

Dynamic analysis

- Instrument code for testing
 - Heap memory: Purify
 - Perl tainting (information flow)
 - Java race condition checking
- Black-box testing
 - Fuzzing and penetration testing
 - Black-box web application security analysis

Purify

- Goal
 - Instrument a program to detect run-time memory errors (out-of-bounds, use-before-init) and memory leaks
- Technique
 - Works on relocatable object code
 - Link to modified malloc that provides tracking tables
 - Memory access errors: insert instruction sequence before each load and store instruction
 - Memory leaks: GC algorithm

Tainting: e.g. Perl

- Run-time checking of Perl code
 - Perl used for CGI scripts, security sensitive
 - Taint checking stops some potentially unsafe calls
- Tainted strings
 - User input, Values derived from user input
 - Except result of matching against untainted string
- Prohibited calls
 - `print $form_data{"email"} . "\n";`
 - OK since print is safe (???)
 - `system("mail " . $form_data{"email"});`
 - Flagged system call with user input as argument

Static Analysis

- Abstracts program properties and/or looks for problems
- Tools come from program analysis
 - Type inference, data flow analysis, theorem proving
- Usually on source code, can be on byte code or assembly code
- Strengths
 - Complete code coverage (in theory)
 - Potentially verify absence/report all instances of whole class of bugs
 - Catches different bugs than dynamic analysis
- Weaknesses
 - High false positive rates
 - Many properties cannot be easily modeled
 - Difficult to build
 - Almost never have all source code in real systems (operating system, shared libraries, dynamic loading, etc.)

Two Types of Static Analysis

- (Rather) Simple code analysis.
 - Look for known code issues: e.g., unsafe string functions `strncpy()`, `sprintf()`, `gets()`
 - Look for unsafe functions in your source base
 - Look for recurring problem code (problematic interfaces, copy/paste of bad code, etc.)
- Deeper analysis
 - Requires complex code parsing and computations
 - Some are implemented in tools like coverity, fortify, visual studio ...
 - Otherwise must be developed on top of parser like LLVM
 - In the case of disassemblers, the security expert is the last part of the static analyzer ...

Static analysis: Soundness, Completeness

Property	Definition
Soundness	<p>“Sound for reporting correctness”</p> <p>Analysis says no bugs \rightarrow No bugs or equivalently</p> <p>There is a bug \rightarrow Analysis finds a bug</p>
Completeness	<p>“Complete for reporting correctness”</p> <p>No bugs \rightarrow Analysis says no bugs</p>

Recall: $A \rightarrow B$ is equivalent to $(\neg B) \rightarrow (\neg A)$

Complete

Incomplete

Sound

Reports all errors
Reports no false alarms

Undecidable

Reports all errors
May report false alarms

Decidable

Unsound

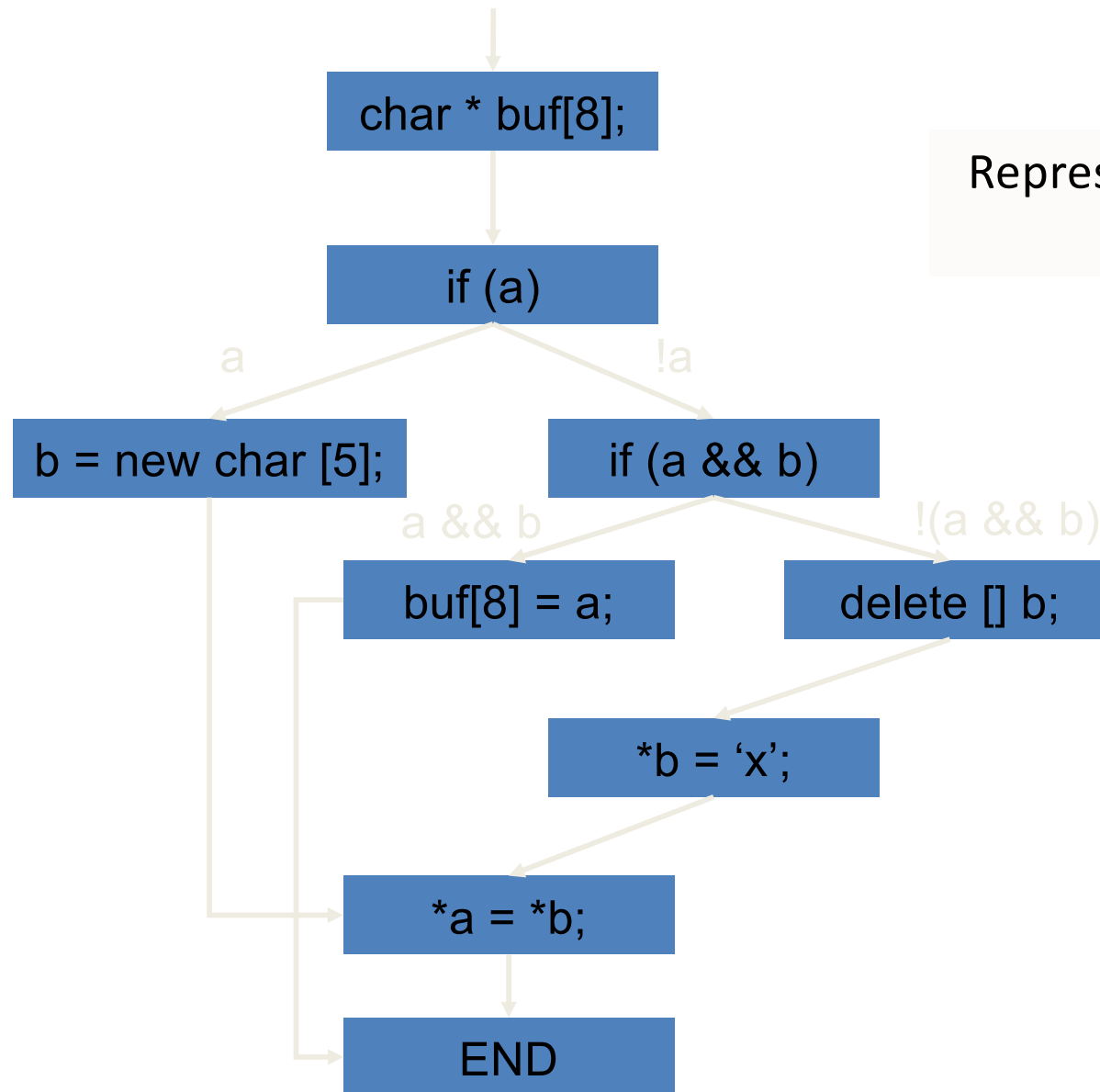
May not report all errors
Reports no false alarms

Decidable

May not report all errors
May report false alarms

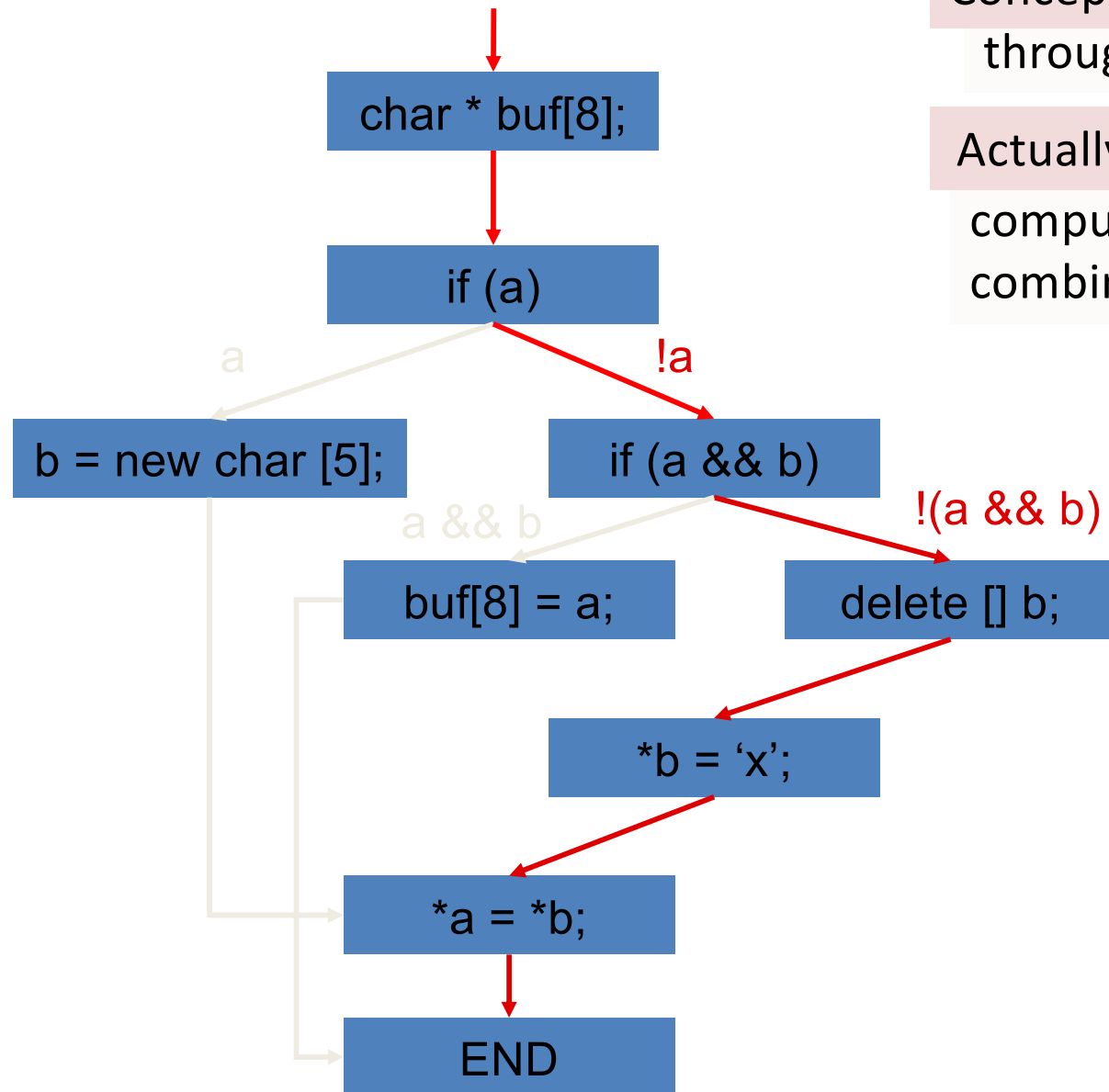
Decidable

Control Flow Graph



Represent logical structure of code
in graph form

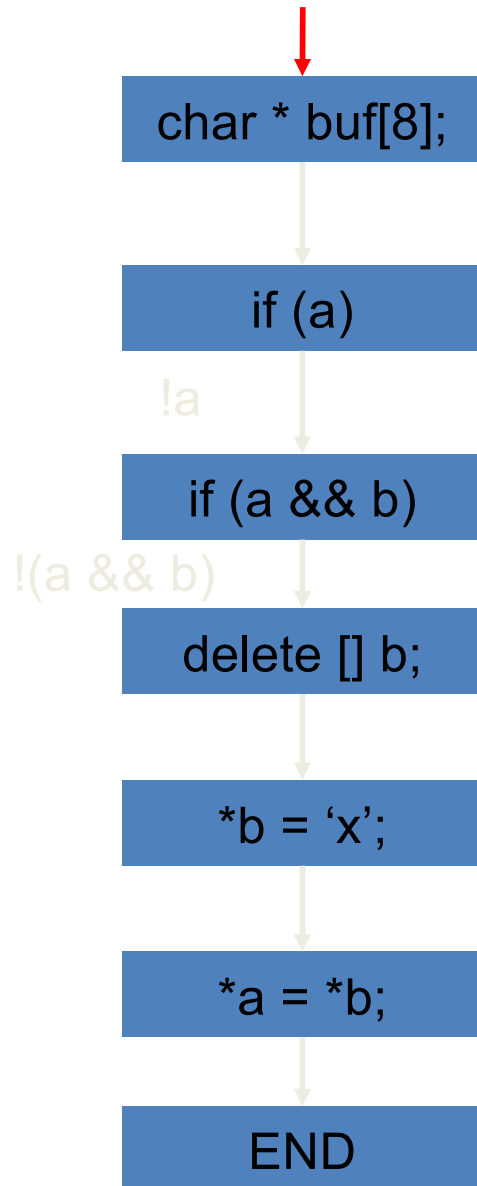
Path Traversal



Conceptually Analyze each path through control graph separately

Actually Perform some checking computation once per node; combine paths at merge nodes

Apply Checking



See how three checkers are run for this path

Null pointers

Use after free

Array overrun

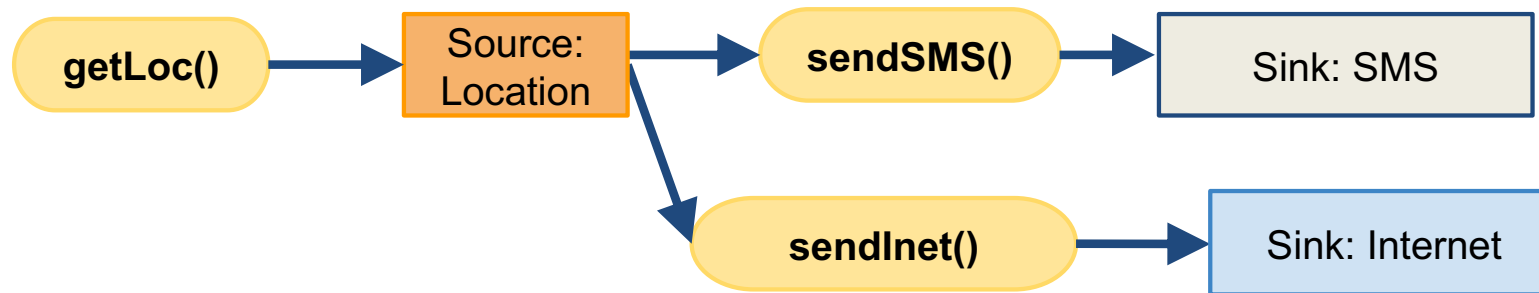
Checker

- Defined by a state diagram, with state transitions and error states

Run Checker

- Assign initial state to each program var
- State at program point depends on state at previous point, program actions
- Emit error if error state reached

Data Flow Analysis



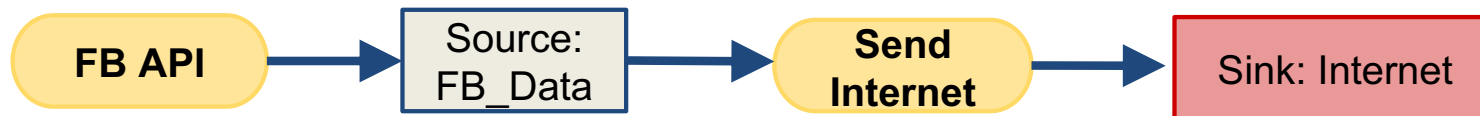
- Source-to-sink flows
 - Sources: Location, Calendar, Contacts, Device ID etc.
 - Sinks: Internet, SMS, Disk, etc.



