Universität Hamburg

Finding and Preventing Buffer Overflows

An overview of scientific approaches

Martin Johns

Fachbereich Informatik
SVS – Sicherheit in Verteilten Systemen

Me, myself and I

- Martin Johns
- johns at informatik.uni-hamburg.de
- Security researcher at the University of Hamburg
- Member of the secologic project
  - Research project carried out by SAP, Commerzbank, Eurosec and the University of Hamburg
  - Goal: Improving software security
  - Visit us at http://www.secologic.org

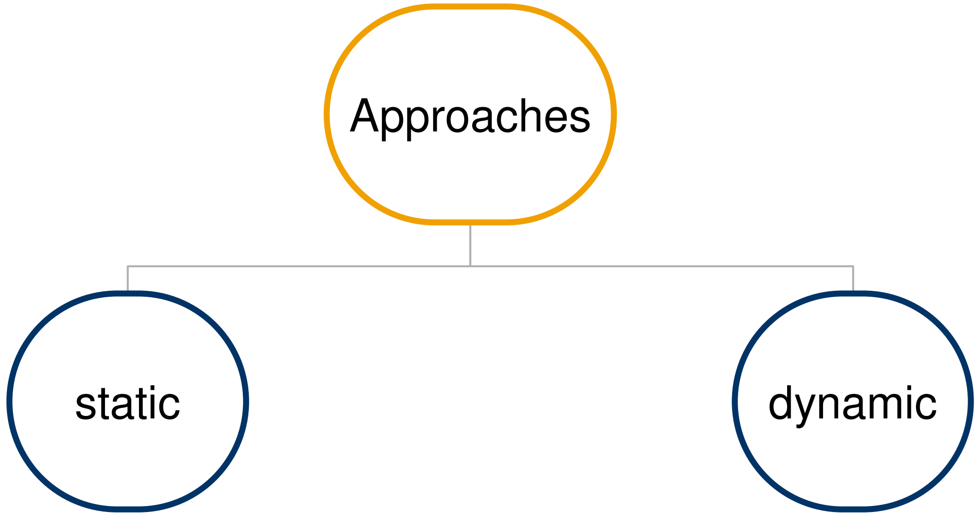
Overview (I)

- Only tools that are applicable by the programmer are presented
  - There are also counter measure that can be applied by the administrator of the application
- (mostly) tools with origin in academia
- Commercial tools exist but are rarely explicit about their internal algorithms

Overview (II)

- Most approaches concentrate on the C language
  - C is widely used (especially for system programming)
  - A lot of security problem are caused by C programs
  - C is "easy" to check
    - The control flow of a program is (mostly) determined on compile time
    - C programs are often vulnerable on a syntactic level
  - C is "hard" to check
    - Pointer arithmetic and type casting
    - Heavy use of preprocessor
    - More than one C
- Depending on the tool different kinds of vulnerabilities are detected / prevented
  - Buffer Overflows
  - Heap Corruption
  - Format String Exploits

