Vielen Dank zu Jana Rebecca Tavender, sie ist eine sehr wichtige Freundin seit langer Zeit und ich habe viel von ihr in diesen Jahren gelernt.

Thanks to my travel companions Lorenn Ruster and Hadi Rabbani, great friends since the time I was in Paris (Cachan, to be precise...): I have the dearest memory of the time we have spent together and I hope we'll soon be able to meet and discover new destinations together.

Un ringraziamento ad Eleonora Castaldo e Laura Turrini, persone stupende che hanno saputo rendere "casa" addirittura il Bâtiment M.

Pendant mon séjour parisien j'ai connu des autres jeunes vraiment exceptionnels, qui ont rendu mes jours en France autant exceptionnels: pour cela je dois remercier Saori Conchet, Labib Daoud, Alkisti Florou, Giovanni Fornasa, Nicole Gayard, Svilen Iskrov, Aude Lanoe, Solène Mallet, Alex Meloni, Anna Monusova, Carmelina Rea, Chiara Saggioro, Jana Vancikova et Julia Whitby. Per le stesse ragioni grazie anche a Simona Beltramo, che concettualmente rientra in quest'elenco, ma che a rigore conoscevo già da ben prima di andare in Francia.

Ringrazio sentitamente la prof. Maria Chiara Carrozza per il suo supporto in relazione alla mia fuga verso l'ENS-Cachan, l'informatica e la Francia; je remercie aussi tous ceux que j'ai rencontré là-bas au LSV: Hedi Benzina, Elie Burszstein, Stefan Ciobaca, Jean Goubalt-Larrecq, Manuela Grindei, Benôit Groz et Steve Kremer.

Serbo un ottimo ricordo della cultura e della finezza intellettuale di Elena Tabacchi, chiaramente senza dimenticare le sue indiscutibili qualità come persona, e le sono estremamente grato per tutto questo.

Много благодарности на любимата ми маруля, Весела Евтимова, не просто намусената съквартиранка, която готви риба цялото време, а една истинска приятелка.

Many thanks to my other flatmate Tamara Jurca, it's been a pleasure sharing the flat with her (and with her husband Darko Novakovic, when he was in Dublin) for the last couple of years.

Thanks to the great good giant German guy Marc Rottler, arguably one of the kindest people I've ever met.

Muchas gracias a Neus Mariejas, maravillosa voz de los *Grafton 42*, por el tiempo juntos en la primavera 2009, entre *Ciutadelas*, musica y ajedrez.

Grazie a Susanna Foppoli, che è diventata un'amica stretta fin dal mio primo anno in Irlanda, quando si girava con la mandria Erasmus: è stata una presenza importante, in assenza della quale tante cose sarebbero state diverse.

My stay in Ireland gave me the chance to spend some time with other people who have also contributed to make it a worthwhile experience: Diego Albano, Laura Casarotto, Andrea Cerone, Sara Costanzo, Fanny Crabié, Federica de Sisto, Sara Fikrat, Alessio Frenda, Giulia Giannotti, Paweł Gancarski, Christine Groba, Luca Longo, Valentina Paolucci, Lorenzo Piccoli, Enrico Vendramin, Marco Vignoli.

Nonostante tutti questi anni lontano da casa è sempre rimasto il rapporto con la *Compagnia*, con modalità diverse a seconda del momento storico e delle congiunzioni astrali: le mutevoli dinamiche interne ne hanno fatto un essere dotato di vita propria, che ha a tratti rivestito un ruolo fondamentale per me negli ultimi dodici anni. In maniera più o meno effimera ne hanno fatto parte tante persone nel corso degli anni, il mio ringraziamento va a tutti loro ed in particolare a Natalia Fanton, Davide Tosetti, Francesca Vanzo, Elena Zanetti.

Un ringraziamento all'amico di sempre, Raffaele Cutolo, ~~anche se non so dove cavolo sia finito~~ periodicamente disperso ma sempre puntualmente ritrovato.