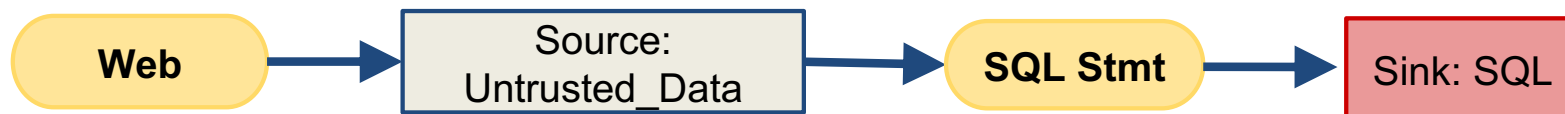
Applications of Data Flow Analysis

- Vulnerability Discovery
- Malware/Greyware Analysis
 - Data flow summaries enable enterprise-specific policies

- API Misuse and Data Theft Detection



- Automatic Generation of App Privacy Policies
 - Avoid liability, protect consumer privacy

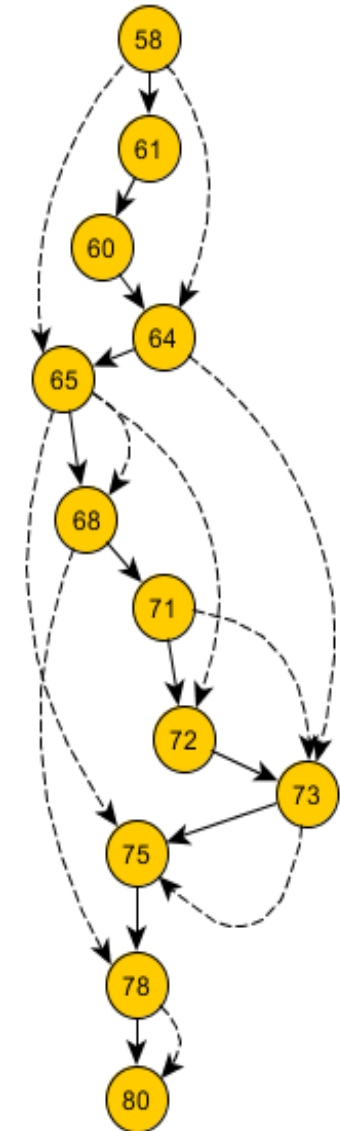


Privacy Policy
This app collects your:
Contacts
Phone Number
Address

Program Dependence Graph (PDG)

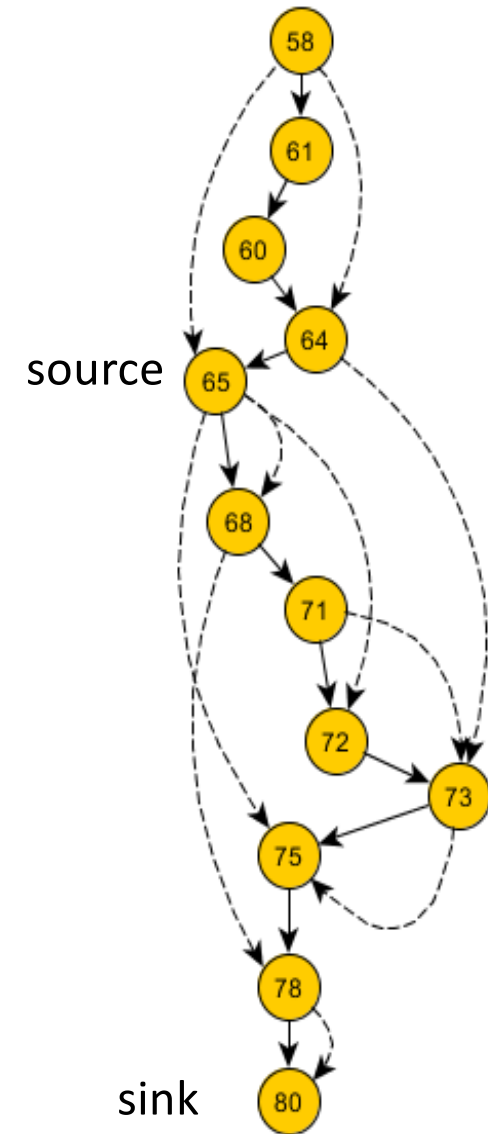
- ❑ Control Dependences
- ❑ Explicit + Implicit Data Dependences
- ❑ Properties:
 - ❑ Path-sensitive
 - ❑ Context-Sensitive
 - ❑ Object-Sensitive

→ Control dependence
- - - - -> Data dependence



JOANA IFC tool

- Intended for Information Flow Analysis
- Annotations: SINK / SOURCE
- Non-Interference: Security Levels (HIGH / LOW)



Benefits: automated code analysis

- Examples
 - C language: FindBugs, Fortify, Coverity, MS tools (commercial), KLEE (academic)
 - Java language: JOANA, FindBugs, Soot, PMD (academic)
- Objectives
 - Capture recommended practices, known to experts, in tool available to all
 - Also Capture information flows, non-interference

Pentesting: Reverse Engineering

- A special form of static analysis
 - Expert user required
 - Used to study how a program works and to find vulnerabilities that can be exploited in closed source software
 - Find backdoors (insider attack ...)
 - Also used to study malware, and how it exploits software/systems
- Reversing binary into:
 - Assembly form (given file format)
 - Executable and Linkable (ELF) Format – Linux
 - Portable Executable (PE) Format - Windows
 - Source form (less common, e.g. Java disassembly from bytecode)

Pentesting: Reverse Engineering

- Requires a tool for performing disassembly:
 - IDA (Pro): world famous disassembly tool
 - Ghidra: disassembler authored by NSA
 - Objdump (Linux/Mac)
- May also require using a debugger and an assembler for modifying the assembly code:
 - Ollydbg (Windows)
 - GDB (Linux/Mac)
 - NASM

Disassemblers vs. Debuggers

- Debuggers are designed to run code
 - They can disassemble code (e.g. gdb « disas »)
 - Single functions
 - Based on the instruction pointer
 - Generally don't do batch disassembly
- Disassemblers don't run the code
 - Output is a disassembly listing
 - Often quite to extremely large output
 - Hard to navigate
 - Harder to understand than source code!
 - Advanced tools also provide a control-flow graph view with an intuitive navigation
 - And many other tools/functionalities (renaming, reformatting, introducing comments, hexdump, code structure analysis, library analysis)

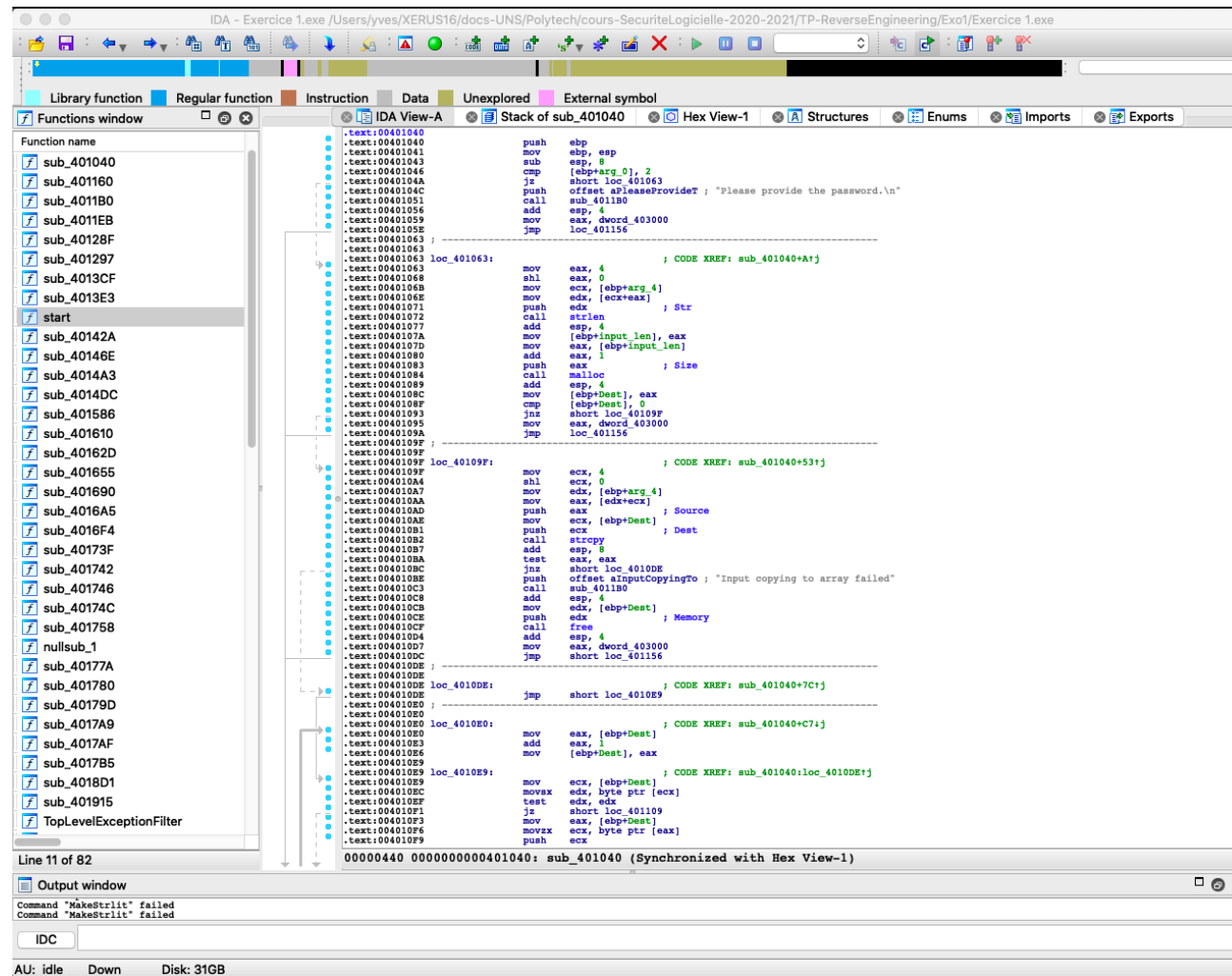
IDA Operation

- Load your binary of interest (e.g. drag&drop)
- IDA analyzes and characterizes each byte of the binary file
 - Builds a database (see files under your directory)
 - Further manipulations will involve database interactions (reads for navigation, updates for renaming, etc.)
- Performs a detailed analysis of the code:
 - Recognizes function boundaries and library calls (and even names for known library calls)
 - Recognizes data types for known library calls
 - Recognizes string constants
- You can navigate code and graph (double-click)
 - Web browser like history (and ESC = back)
- You can modify content as you recognize data and functions (change names)
 - Beware: many hotkeys, and there is no undo!

IDA: Disassembly listing

- main window
 - initially positioned at entry point
 - Entry point = generally not main, but instead start or _start
- Can switch with graph view using space bar
- Also contains jumps (conditional or not) in the margin at the left of the assembly code dump
 - Useful for identifying branching and looping constructs
 - Conditional jumps – dashed
 - Unconditional jumps – solid
 - Backward jumps – heavier line

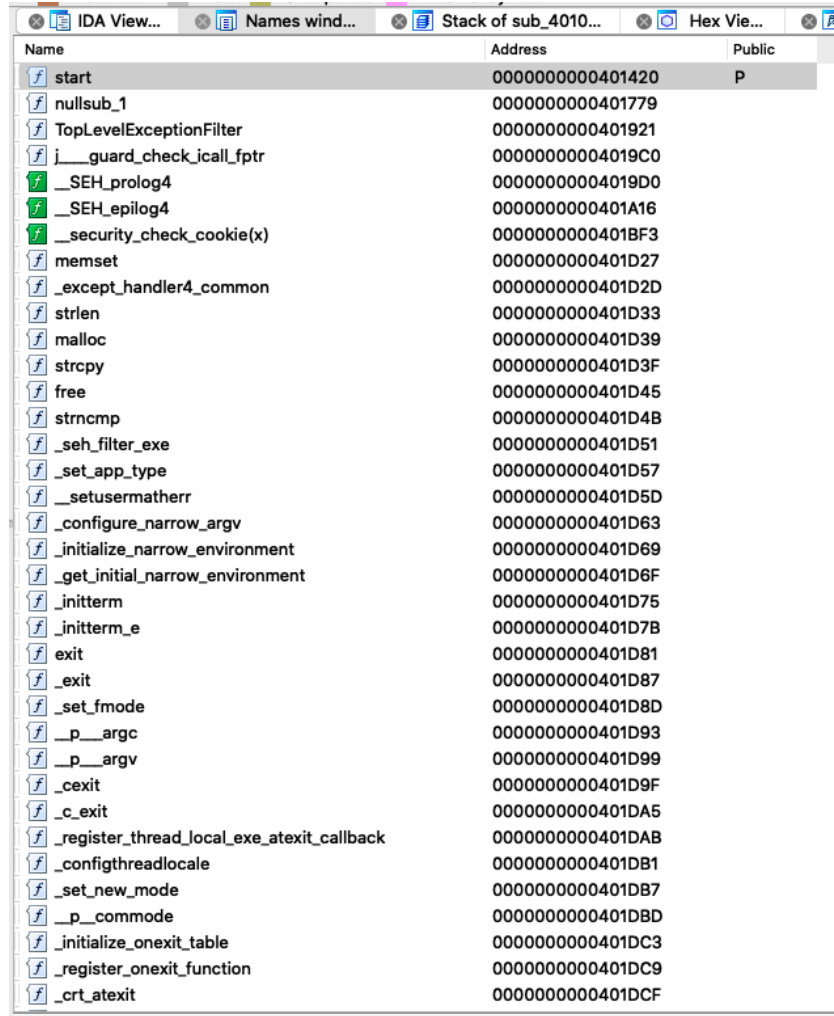
IDA: disassembly listing



IDA: Names window

- Based on imports, exports, and some analysis
 - F is a function
 - L is a library function
 - C is code/instruction
 - A is a string
 - D is defined data
 - I is an imported function (dynamically linked)