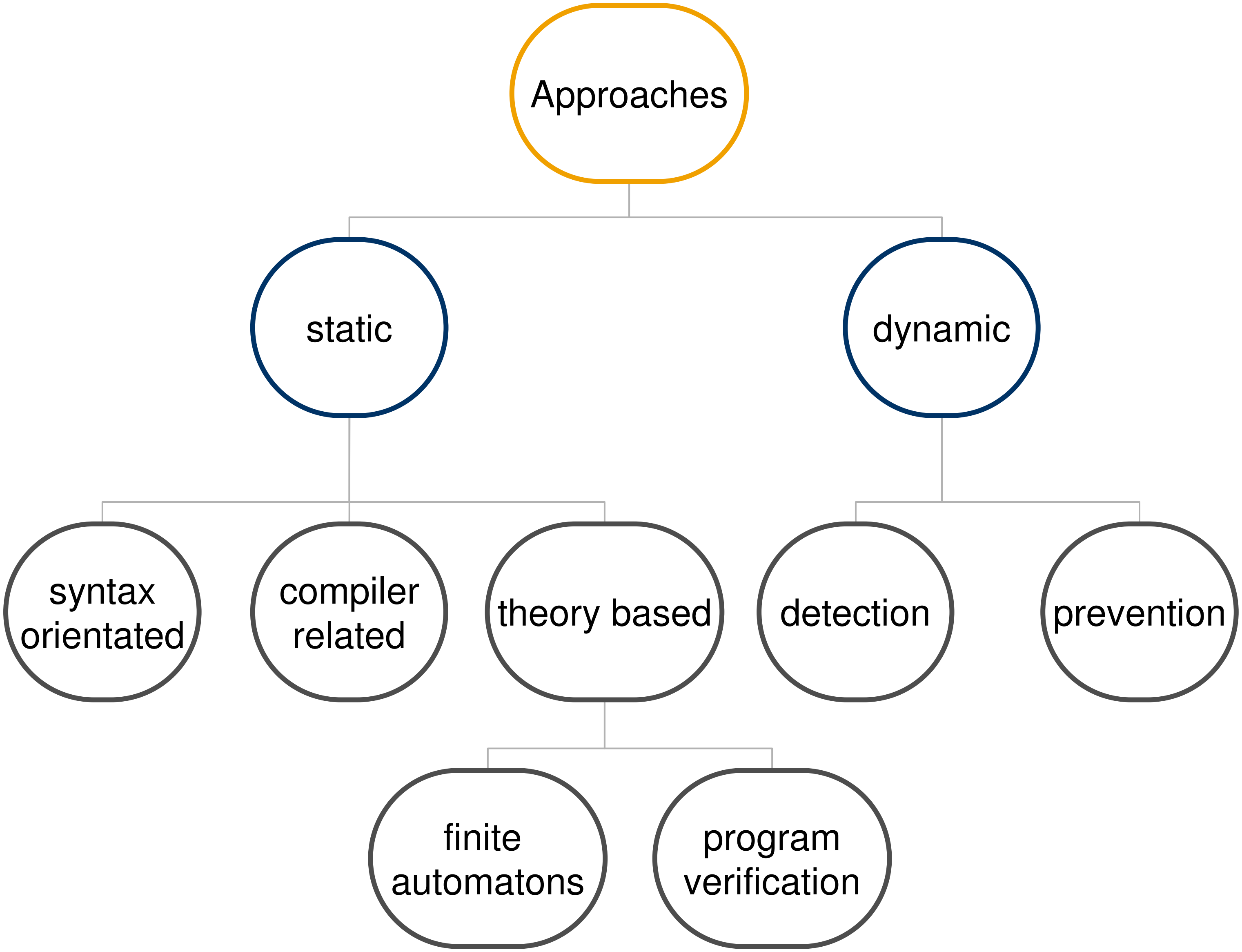
A classification



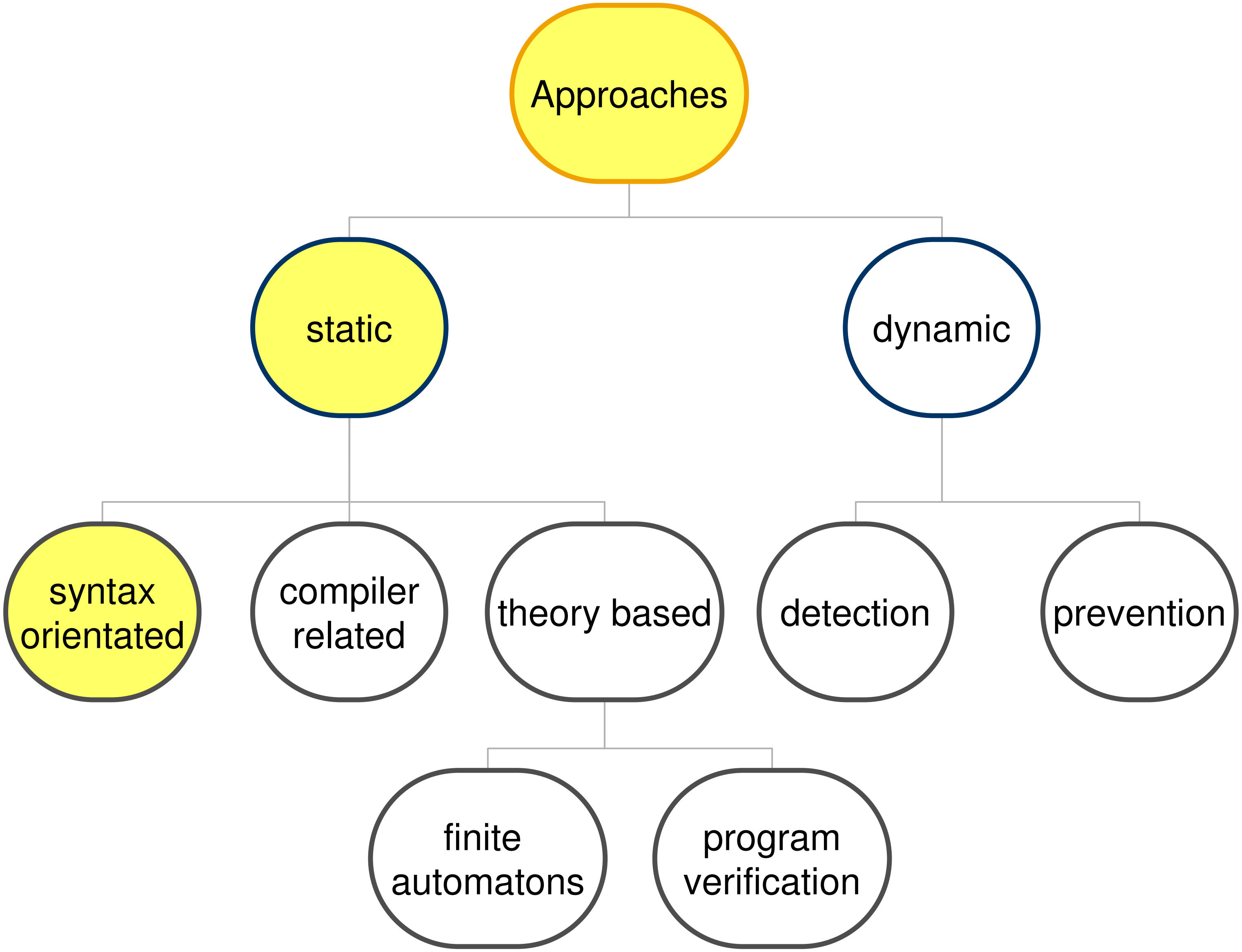
Static: check is done before / during compilation

Dynamic: check is done on runtime

A classification (teaser)



A classification



Static tools: syntactic analysis

- A lot of security problems of C programs are caused by "unsafe" library functions
- Example:
  `strcpy(dst,src)`
- These functions are comparably easy to spot

Internals:

- Syntactic tools examine on a per statement basis
- Usually these kind of tools operate on an internal representation of the source code
  - E.g. token stream, AST, etc.
- This helps to eliminate obvious sources of false positives
  - Comments
  - Strings

Static tools: syntactic analysis (II)

Some tools:

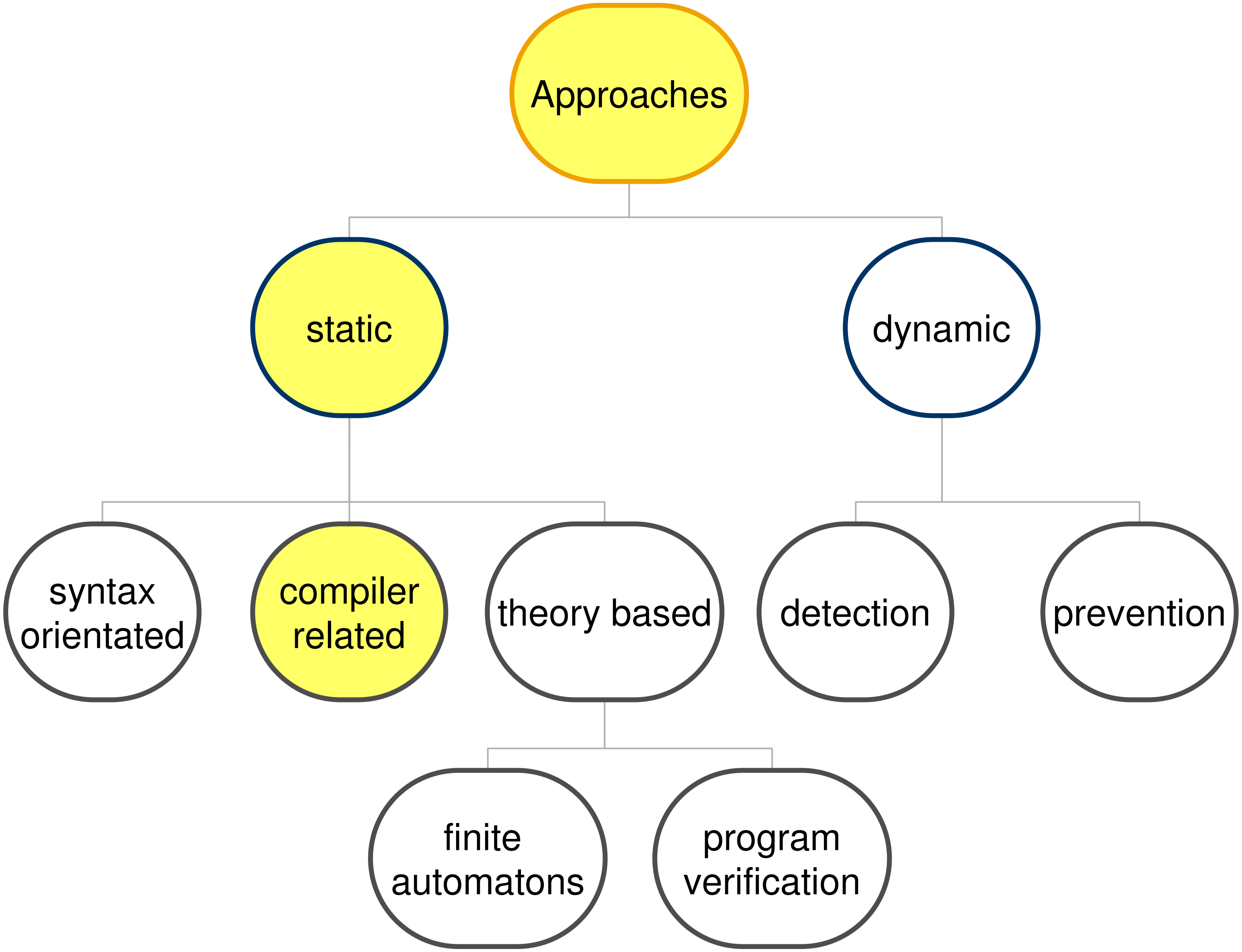
- **Flawfinder (2001)**
  - **Written by David Wheeler (Author of “Secure Programming for Linux and Unix)**
  - **Displays the code context of the found vulnerable constructs**
- **ITS4 (2001)**
  - **Assigns severity levels to warning**
  - **Also checks for some TOCTOU-problems**
- **RATS (2001)**
  - **Differentiates between heap- and stack allocated buffers**
  - **Dictionaries for C(++), Python, Perl and PHP**

Static tools: syntactic analysis (III)

Limitations of syntactic analysis:

- **Only a limited context is taken into account**
  - **(sometimes) type qualifier (e.g. `strcpy` with a `const` source buffer is not exploitable)**
  - **(sometimes) preliminary checks**
- **Complex contexts are ignored**
  - **Intra-/Interprocedural dependencies**
  - **control flow**
  - **data flow**
- **Consequences**
  - **Syntactic analysis is prone to false positives (e.g. *every* `strcpy()` gets reported)**
  - **Syntactic analysis is unable to find problems of higher semantic level (e.g. “double free”, access violations, etc.)**