Grazie a Chiara Pettenella, ricordo sempre con affetto il suo "tranquillo Ric, qui da noi troverai sempre un piatto caldo ed un sorriso".

Grazie a Giulia Inzalaco, perché nel momento del bisogno lei c'era, e grazie a Marinella Marani e Silvia Burro, amiche dai tempi del liceo ma ancora più amiche negli ultimi anni: un fantastico trio che contribuisce considerevolmente a farmi sentire mancanza di casa.

Devo tantissimo a Pietro Antolini e Francesco Campara: abbiamo condiviso tanto nel tempo passato insieme dai primi anni del liceo e questo fa di loro due amici insostituibili.

Nel quinquennio al Fracastoro hanno avuto un ruolo fondamentale le mie prof Mimia Padoan e Bianca Barattelli, la cui presenza non è mai venuta a mancare in tutti questi anni.

Last but not least, la *famigghia*, a cui questa laurea è sempre stata più a cuore di quanto mai lo sia stata per me: grazie a mamma e papà per il loro affetto e supporto, per la loro sfibrante insistenza nel domandare quando avrei fatto gli esami e quando mi sarei laureato, per non aver mai cercato di condizionarmi nelle mie scelte; grazie a mia sorella Jenny, per essere in qualche modo sopravvissuta ad un fratello come me; grazie a nonne, zii e cugini, che mi hanno sempre incoraggiato; grazie a Barbara, Silvia, Luca e Giorgio per avermi praticamente adottato.

Infine un ringraziamento a chi di questa tesi ha letto solamente dedica, citazioni e ringraziamenti, a chi l'ha semplicemente sfogliata distrattamente e a chi non ha fatto nemmeno questo.

Slainte.

References

[AB05] Martín Abadi and Bruno Blanchet. "Analyzing security protocols with secrecy types and logic programs". In: *J. ACM* 52.1 (2005), pp. 102–146.

[Aba99] Martín Abadi. "Secrecy by typing in security protocols". In: *J. ACM* 46.5 (1999), pp. 749–786.

[ABCL09] Martín Abadi, Bruno Blanchet, and Hubert Comon-Lundh. "Models and Proofs of Protocol Security: A Progress Report". In: *21st International Conference on Computer Aided Verification (CAV'09)*. Lecture Notes on Computer Science. Grenoble, France: Springer Verlag, July 2009.

[ABF07] Martín Abadi, Bruno Blanchet, and Cédric Fournet. "Just fast keying in the pi calculus". In: *ACM Trans. Inf. Syst. Secur.* 10.3 (2007), p. 9.

[Ace+07] Luca Aceto et al. *Reactive Systems: Modelling, Specification and Verification*. New York, NY, USA: Cambridge University Press, 2007.

[Adã+06] Pedro Adão et al. "Towards a Quantitative Analysis of Security Protocols". In: *Electr. Notes Theor. Comput. Sci.* 164.3 (2006), pp. 3–25.

[AF01] Martín Abadi and Cédric Fournet. "Mobile values, new names, and secure communication". In: *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. London, United Kingdom: ACM, 2001, pp. 104–115.

[AG97] Martín Abadi and Andrew D. Gordon. "A Calculus for Cryptographic Protocols: The Spi Calculus". In: *Fourth ACM Conference on Computer and Communications Security*. ACM Press, 1997, pp. 36–47.

[AR02] Martín Abadi and Phillip Rogaway. "Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)". In: *J. Cryptology* 15.2 (2002), pp. 103–127.

[Ash08] Peter J. Ashenden. *The Designer's Guide to VHDL*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[Bau06] Mathieu Baudet. "Random Polynomial-Time Attacks and Dolev-Yao Models". In: *Journal of Automata, Languages and Combinatorics* 11.1 (2006), pp. 7–21.