IDA: Names window



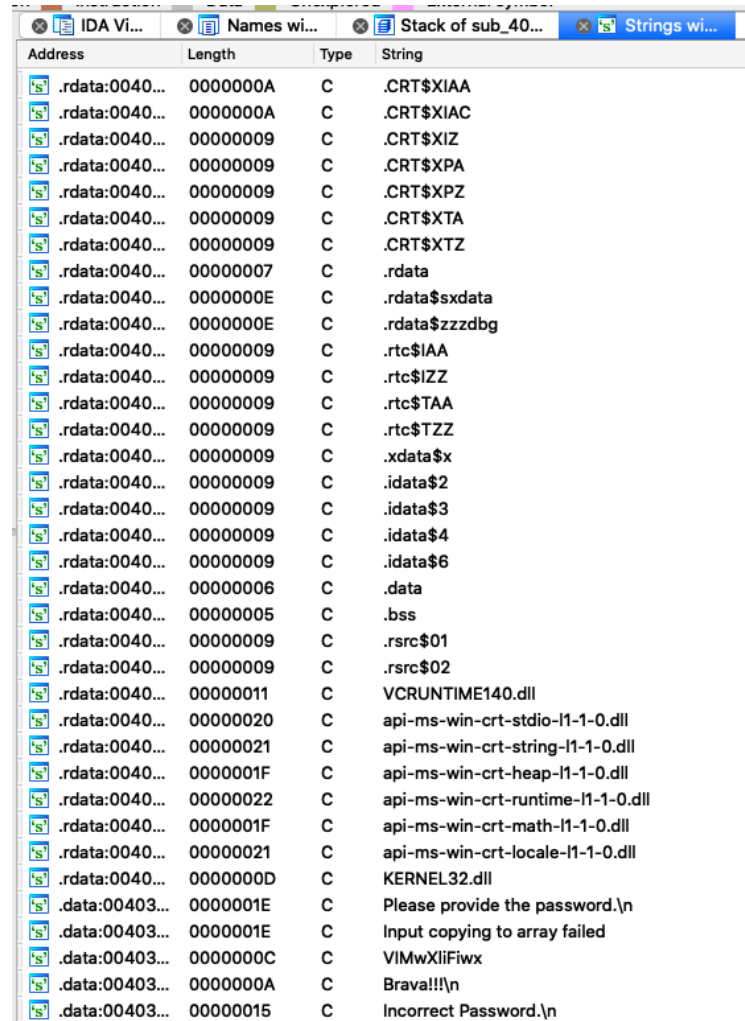
The screenshot shows the IDA Names window with a list of symbols. The window has tabs for 'IDA View...', 'Names wind...', 'Stack of sub_4010...', 'Hex Vie...', and 'A'. The list includes various system and application-specific functions and symbols, each with a small icon (function, data, or public) and a 'Public' column.

Name	Address	Public
start	000000000401420	P
nullsub_1	000000000401779	
TopLevelExceptionFilter	000000000401921	
j__guard_check_icall_fptr	0000000004019C0	
__SEH_prolog4	0000000004019D0	
__SEH_epilog4	000000000401A16	
__security_check_cookie(x)	000000000401BF3	
memset	000000000401D27	
_except_handler4_common	000000000401D2D	
strlen	000000000401D33	
malloc	000000000401D39	
strcpy	000000000401D3F	
free	000000000401D45	
strncmp	000000000401D4B	
_seh_filter_exe	000000000401D51	
_set_app_type	000000000401D57	
_setusermatherr	000000000401D5D	
_configure_narrow_argv	000000000401D63	
_initialize_narrow_environment	000000000401D69	
_get_initial_narrow_environment	000000000401D6F	
_initterm	000000000401D75	
_initterm_e	000000000401D7B	
exit	000000000401D81	
_exit	000000000401D87	
_set_fmode	000000000401D8D	
_p_argc	000000000401D93	
_p_argv	000000000401D99	
_cexit	000000000401D9F	
_c_exit	000000000401DA5	
_register_thread_local_exe_atexit_callback	000000000401DAB	
_configthreadlocale	000000000401DB1	
_set_new_mode	000000000401DB7	
_p_commode	000000000401DBD	
_initialize_onexit_table	000000000401DC3	
_register_onexit_function	000000000401DC9	
_crt_atexit	000000000401DCF	

IDA: String window

- Complete listing of strings embedded within the program
- Configurable
 - Right click in Strings window and choose setup
 - Can change minimum length or style of string to search for (IDA rescans for strings if you change settings)
- Excellent tool for locating interesting inputs/outputs or text data
 - E.g., detect success conditions in the code

IDA: String window

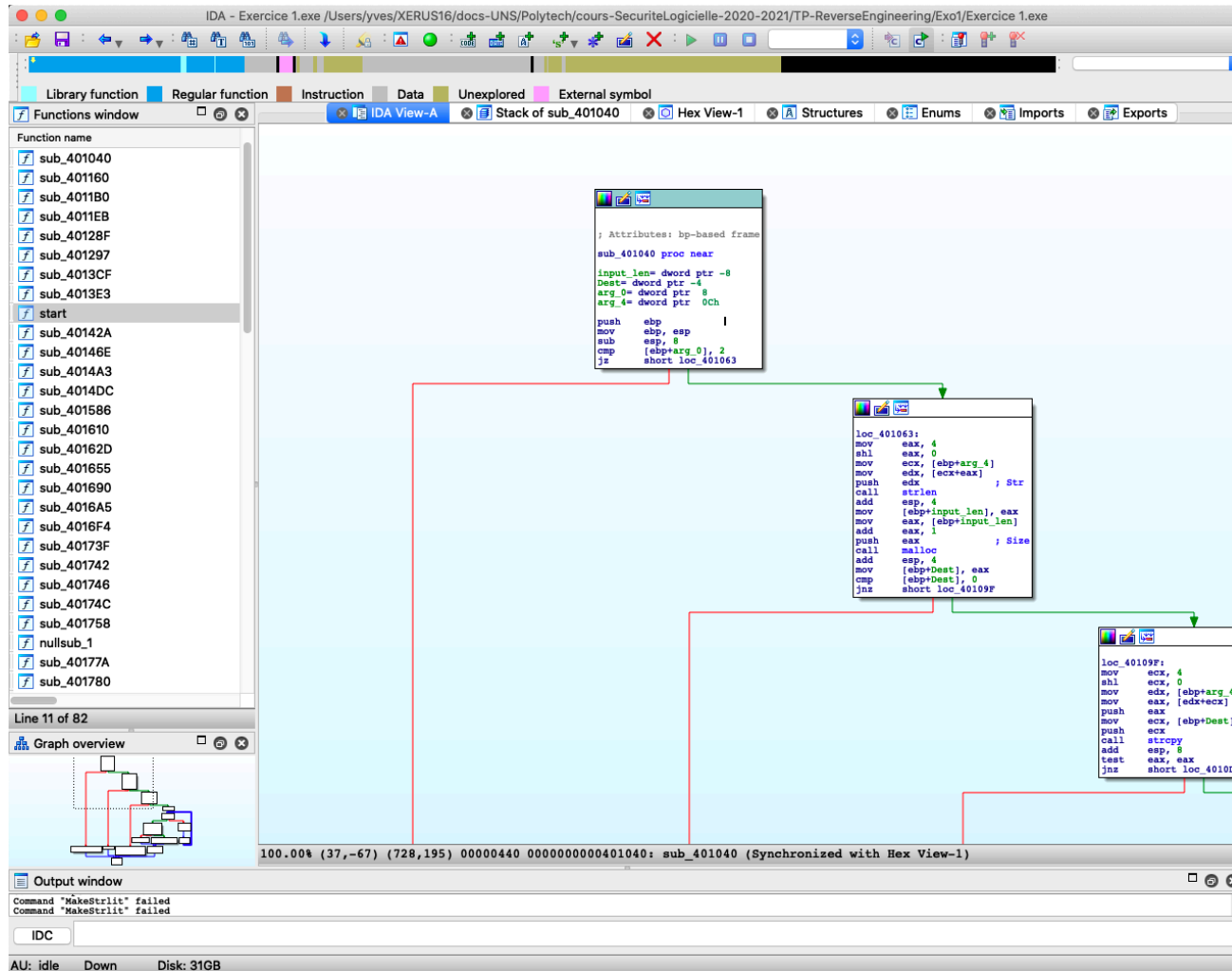


Address	Length	Type	String
.rdata:0040...	0000000A	C	.CRT\$XIAA
.rdata:0040...	0000000A	C	.CRT\$XIAC
.rdata:0040...	00000009	C	.CRT\$XIZ
.rdata:0040...	00000009	C	.CRT\$XPA
.rdata:0040...	00000009	C	.CRT\$XPZ
.rdata:0040...	00000009	C	.CRT\$XTA
.rdata:0040...	00000009	C	.CRT\$XTZ
.rdata:0040...	00000007	C	.rdata
.rdata:0040...	0000000E	C	.rdata\$sxdata
.rdata:0040...	0000000E	C	.rdata\$zzzdbg
.rdata:0040...	00000009	C	.rtc\$IAA
.rdata:0040...	00000009	C	.rtc\$IZZ
.rdata:0040...	00000009	C	.rtc\$TAA
.rdata:0040...	00000009	C	.rtc\$TZZ
.rdata:0040...	00000009	C	.xdata\$x
.rdata:0040...	00000009	C	.idata\$2
.rdata:0040...	00000009	C	.idata\$3
.rdata:0040...	00000009	C	.idata\$4
.rdata:0040...	00000009	C	.idata\$6
.rdata:0040...	00000006	C	.data
.rdata:0040...	00000005	C	.bss
.rdata:0040...	00000009	C	.rsrc\$01
.rdata:0040...	00000009	C	.rsrc\$02
.rdata:0040...	00000011	C	VCRUNTIME140.dll
.rdata:0040...	00000020	C	api-ms-win-crt-stdio-l1-1-0.dll
.rdata:0040...	00000021	C	api-ms-win-crt-string-l1-1-0.dll
.rdata:0040...	0000001F	C	api-ms-win-crt-heap-l1-1-0.dll
.rdata:0040...	00000022	C	api-ms-win-crt-runtime-l1-1-0.dll
.rdata:0040...	0000001F	C	api-ms-win-crt-math-l1-1-0.dll
.rdata:0040...	00000021	C	api-ms-win-crt-locale-l1-1-0.dll
.rdata:0040...	0000000D	C	KERNEL32.dll
.data:00403...	0000001E	C	Please provide the password.\n
.data:00403...	0000001E	C	Input copying to array failed
.data:00403...	0000000C	C	VIMwXliFiwx
.data:00403...	0000000A	C	Brava!!!\n
.data:00403...	00000015	C	Incorrect Password.\n

IDA: graphs

- Many graphs can be generated
- Function flow charts
 - Unconditional jumps – blue line
 - Conditional jump if true – green line
 - Conditional jump if false – red line
 - Move your mouse on top of graph to get further info
- Function call tree (forest) for a program
- All crossrefs from a function (« who do I call? »)
- All crossrefs to a function (« who calls me? »)

IDA: function flow chart



x86 Assembly basics: instructions

- Memory manipulation:
 - Mov <dst>, <src> - addresses can be described by « [address value] »
 - Push/Pop <registry>
 - Xcgh <registry 1>, <registry 2>
- Arithmetic operators:
 - Add/Dec/Mul/Div <registry>, <operand2>
 - Inc/Dec <registry>
 - Neg <registry> - two's complement
- Bit-level manipulation:
 - And/Or/Xor <registry1>, <registry2>
 - Not <registry>
 - Shl <registry>, <added_bit>
 - Shr <registry>
 - Rol/Ror <registry>

x86 Assembly basics: instructions

- Tests
 - Cmp <registry 1>, <registry 2>
- Jumps
 - Jmp <code location>
 - Je <code location> (if previous test is equal)
 - Jne <code location> (if previous test not equal)
 - Jz <code location> (if previous operation is zero)
 - Jnz <code location> (if previous operation not zero)
- Subroutine calls
 - Call <code location>
 - Ret

Security Requirements Engineering: Two Approaches

■ Security By Certification

- Best Practices, Security Guidelines
- Information Flow Control
- Static & Dynamic Analysis

■ Security By Design

- Security Objectives
- Threat Analysis

Existing methodologies

- UML Proposals

- SecureUML, Model-Driven Architecture [Basin et al.]
- UMLsec [Juriens]
- Abuse Cases [McDermott & Fox]
- Misuse Cases [Sindre & Opdhal]