A classification



Static Tools: compiler related approaches

- **General observation: static analysis tools and compilers share common techniques**
- **Compiler actions:**
  - **Parsing source code to abstract representation (token stream, AST, etc.)**
  - **Generating control- and data-flow graphs (for optimization)**
  - **Enforcement/check of constraints (e.g. type checks)**
- **The more advanced the compiler is, the better its "understanding" of the source code's semantics**
  → interesting aspect for security related analysis

Compiler related approaches: BOON

Buffer Overrun detectiON (2000)

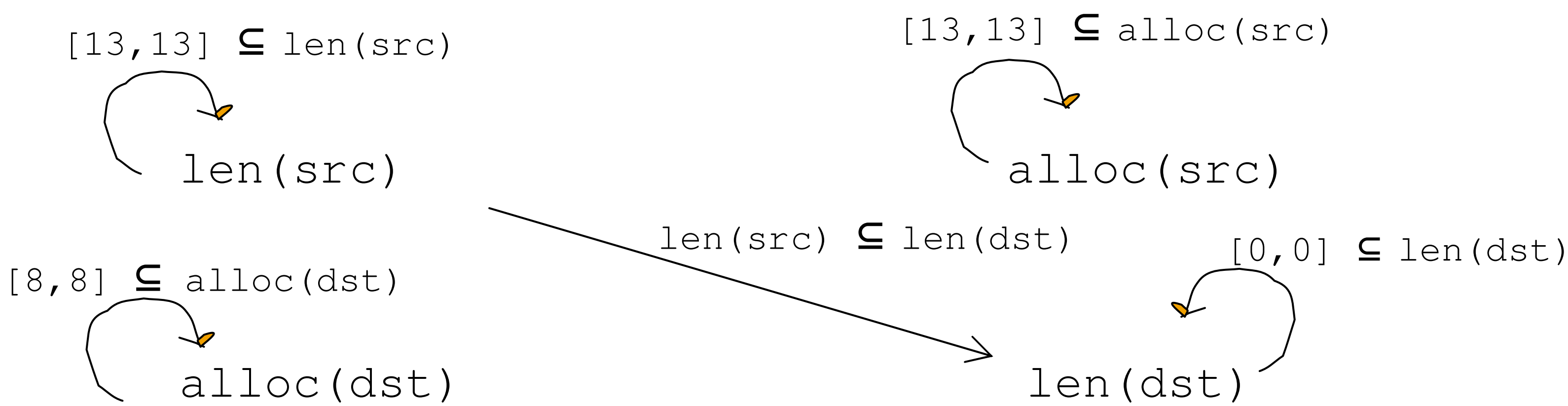
- **Introduces a theoretical "C String" abstract data type consisting of char-buffer & string-library functions**
- **Examines code for potential C String overflows**
  - **→ Only char-buffers are considered**
  - **→ Only Buffer Overruns caused by library functions are detected**
- **Ignores control flow**
- **The state of every C-String s is represented as two integer ranges** `alloc(s)` **and** `len(s) = [min,max]`
- **The safety property to be verified is** `max(len(s)) <= min(alloc(s))`
- **Integer range algebra:**
  `a ⊆ b → b = [min(min(a),min(b)), max(max(a), max(b))]`
  example: `a = [2,5]; [4,7] ⊆ a → a = [2,7]`
- **For each statement an integer range constraint is constructed:**

```
s = malloc(6*sizeof(char));      → [6,6] ⊆ alloc(s),
fgets(s,n,…);                    → [0,n] ⊆ len(s)
strcpy(dst,src)                  → len(src) ⊆ len(dst)
```

Compiler related approaches: BOON (II)

- **A directed graph representing the constrain system is constructed:**
  - **Vertices: the variables** (`len(s), alloc(s)`)
  - **Edges: the constraints (representing the functions)**