[Ber08] Gérard Berry. "Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel". In: *Formal Methods for Industrial Critical Systems*. Ed. by Stefan Leue and Pedro Merino. Vol. 4916. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008.

[BH06] J.P. Bowen and M.G. Hinchey. "Ten commandments of formal methods... ten years later". In: *Computer* 39.1 (Jan. 2006), pp. 40–48.

[BH95a] J.P. Bowen and M.G. Hinchey. "Seven more myths of formal methods". In: *Software, IEEE* 12.4 (July 1995), pp. 34–41.

[BH95b] J.P. Bowen and M.G. Hinchey. "Ten commandments of formal methods". In: *Computer* 28.4 (Apr. 1995), pp. 56 –63.

[Bla01] Bruno Blanchet. "An efficient cryptographic protocol verifier based on Prolog rules". In: *14th IEEE Computer Security Foundations Workshop*. 2001, pp. 86–100.

[Bla04] Bruno Blanchet. "Automatic Proof of Strong Secrecy for Security Protocols". In: *Security and Privacy, IEEE Symposium on* 0 (2004), p. 86.

[Bla08] Bruno Blanchet. "Vérification automatique de protocoles cryptographiques: modèle formel et modèle calculatoire". Mémoire d'habilitation à diriger des recherches. Université Paris-Dauphine, Nov. 2008.

[Bla10] Bruno Blanchet. *ProVerif Automatic Cryptographic Protocol Verifier User Manual*. July 2010 (accessed on September 2nd, 2010). URL: `http://www.proverif.ens.fr/proverif-manual.pdf`.

[BR70] J. N. Buxton and B. Randell, eds. *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. NATO Science Committee, 1970.

[Cha99] K. C. Chang. *Digital Systems Design with VHDL and Synthesis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999.

[Cip95] Barry Cipra. "How number theory got the best of the Pentium chip". In: *Science* 267.5195 (1995), p. 175.

[Coh99] Ben Cohen, ed. *VHDL Coding Styles and Methodologies*. Norwell, MA, USA: Kluwer Academic Publishers, 1999.

[Dij72] Edsger W. Dijkstra. "Chapter I: Notes on structured programming". In: (1972), pp. 1–82.

[DMV04] Paul Hankes Drielsma, Sebastian Mödersheim, and Luca Viganò. "A Formalization of Off-Line Guessing for Security Protocol Analysis". In: *LPAR*. 2004, pp. 363–379.

[DRS08] Stéphanie Delaune, Mark D. Ryan, and Ben Smyth. "Automatic verification of privacy properties in the applied pi-calculus". In: *Proceedings of the 2nd Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM'08)*. Ed. by Yuecel Karabulut et al. Vol. 263. IFIP Conference Proceedings. Trondheim, Norway: Springer, June 2008, pp. 263–278.

[DY83] Danny Dolev and Andrew C. Yao. "On the security of public-key protocols". In: *IEEE Transaction on Information Theory* 2.29 (Mar. 1983), pp. 198–208.

[Fdr] *Failures-Divergence Refinement, FDR2 User Manual*. 6th. Formal Systems (Europe) Ltd. June 2005 (accessed on September 4th, 2010). URL: `http://www.fsel.com/fdr2_manual.html`.

[Gen10] Harry J. Gensler. *Introduction to Logic*. Routledge, 2010.

[GK83] D.D. Gajski and R.H. Kuhn. "New VLSI Tools". In: *Computer* 16.12 (Dec. 1983), pp. 11 –14.

[GL08] Jean Goubault-Larrecq. *Towards Producing Formally Checkable Security Proofs, Automatically*. Rapport de Recherche LSV-08-03. LSV, ENS Cachan, 2008.

[GL94] Manfred Selz Gunther Lehmann Bernhard Wunder. *Schaltungsdesign mit VHDL*. Poing, Germany: Franzis, 1994.