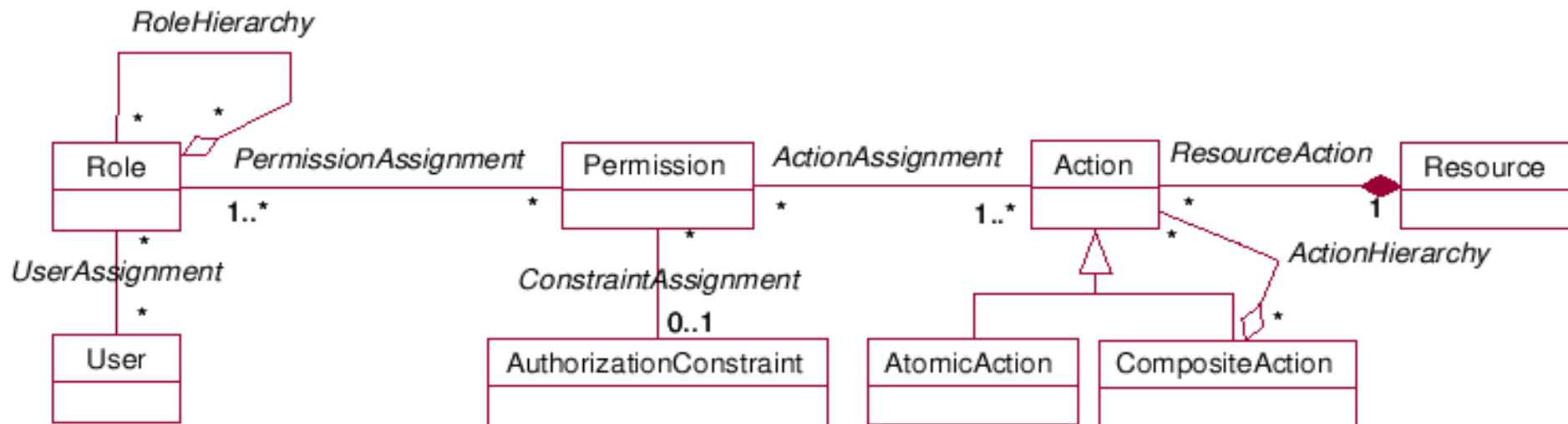
- Early Requirements Proposals

- Anti-requirements [van Lamsweerde et al., Crook et al.],
- Problem-Frames, Abuse Frames [Hall et al., Lin et al]
- Security Patterns [Giorgini & Mouratidis]
- Privacy Modelling [Liu et al., Anton et al,]

SecureUML

- D. Basin, J. Doser, and T. Lodderstedt, 2003
- They provide support for specifying access control policies
- The concepts of RBAC are represented as metamodel types
 - User, Role, Permission, Actions are types
 - UserAssignment, PermissionAssignment, RoleHierarchy are relations
 - AuthorizationConstraint is a predicate attached to a permission by the association ConstraintAssignment
 - Authorization constraints expressed in first-order logic
 - Used to establish the validity of the permission

SecureUML Metamodel



D. Basin, J. Doser, and T. Lodderstedt. Model driven security for process-oriented systems. In Proc. of SACMAT '03, pages 100-109. ACM Press, 2003.

SecureUML Semantics

- An access control configuration is an assignment of users and permissions to role
- SecureUML makes access control decisions based on the access control configuration and on the validity of authorization constraints in a certain system state
 - Verify if a user is allowed to perform actions in the system state at a certain time with respect to an access control configuration

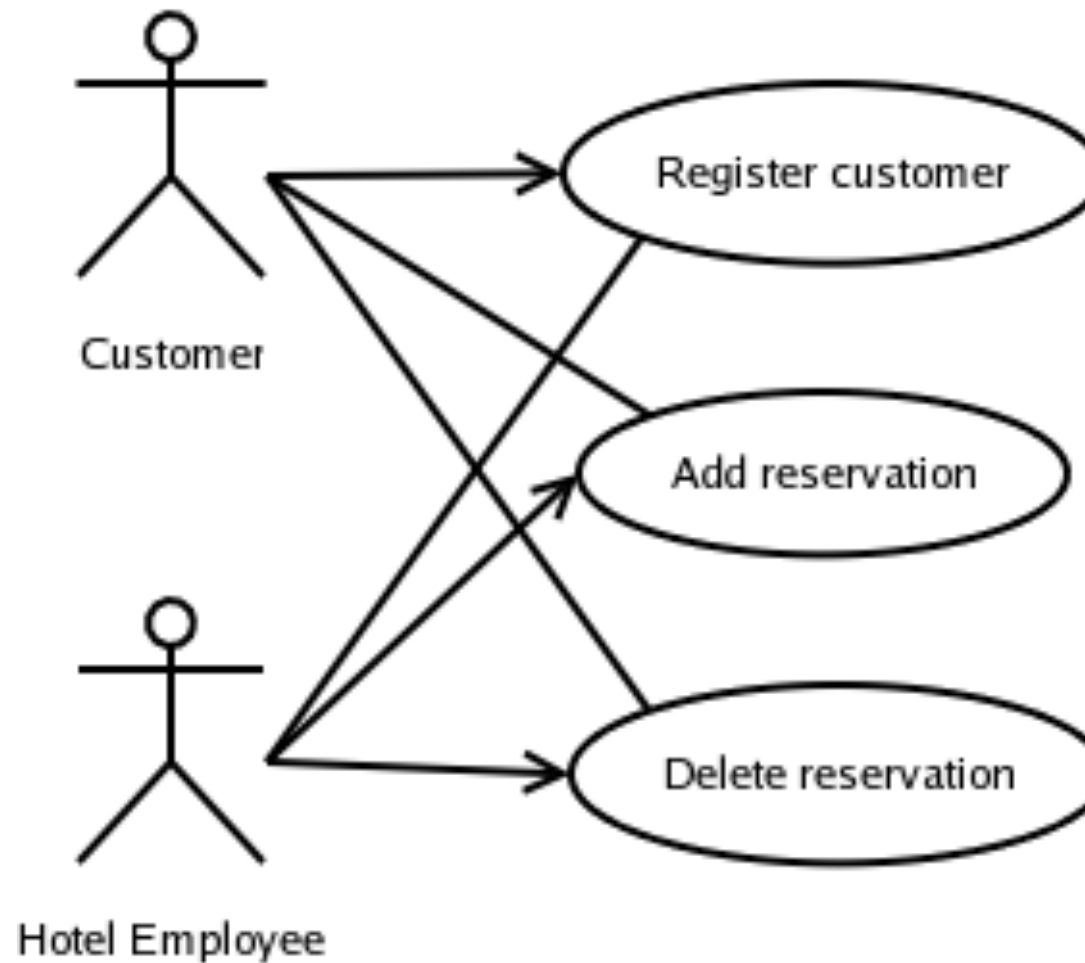
Limits of SecureUML

- NO analysis of security requirements within the organizational environment in which the software system will operate
- Need to know conflicting roles a priori
 - NO detection of conflicts from the requirements model of the system

Use Case Diagram

- Build a first sketch model of a system
- Characterize a way of using a system
- Offer a notation for describe the functionality of a system
 - Actors: an abstraction of an external agent that interact with the system
 - Use cases: specification of a type of interaction between a system and agents
 - Association lines: connect agents with the use cases in which they participate

Reservation System



KAOS

- A research project
- Used to formalize goals into requirements
- Derive a description of a system's behavior
- Analyze system structure through acquiring and formalizing functional and non-functional requirements
- Supported by GRAIL tool

KAOS concepts

- Agents
 - active component of the system
 - play some role
- Goals
 - prescriptive statements of intent about the system
 - functional goals: service to be provided
 - non-functional goals: quality of service
- Domain properties
 - descriptive statements about the environment (e.g., physical laws, norms)

Goals

- Organized in terms of AND/OR hierarchies
 - Goal refinement used to construct a refinement-abstraction hierarchy
 - High level goals are strategic
 - Coarse grained with many agents
 - Low level goals are technical
 - Fine grained involving less agents
- Requirement: terminal goal for one agent

Obstacles

- Identify goal violation scenarios
- An obstacle to some goal is a condition whose satisfaction may prevent the goal from being achieved
- An obstacle O is said to obstruct a goal G in domain Dom iff

$\{O, Dom\} \models \neg G$ Obstruction

$Dom \not\models \neg O$ Domain consistency

Obstacle Analysis

- Obstacle analysis consists in taking a pessimistic view of goals
- Identify as many ways of breaking goals as possible in order to resolve each of such situations
- Formal techniques for generation and logical (AND/OR) refinement of obstacles are available
- Obstacles need to be resolved once they have been generated
 - Resolution techniques: goal substitution, agent substitution, goal weakening, goal restoration, obstacle prevention and obstacle mitigation
- Obstacle analysis is a recursive process
 - it may produce new goals for which new obstacles may be generated and resolved

Security Goals

- Considered a meta-class
- High level of abstraction
 - Confidentiality
 - Integrity
 - Availability
 - Privacy
 - Authenticity
 - Non-repudiation
- Each security goal has to be instantiated into application-specific security goal

Confidentiality

Goal Confidentiality

Goal *Avoid*[SensitiveInfoKnownByUnauthorizedAgent]

FormalSpec \forall ag: Agent, ob: Object

\neg Authorized(ag, ob.Info) \Rightarrow \neg Knows(ob.Info)

Goal *Avoid*[PaymentMediumKnownBy3rdParty]

FormalSpec \forall p: Agent, acc: Account

\neg (Owns(p, acc) \vee Manages(p, acc))

\Rightarrow \neg (Knows(acc.Acc#) \wedge Knows(acc.PIN))

*If agent **p** is not the owner of account **acc** and he should not manage it, he does not know **number** and **PIN** of the account*

Obstacle Analysis

- Negating a goal

$\forall p: \text{Agent}, \text{acc}: \text{Account}$

(NG) $\neg (\text{Owns}(p, \text{acc}) \vee \text{Manages}(p, \text{acc}))$
 $\wedge (\text{Knows}(\text{acc}, \text{Acc\#}) \wedge \text{Knows}(\text{acc}, \text{PIN}))$

- Assuming domain theory contains the following properties:

(D1) $\forall p: \text{Agent}, \text{acc}: \text{Account} \text{ Owns}(p, \text{acc}) \wedge \text{Knows}(p, \text{name}) \Rightarrow \text{Knows}(\text{acc}, \text{Acc\#})$

(D2) $\forall \text{acc}: \text{Account} \text{ Knows}(\text{acc}, \text{Acc\#}) \Rightarrow \text{Knows}(\text{acc}, \text{PIN})$

- We can formally derive the following potential obstacle:

(O) $\forall p: \text{Agent}, \text{acc}: \text{Account}$

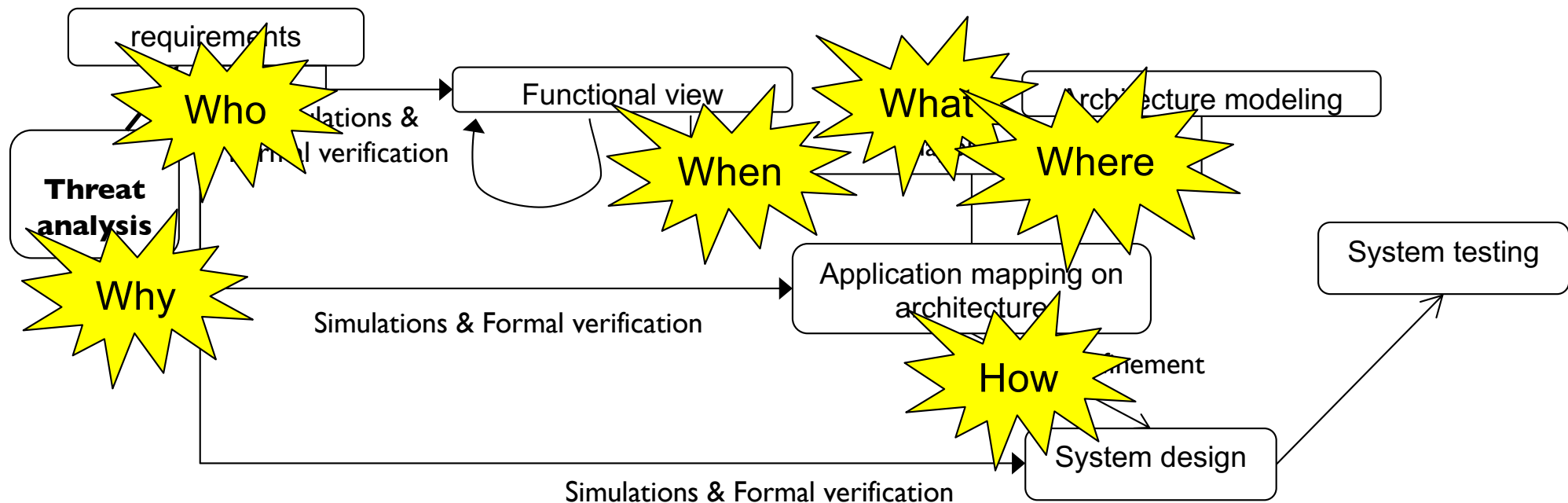
$\neg (\text{Owns}(p, \text{acc}) \vee \text{Manages}(p, \text{acc})) \wedge \text{Knows}(p, \text{name})$

Security Requirements Beyond pure IT Security

- SE key issues:
 - Incremental design ?
 - Continuous usage of requirements throughout SDLC?
 - Specifying security requirements rather than security mechanisms?
- System key issues:
 - Capturing System perspective (HW / SW partitioning)?
 - Capturing environmental constraints (e.g., real-time constraints)?
 - Addressing functional AND safety requirements?
- Model Driven Engineering as a Holistic Approach: SysML-Sec
 - Deployment of software components in architecture
 - Architecture = CPUs + memories + buses + software
 - Bring together system engineers & security experts