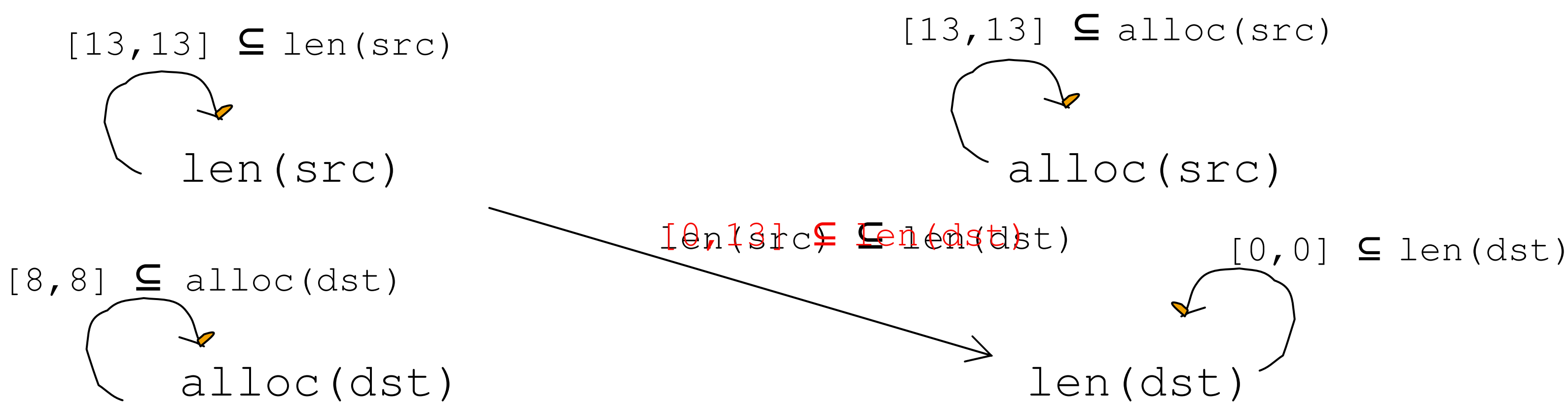
```
1: char* src = "testtesttest";
2: char* dst = malloc(8*sizeof(char));
3: strcpy(dst,src);
```



Compiler related approaches: BOON (II)

- **The constrain solving algorithm descends through the graph until all variables stopped changing → a fixpoint is found**
- **A potential Buffer Overrun is found if for some string s**
  `max(len(s)) > min(alloc(s))`

```
1: char* src = “testtesttest”;
2: char* dst = malloc(8*sizeof(char));
3: strcpy(dst,src);
```



Compiler related approaches: CQUAL

CQUAL (2001)

- **Inspired by Perl's "tainted" mode**
- **Detects format string vulnerabilities**
- **Uses an extension to the language's type system**

How it works:

- **Introduces new type qualifiers "tainted" and "untainted"**

```
untainted int i;
int main(int argc, tainted char* argv[]);
```

- **Type inference rules are applied to propagate the type qualifier:**

```
int a;
tainted int b;
a = b + 2;
```

→ a inherits the type qualifier "tainted"

Compiler related approaches: CQUAL (II)

Type qualifiers induce a subtyping relationship on qualified types

- **untainted is a subtype of tainted**
- **The consequences:**
  - **It is allowed to assign an "untainted" value to a "tainted" variable**
  - **It is forbidden to assign a "tainted" value to a "untainted" variable**

```
void f(tainted int);
untainted int a;
f(a);
```

OK

```
void g(untainted int);
tainted int b;
g(b);
```

TYPE ERROR

Compiler related approaches: CQUAL (III)