[GLP05] Jean Goubault-Larrecq and Fabrice Parrennes. “Cryptographic protocol analysis on real C code”. In: *In 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05), volume 3385 of LNCS*. Springer, 2005, pp. 363–379.

[Gol03] Warren Goldfarb. *Deductive Logic*. Hackett Publishing, 2003.

[Goo95] Kees G. W. Goossens. “Reasoning about VHDL using operational and observational semantics”. In: *CHARME '95: Proceedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. London, UK: Springer-Verlag, 1995, pp. 311–327.

[Hal90] A. Hall. “Seven myths of formal methods”. In: *Software, IEEE* 7.5 (Sept. 1990), pp. 11–19.

[HK91] Ian Houston and Steve King. “CICS Project Report: Experiences and Results from the use of Z in IBM”. In: *VDM Europe (1)*. 1991, pp. 588–596.

[HM09] C. A. R. Hoare and Jayadev Misra. “Preface to special issue on software verification”. In: *ACM Comput. Surv.* 41.4 (2009).

[Hoa02] C. A. R. Hoare. “Assertions in Modern Software Engineering Practice”. In: *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 459–462.

[Hoa+09] C.A.R. Hoare et al. “The verified software initiative: A manifesto”. In: *ACM Comput. Surv.* 41.4 (2009).

[Huf+10] T. Huffmire et al. *Handbook of FPGA Design Security*. Springer, 2010.

[Hym02] Charles Hymans. “Checking safety properties of behavioral VHDL descriptions by abstract interpretation”. In: *In 9th International Static Analysis Symposium (SAS02)*. Springer, 2002, pp. 444–460.

[Hym04] Charles Hymans. “Vérification de composants VHDL par interprétation abstraite”. PhD thesis. Paris: École Polytechnique, 2004.

[Iee] *IEEE Standard VHDL Language Reference Manual*. 1076-2008. IEEE. 2009.

[Kle67] Stephen Cole Kleene. *Mathematical Logic*. Dover Publications, 1967.

[Koo07] Philip Koopman. “Reliability, Safety, and Security in Everyday Embedded Systems (Extended Abstract)”. In: *LADC*. 2007, pp. 1–2.

[Low95] Gavin Lowe. “An attack on the Needham-Schroeder public-key authentication protocol”. In: *Inf. Process. Lett.* 56.3 (1995), pp. 131–133.

[Low96] Gavin Lowe. “Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR”. In: *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*. London, UK: Springer-Verlag, 1996, pp. 147–166.

[LV05] Peeter Laud and Varmo Vene. “A Type System for Computationally Secure Information Flow”. In: *FCT*. 2005, pp. 365–377.

[Mäd09] Andreas Mäder. *VHDL Kompakt*. 2009 (accessed on September 2nd, 2010). URL: `http://tams-www.informatik.uni-hamburg.de/vhdl/doc/ajmMaterial/vhdl.pdf`.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus*. New York, NY, USA: Cambridge University Press, 1999.

[Mis00] Michael W. Mislove. "Nondeterminism and Probabilistic Choice: Obeying the Laws". In: *CONCUR 2000, LNCS 1877*. 2000, pp. 350–364.

[Mit+01] John C. Mitchell et al. "A probabilistic polynomial-time calculus for analysis of cryptographic protocols". In: *Electronic Notes in Theoretical Computer Science*. 2001.

[MM09] Annabelle McIver and Carroll C. Morgan. "*ums and Lovers:* Case Studies in Security, Compositionality and Refinement". In: *FM 2009, LNCS 5850*. 2009, pp. 289–304.

[MMM09] Annabelle McIver, Larissa Meinicke, and Carroll Morgan. "Security, Probability and Nearly Fair Coins in the Cryptographers' Café". In: *FM 2009, LNCS 5850*. 2009, pp. 41–71.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[NE00] Bashar Nuseibeh and Steve Easterbrook. "Requirements engineering: a roadmap". In: *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000, pp. 35–46.