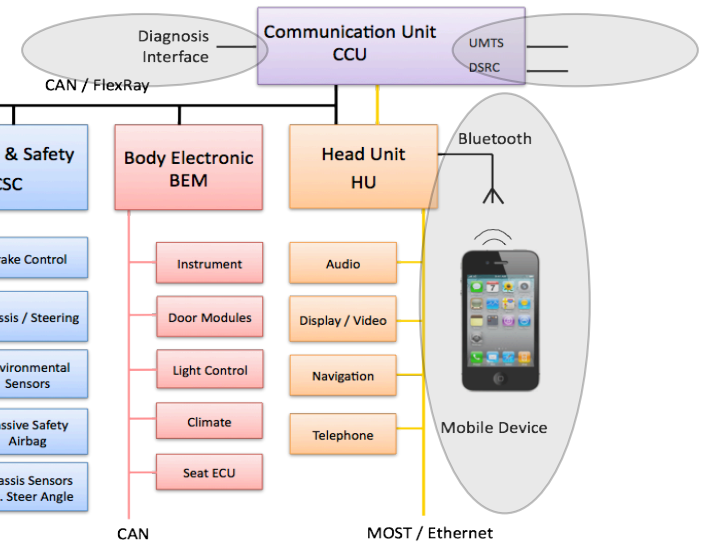
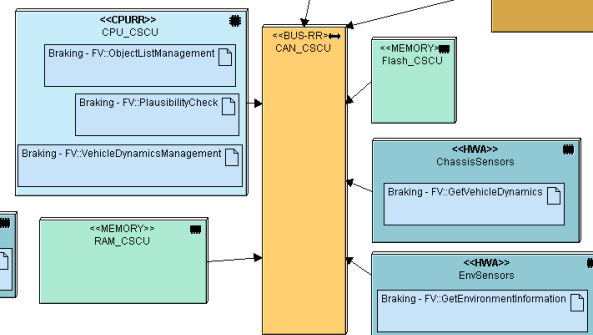
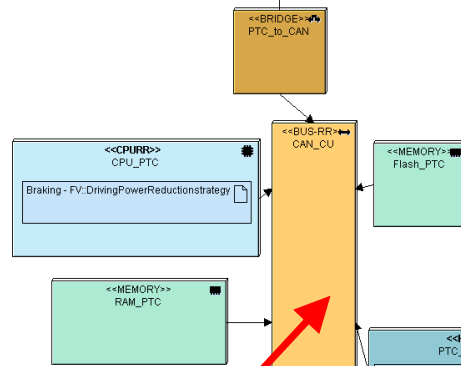
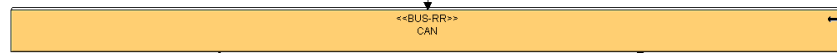
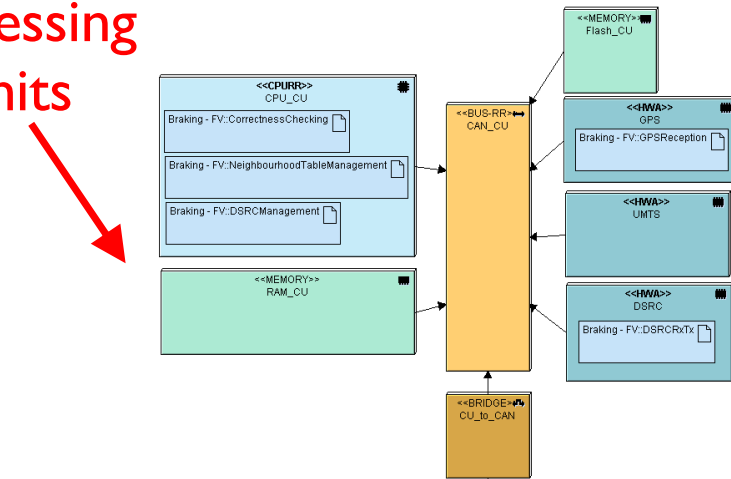
SysML-Sec : The Y-Chart Revisited

- Security concerns:
 - *What*: **assets** to be protected
 - *When*: operation **sequences** in functions involving those assets
 - *Where*: **architecture mapping** of functions involving those assets
 - *Why*: **attacks / threats** envisioned that motivate security countermeasures
 - *Who*: stakeholders + attackers & capabilities (**risk** analysis)
 - *How*: **security objectives** due to architecture (e.g., network topology, process isolation, etc.)



Architectural Mapping Model (Deployment Diagram for Actual Topology)

Processing
Units

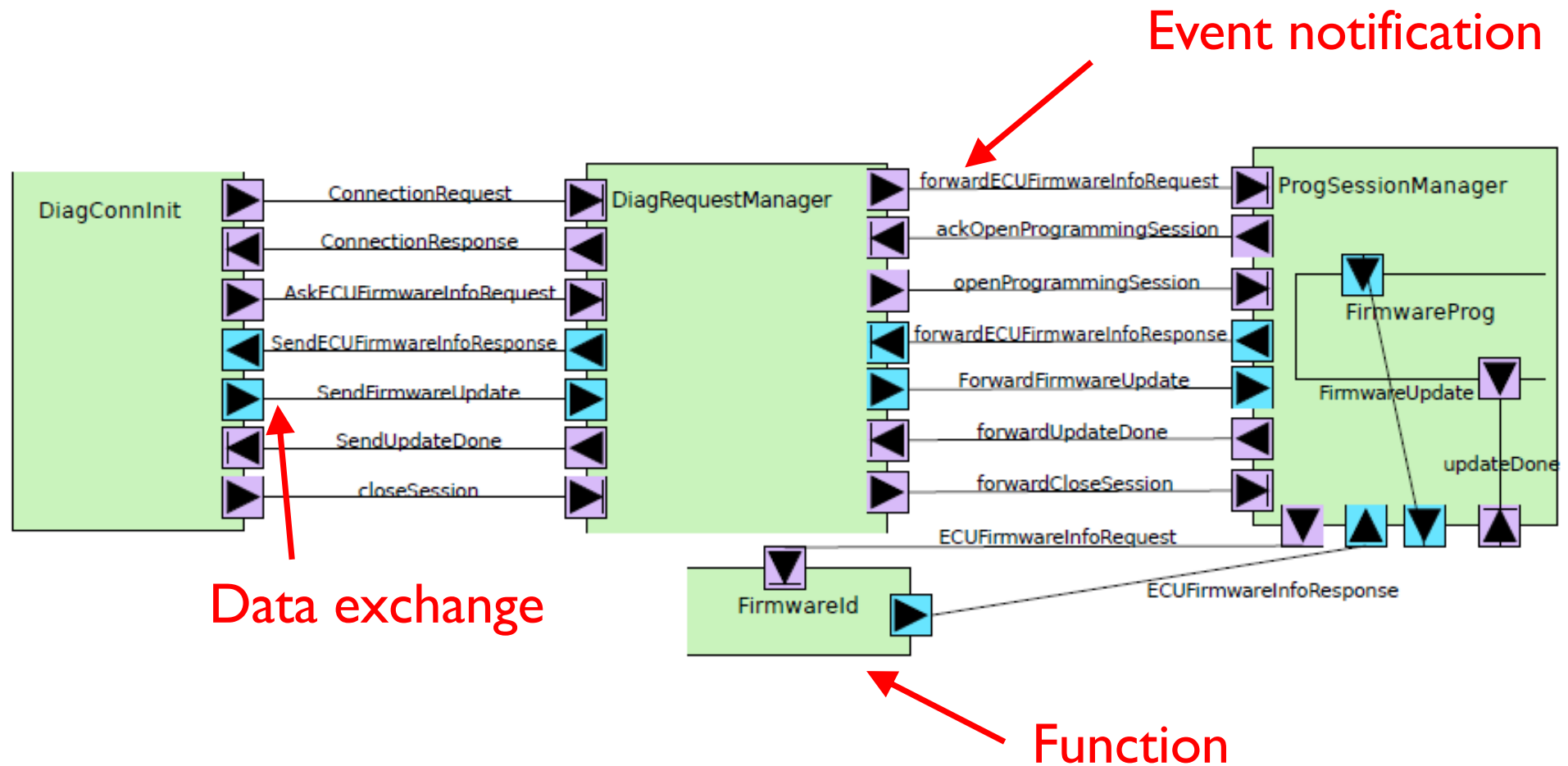


Function mapping

Event/dataflow mapping

Functional View: Specifying Information Flows

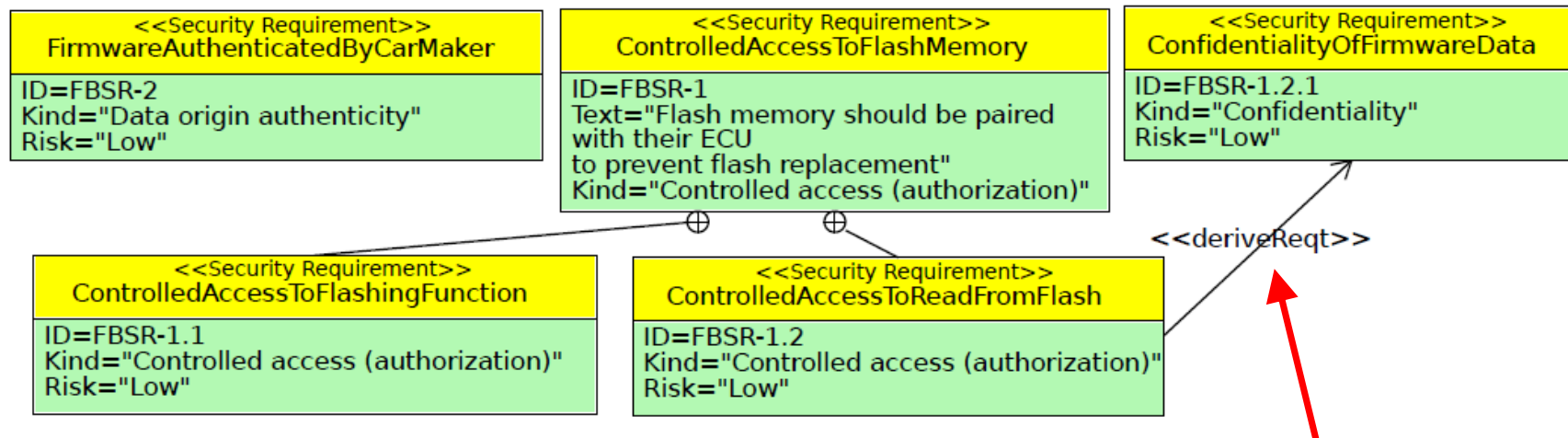
Internal Block Diagram



Security Properties and Types of Countermeasures

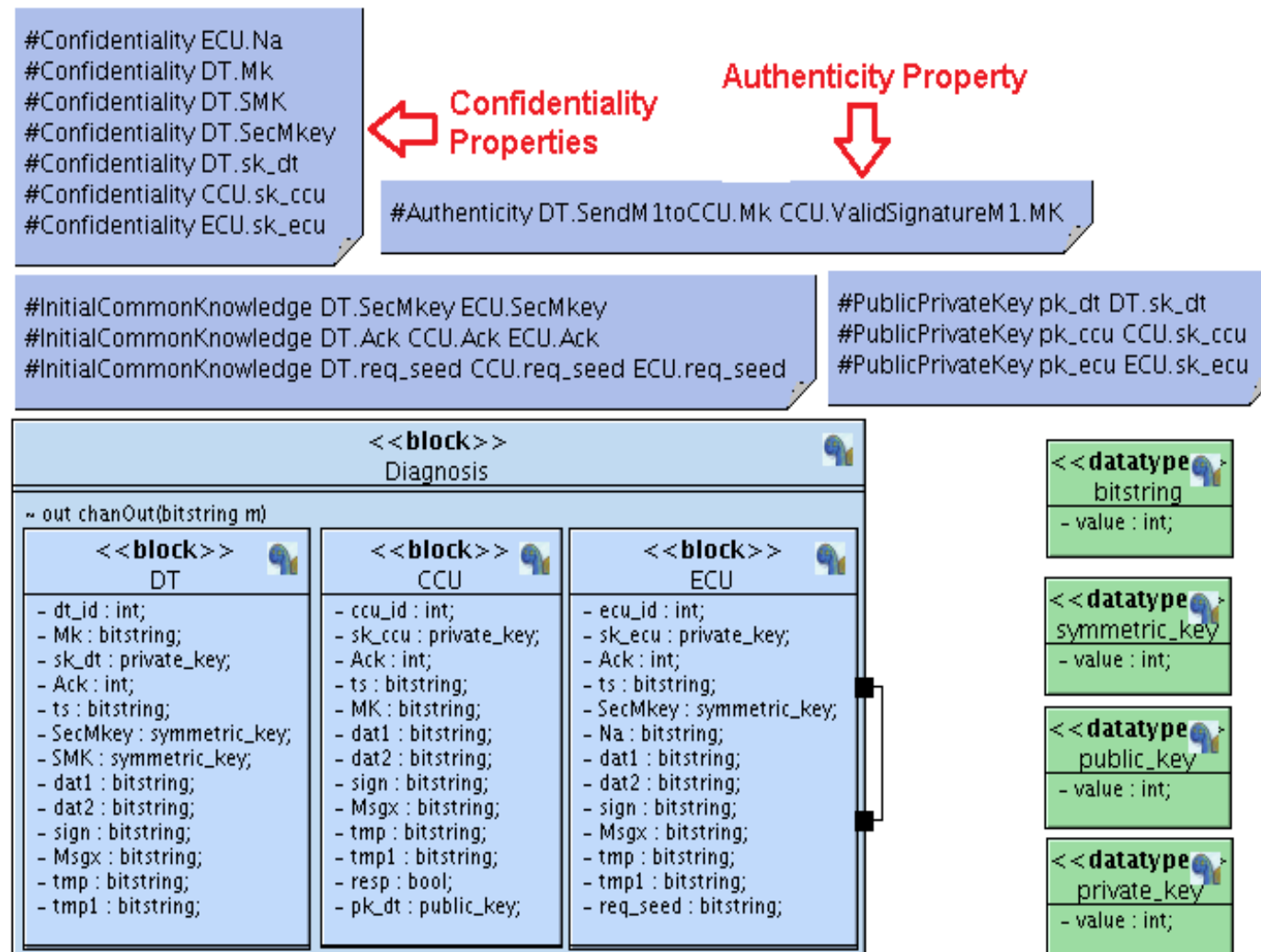
Requirements Diagram

Security Properties: e.g., Confidentiality, Authenticity, Integrity, Freshness, Availability... applicable to some asset (HW/SW or more notably data or information flow)

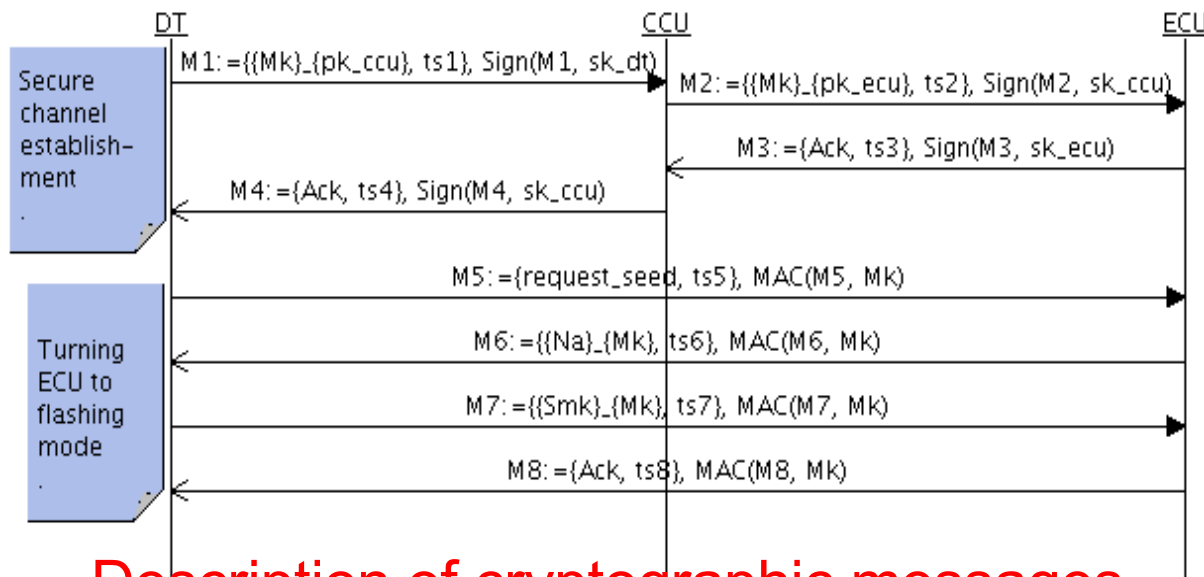


Trace refinements and dependencies

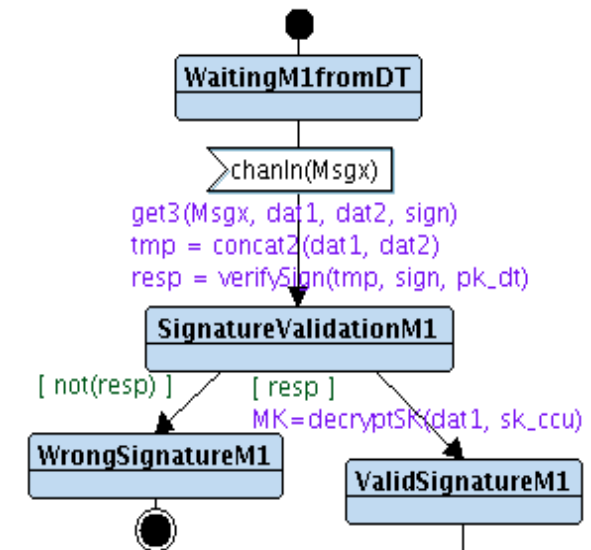
SysML Block Definition Diagram: cryptographic protocol environment support



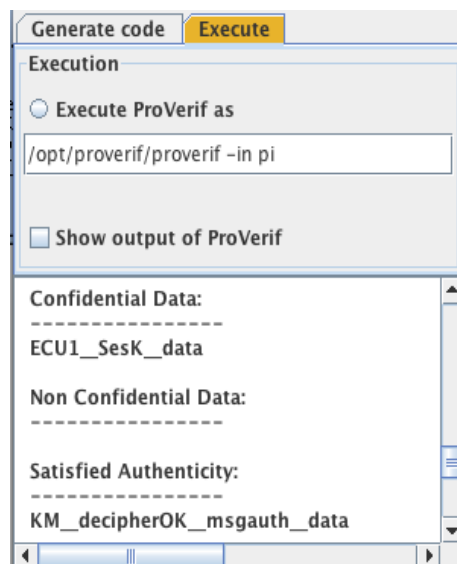
Cryptographic Protocol Messages: Content and Handling



Description of cryptographic messages

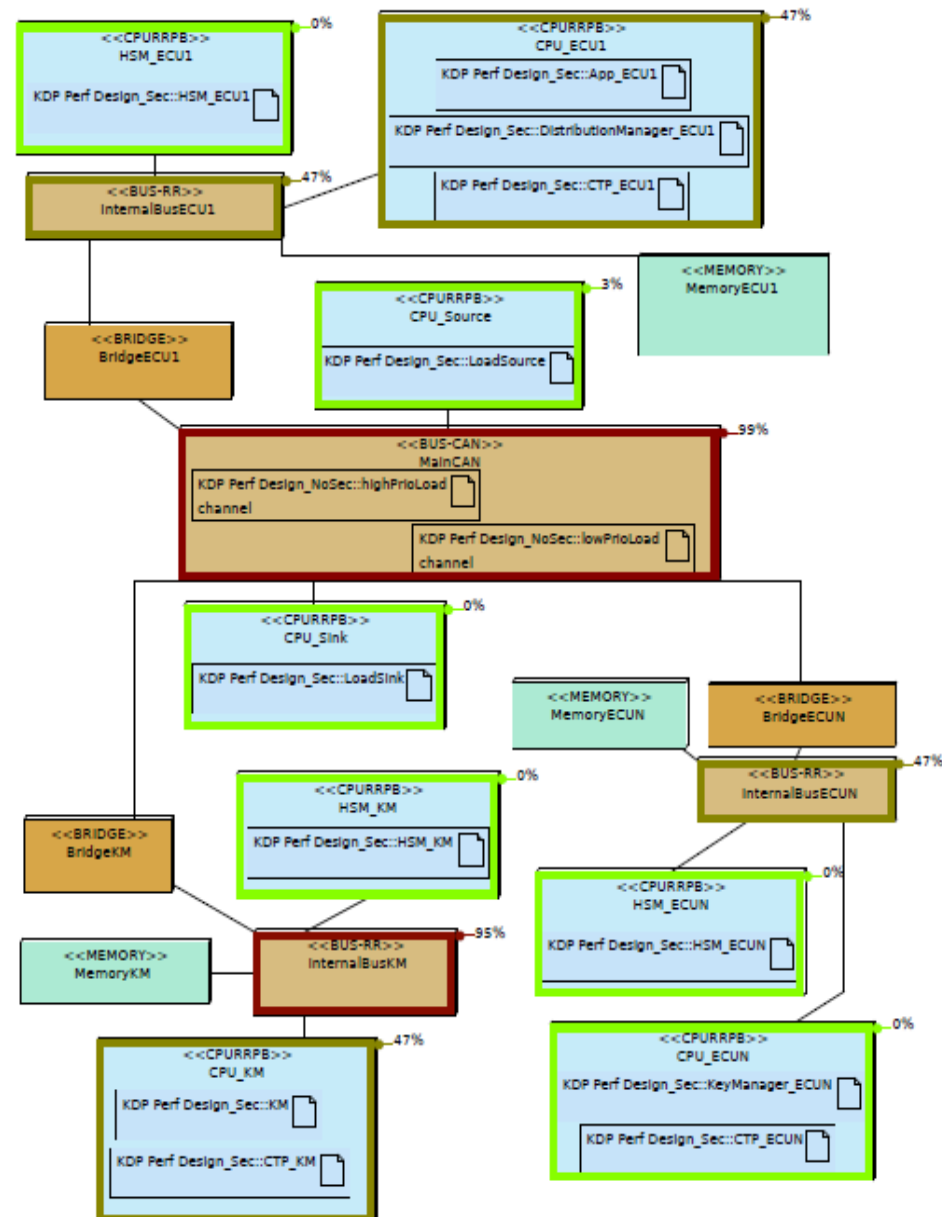


Handling of M1 at CCU



Formal verification in ProVerif
(Dolev-Yao attacker)

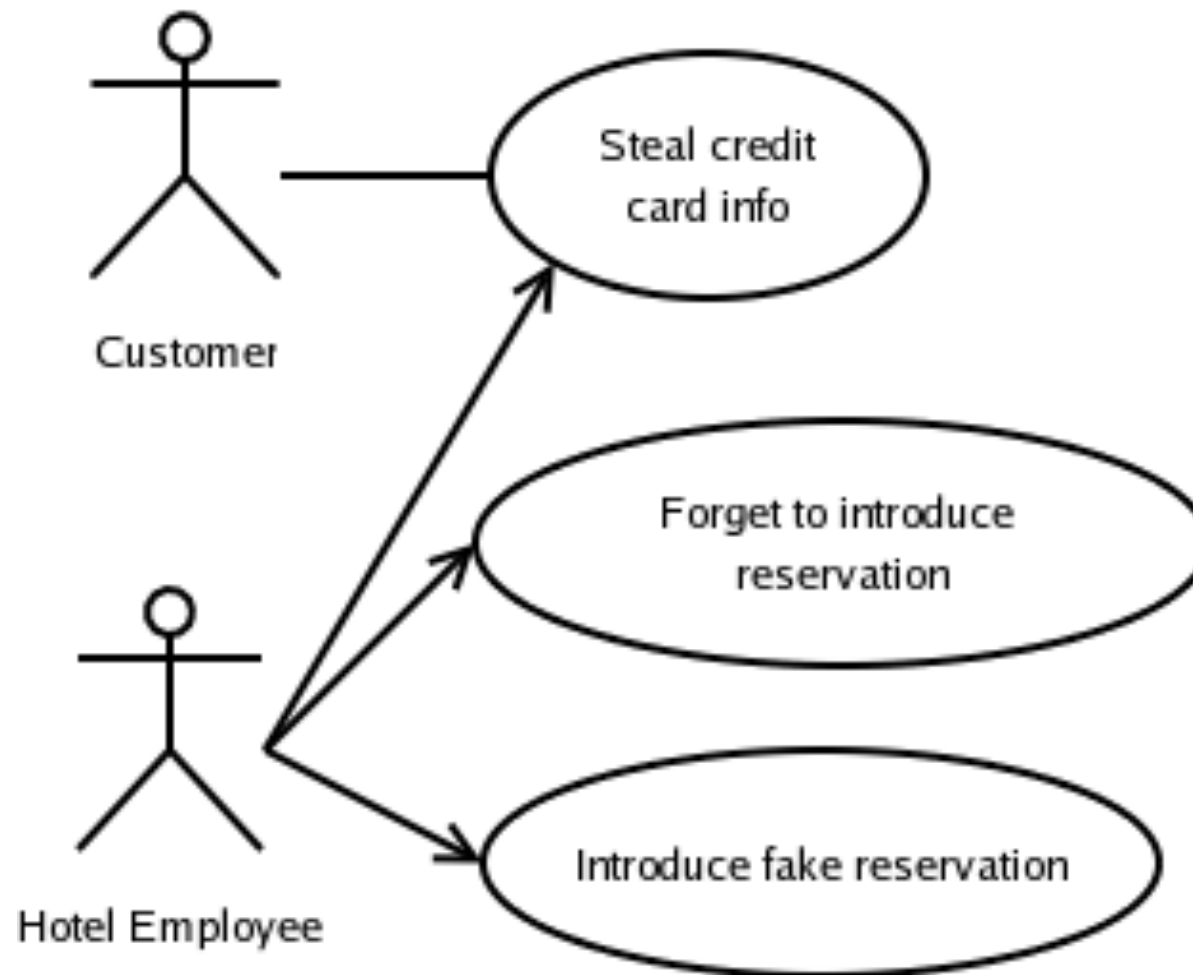
Simulations: Analyzing the Impact of Security



Abuse Cases

- McDermott & Fox, 1999
- Negative use cases for modeling security requirements
- Specify an interaction between a system and one or more actors, where the results of the interaction are harmful to the system or one of the actors in the system
- Actors are the same that participate in use cases

Abuse case: Reservation System



Limits of Abuse Cases

- Model security requirements separately from functional requirements
 - Abuse case diagrams show abuse only, not abuse together with normal use
 - They do not investigate relations between use and abuse

Misuse Cases

- Guttorm Sindre and Andreas Opdahl, 2000
- Extend use cases for modeling security requirements
- Specify behaviour that the system should avoid
- Specify how a misuser can damage the system

Misuse Case: Concepts

- Misuser
 - hostile actor
 - a similar notation as an actor in use cases, except the misuser has a black "head" instead of white
- Misuse case
 - course of actions performed to do harm to a stakeholder or the system itself
 - behavior that is not wanted in the system
 - illustrated by black circles
- Use cases
 - functionalities of the system
 - countermeasures against misuse
- Relations
 - “includes” and “extends”
 - “prevents”: use case prevents the activation of a misuse case
 - “detects”: use case detects the activation of a misuse case