[NL03] Carl Nehme and Kristina Lundqvist. "A Tool for Translating VHDL to Finite State Machines". In: *Proc. 22nd DASC*. IEEE, Oct. 2003.

[Nus97] Bashar Nuseibeh. "Ariane 5: Who Dunnit?" In: *IEEE Softw.* 14.3 (1997), pp. 15–16.

[PT97] David Pellerin and Douglas Taylor. *VHDL made easy!* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[Rot98] Charles H. Roth. *Digital Systems Design Using VHDL*. Belmont, CA, USA: Wadsworth Publ. Co., 1998.

[Sal10] David Salomon. *Elements of Computer Security*. Springer, 2010.

[Sch95] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. New York, NY, USA: John Wiley & Sons, Inc., 1995.

[Sel01] Peter Selinger. "Models for an Adversary-Centric Protocol Logic". In: *In Proceedings of the 1st Workshop on Logical Aspects of Cryptographic Protocol Verification*. Elsevier, 2001, pp. 73–88.

[Smi06] Geoffrey Smith. "Secure information flow with random assignment and encryption". In: *FMSE '06: Proceedings of the fourth ACM workshop on Formal methods in security*. Alexandria, Virginia, USA: ACM, 2006, pp. 33–44.

[Smi98] Douglas J. Smith. *HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog*. Foreword By-Zamfirescu, Alex. Doone Publications, 1998.

[SS10] Peter Stavroulakis and Mark Stamp. *Handbook of Information and Communication Security*. Springer Publishing Company, Incorporated, 2010.

[Sta10] François-Xavier Standaert. "Secure Integrated Circuits and Systems". In: ed. by Ingrid M.R. Verbauwhede. Springer, 2010. Chap. Introduction to Side-Channel Attacks, pp. 27–42.

[SV08] Aleš Smrčka and Tomáš Vojnar. "Verifying Parametrised Hardware Designs Via Counter Automata". In: *Hardware and Software, Verification and Testing*. Lecture Notes in Computer Science 4899. Heidelberg, Germany: Springer Verlag, 2008, pp. 51–68.

[Tar94] Alfred Tarski. *Introduction to Logic and to the Methodology of the Deductive Sciences*. Oxford University Press, 1994.

[TBS04] Diana Toma, Dominique Borrione, and Ghiath Al Sammane. "Combining Several Paradigms for Circuit Validation and Verification". In: *CASSIS*. 2004, pp. 229–249.

[TE01] Krishnaprasad Thirunarayan and Robert L. Ewing. "Structural Operational Semantics for a Portable Subset of Behavioral VHDL-93". In: *Formal Methods in System Design* 18.1 (2001), pp. 69–88.

[TNN05] Terkel K. Tolstrup, Flemming Nielson, and Hanne Riis Nielson. "Information Flow Analysis for VHDL". In: *PaCT*. 2005, pp. 79–98.

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis". In: *Journal of Computer Security* 4.2/3 (1996), pp. 167–188.

[Wil97] C. Williams. "Intel's Pentium chip crisis: an ethical analysis". In: *Professional Communication, IEEE Transactions on* 40.1 (Mar. 1997), pp. 13 –19.

[Woo+08] Jim Woodcock et al. "The certification of the Mondex electronic purse to ITSEC Level E6". In: *Formal Aspects of Computing* 20 (1 2008), pp. 5–19.

[Woo+09] Jim Woodcock et al. "Formal methods: Practice and experience". In: *ACM Comput. Surv.* 41.4 (2009).

[ZD04] Roberto Zunino and Pierpaolo Degano. "A Note on the Perfect Encryption Assumption in a Process Calculus". In: *FoSSaCS*. 2004, pp. 514–528.

[ZD05] Roberto Zunino and Pierpaolo Degano. "Weakening the perfect encryption assumption in Dolev-Yao adversaries". In: *Theor. Comput. Sci.* 340.1 (2005), pp. 154–178.