Use / Misuse Case diagrams

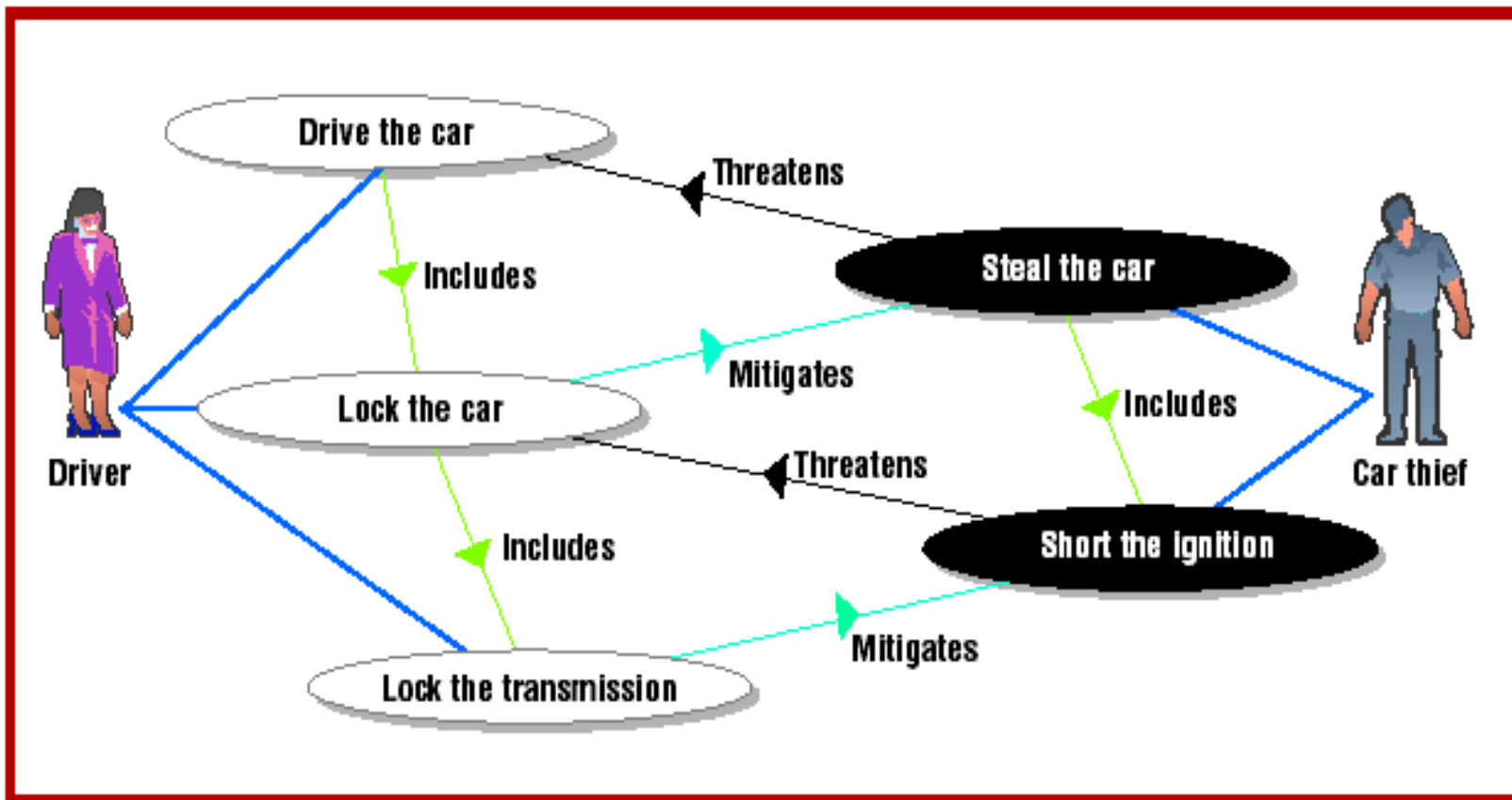
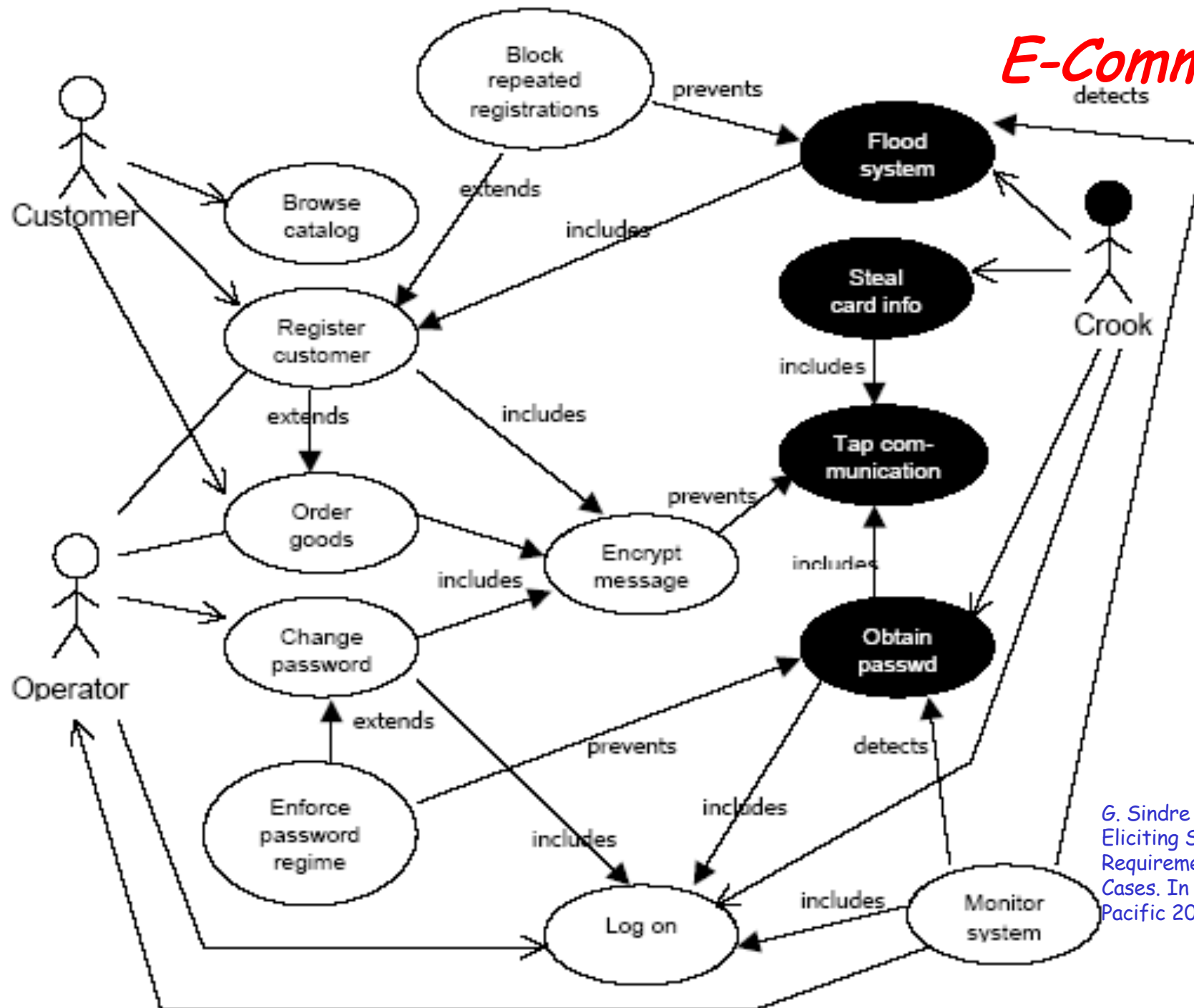


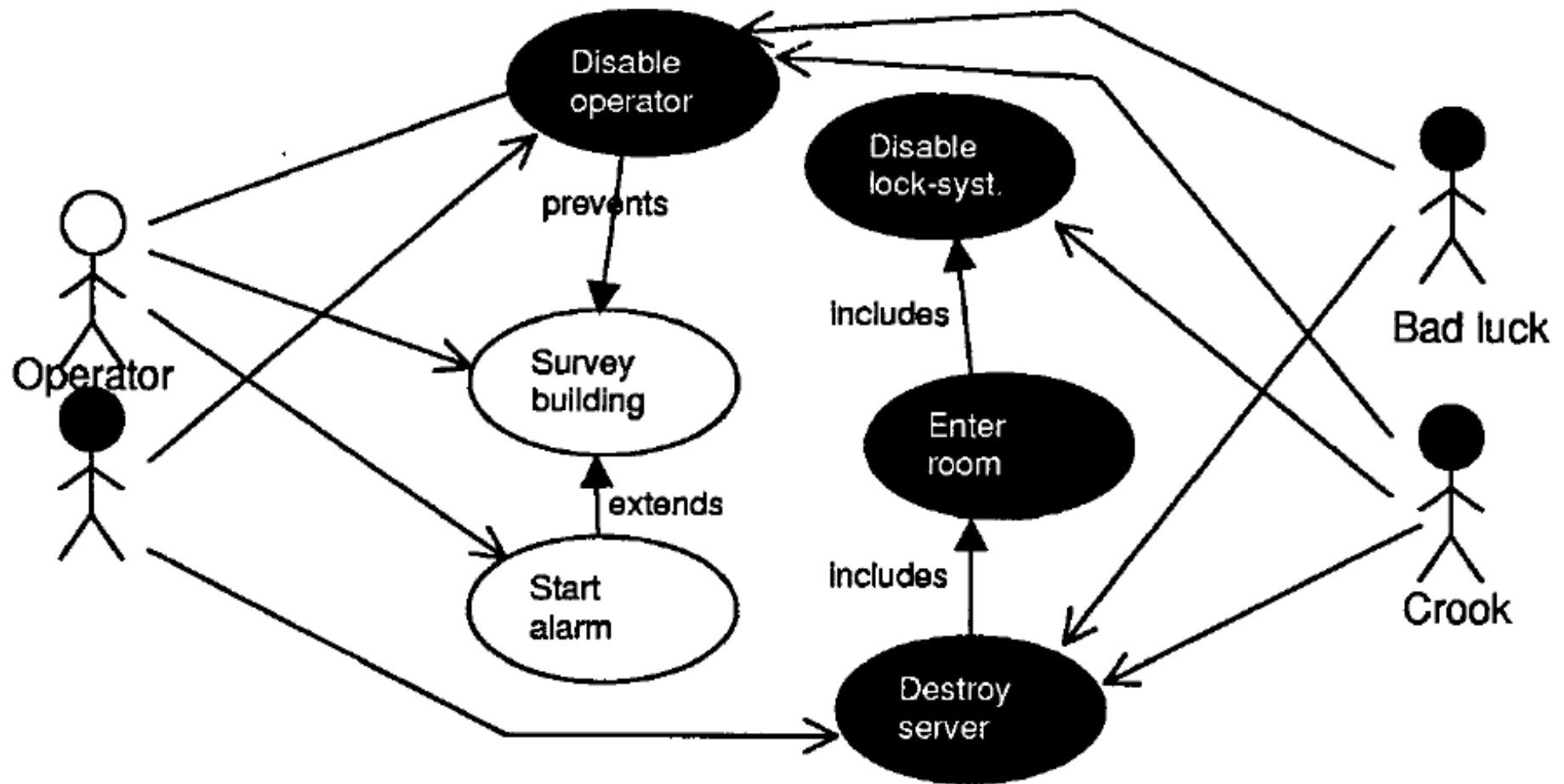
Figure 1. Use/misuse-case diagram of car security requirements. Use-case elements appear on the left; the misuse cases are on the right.

[Alexander 2003]

G. Sindre and A. Opdahl.
Eliciting Security
Requirements by Misuse
Cases. In Proc. of TOOLS
Pacific 2000.



Special mis-actors



Pros

- Focus on security in the early phases of the software development process
- Increase the chance of discovering threats that otherwise would have been ignored
- Help to trace and organize the requirements specification
- Help to evaluate requirements
 - the real cost of implementing a use case includes the protection needed to mitigate all serious threats to it
- Easy to reuse in new development projects

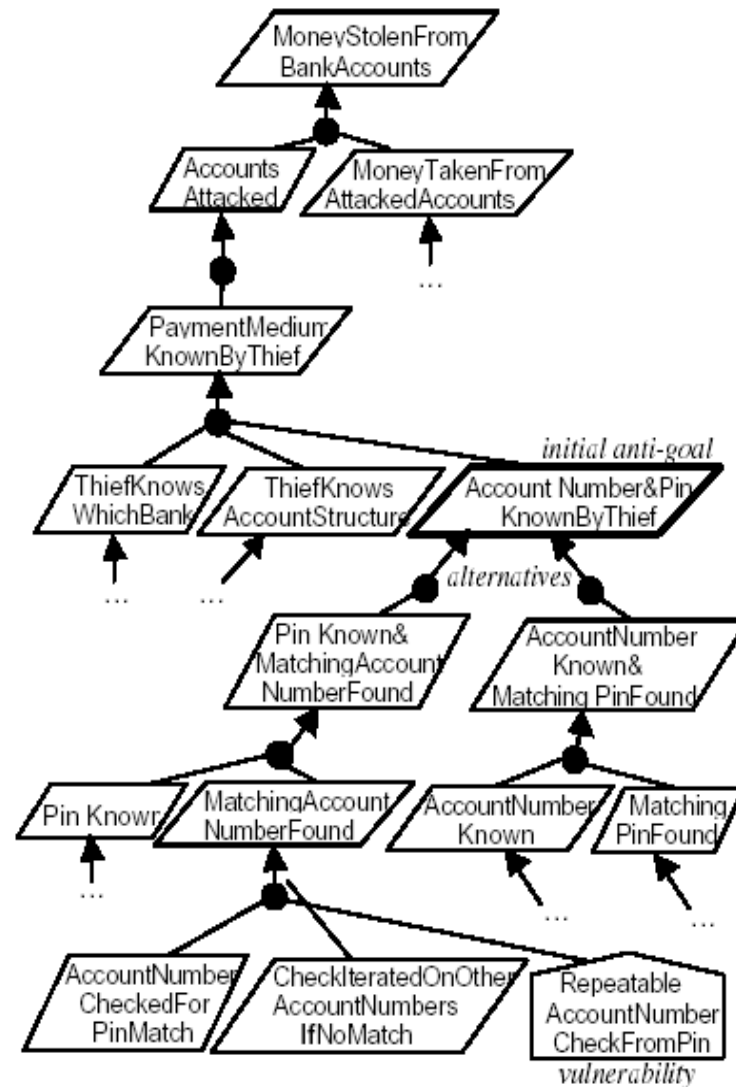
Cons

- Use/Misuse case are informal
 - No clear semantics
 - (Hence) NO formal analysis
- No knowledge on how to write *good quality* misuse cases
- The focus is ONLY on the system-to-be
- NOT suitable for all kinds of threats
- There is not always an identifiable misuser and the misuse case may not always consist of an identifiable sequence of actions

KAOS: Anti-goals

- Obstacles sufficient for modeling and resolving non-intentional obstacles (accidental obstacles)
- Too limited for modeling and resolving intentional obstacles (malicious obstacles)
- Active attackers also modeled together with their own goals, capabilities, and the vulnerabilities they can monitor or control (anti-models)
- Anti-goals are the **intentional obstacles to security goals**

Anti-goals (KAOS)



[Van Lamsweerde 2004]

Building Anti-models

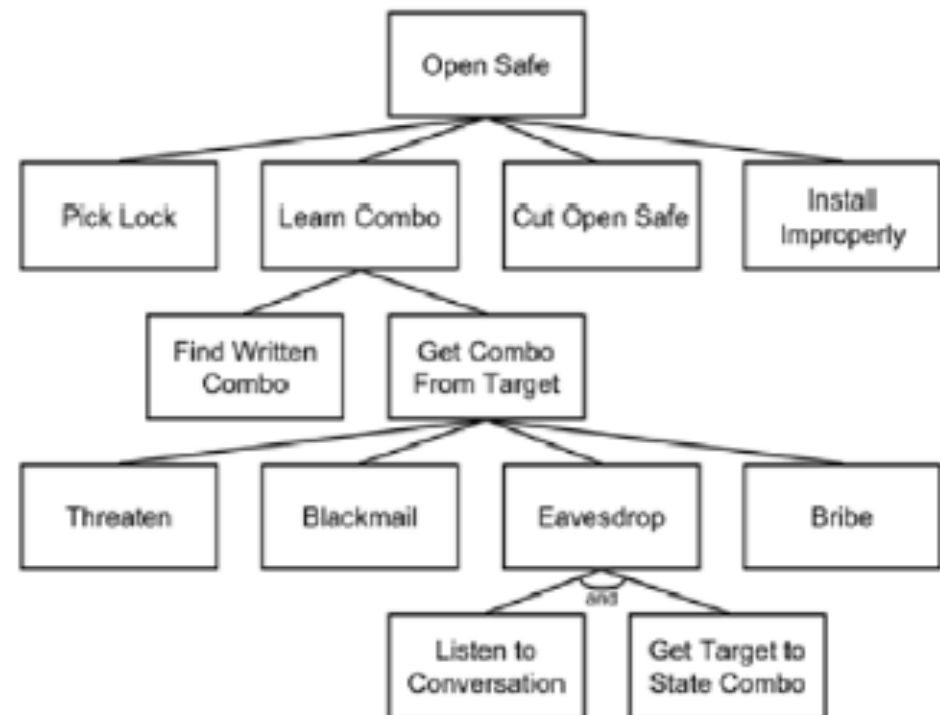
- Root anti-goal are obtained by negation of security goals
- For each anti-goal, potential attacker are identified (WHO)
- For each anti-goal and corresponding attacker, the higher level anti-goals are identified (WHY)
- For each anti-goal and corresponding attacker, the lower level anti-goals are identified (HOW)
- AND/OR refinement process for anti-goals
 - realizable by the attacker (anti-requirements)
 - realizable by the attackee (vulnerabilities)
- Anti-models are derived from anti-goals formulations
- Anti-requirements are defined in terms of the capabilities of the corresponding attacker

Limits of Antigoals

- Modeling attackers is difficult
- We have to consider all the possible obstacles even the ones unknown
 - Many protocols for security are been proved to be incorrectly after some years they are designed
- Many system vulnerabilities depend on the particular implementation
- Software vulnerabilities are not completely known

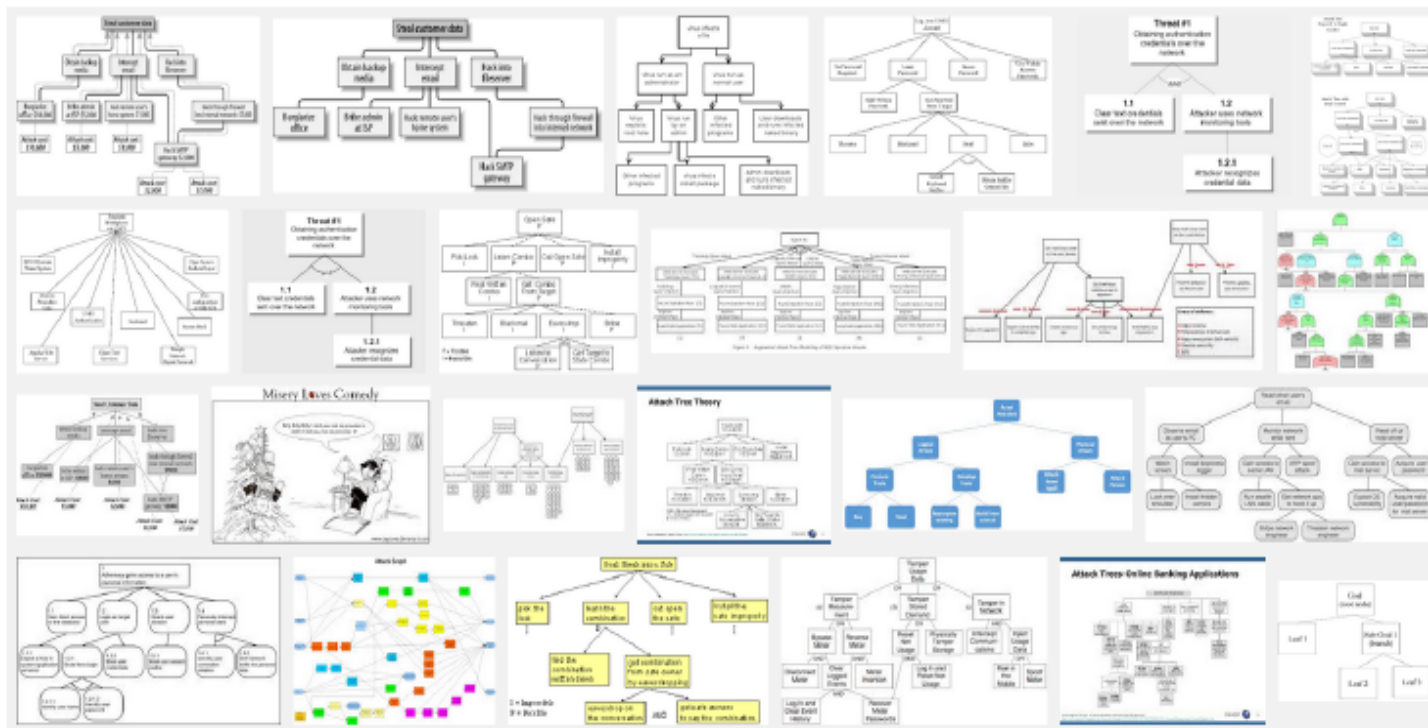
Threat Analysis: Attack Trees

- Originate from fault trees
- Introduced by Bruce Schneier (1999)
- Depict how a system element can be attacked
 - Helps finding attack countermeasures
- Root attack, children, leaves
- OR and AND relations between children
- These are NOT anti-goals like in KAOS! (not obstacles to security objectives)
 - In fact, security objectives derive from threats! (requirements elicitation)
 - Ultimately leads to risk analysis



Attack Trees: not so simple!

- Come in (too) many flavors ...

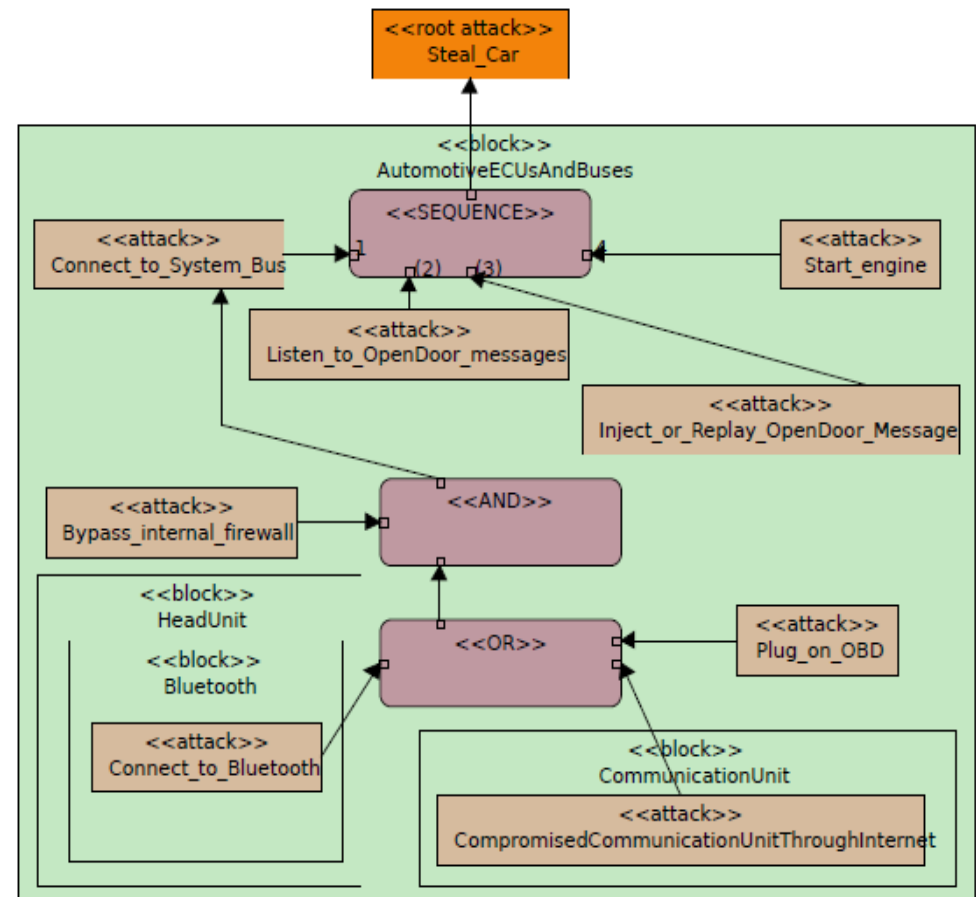


source: Google Images

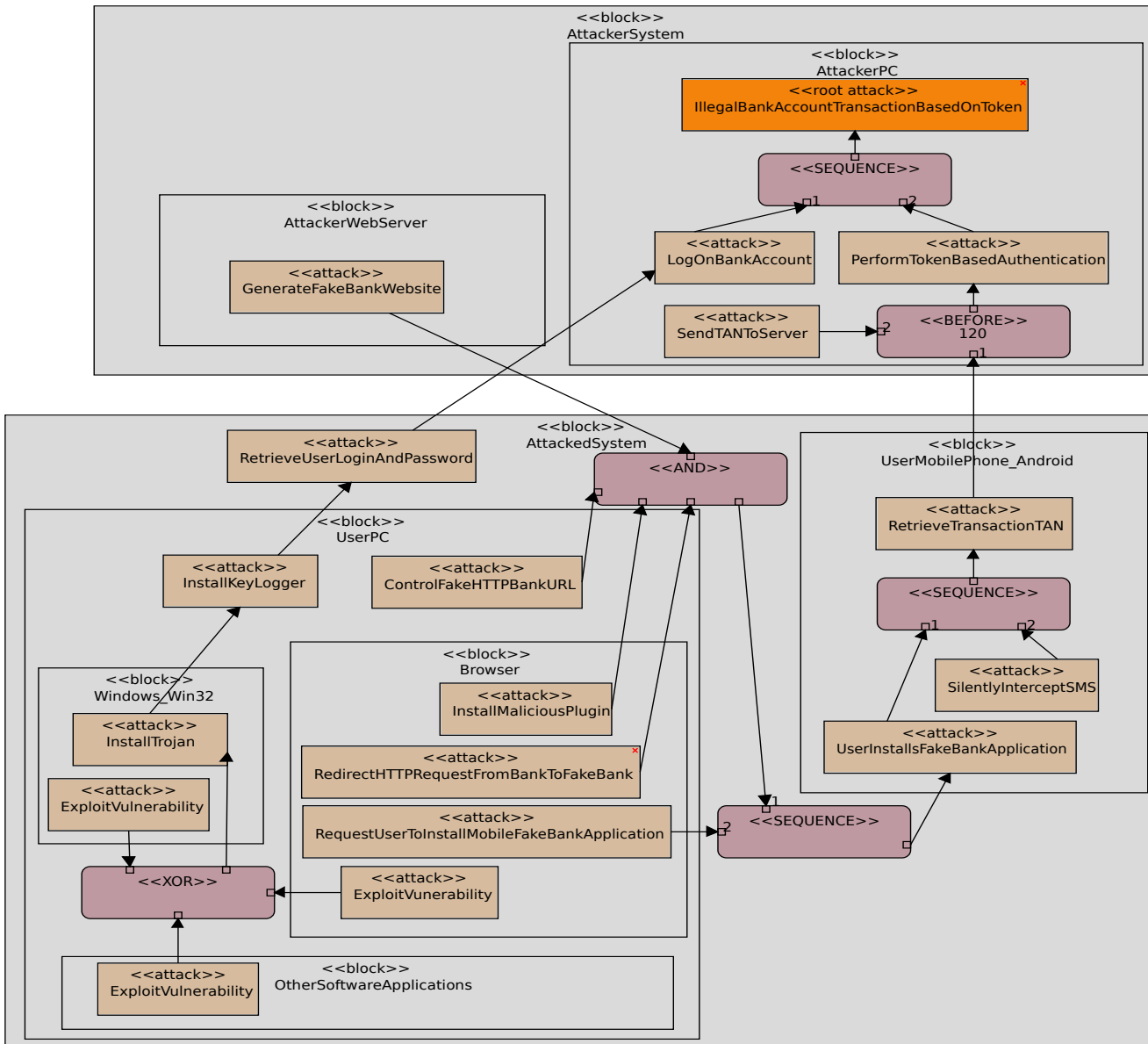
- Reusing attacks?

SysML-Sec: From Attack Trees to Attack Graphs

- Relations between attacks = constraints
 - Logical (AND, OR, XOR)
 - Ordering (SEQUENCE, BEFORE, AFTER)
- HW/SW mapping is very important
 - Documentation of attacks and matching countermeasures
 - Formal analysis of attack perimeter in architecture
- Reuse perspectives
 - E.g. better documentation for CVEs



SysML-Sec: The Zeus/Zitmo malware attack



Attacker System

Target of Attack (Windows Host, Browser, Mobile Phone)