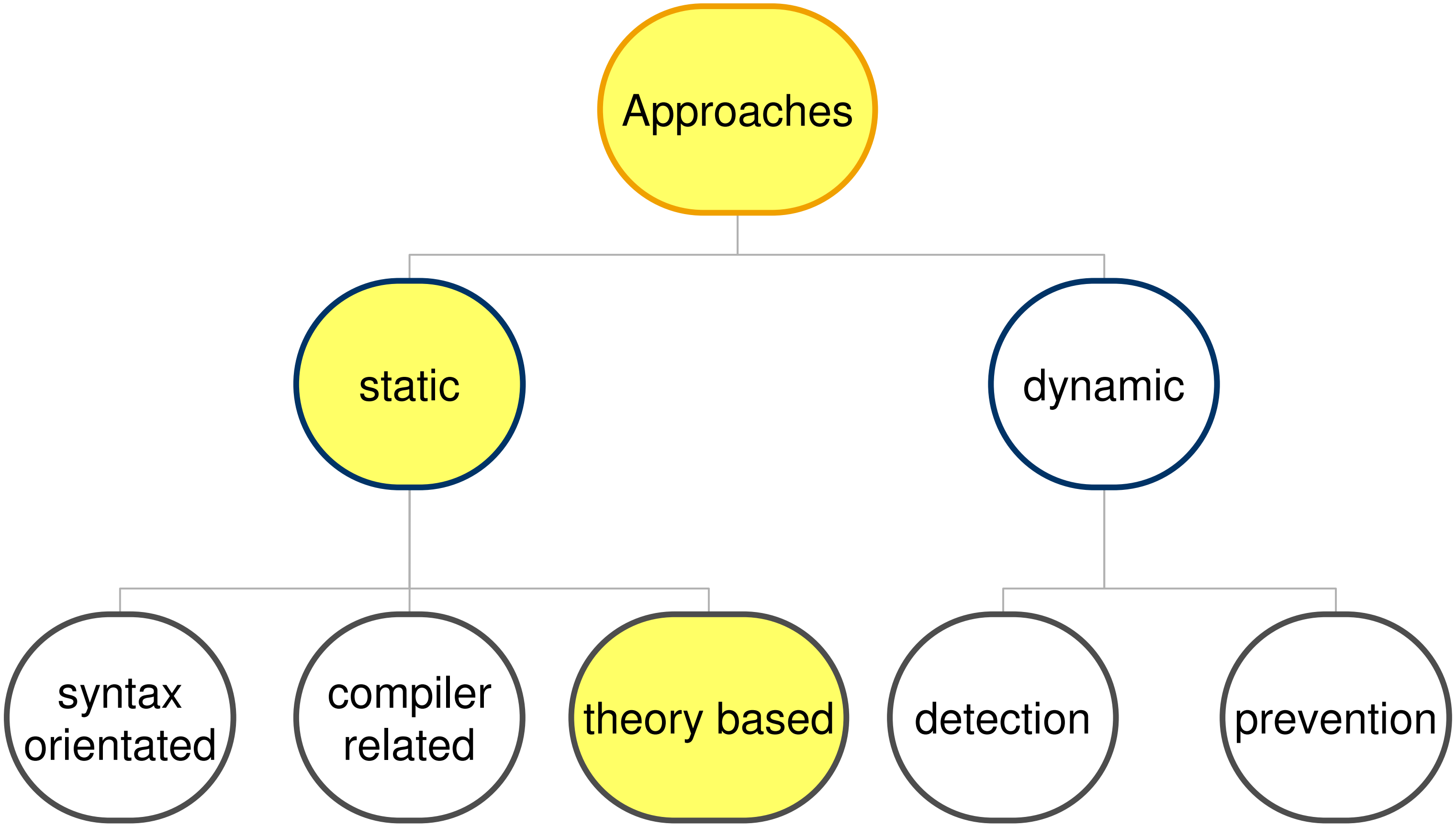
Finding Format String vulnerabilities

- **Goal: find data paths which allows a user controlled variable to define a format string**
- **All return values of function calls that contain user input are marked as "tainted"**
- **All parameters of format string exploit suspicious functions are marked as "untainted"**

Example:

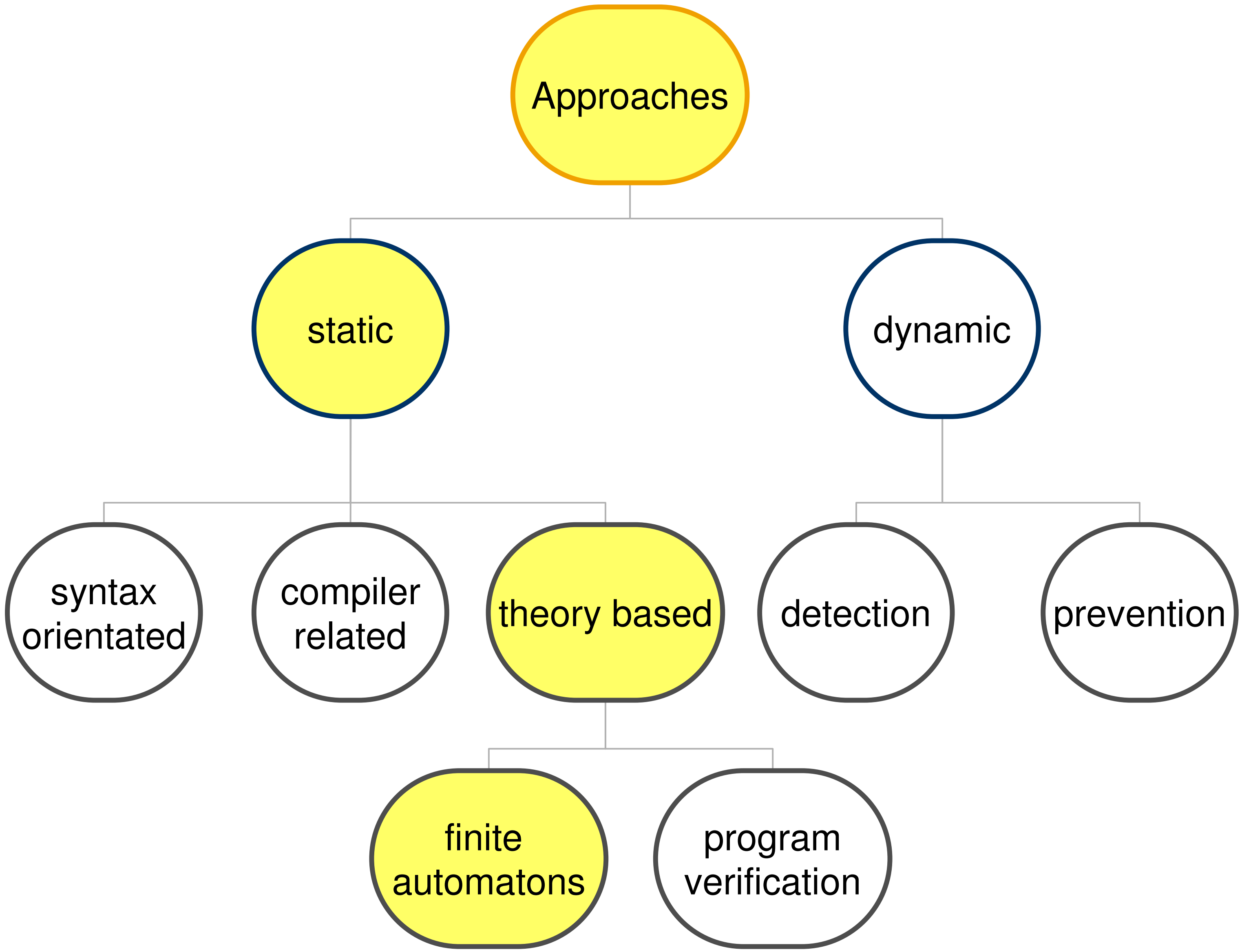
```
tainted char* getenv(char* name);
int printf(untainted char* fmt, …);
char* s;
s = getenv("PATH");          s gets marked as "tainted"
printf(s);                   type error
```

A classification



"Theory based" approaches borrow concepts form the field of theoretical computer science

A classification



Finite Automatons: XGCC

XGCC (2002)

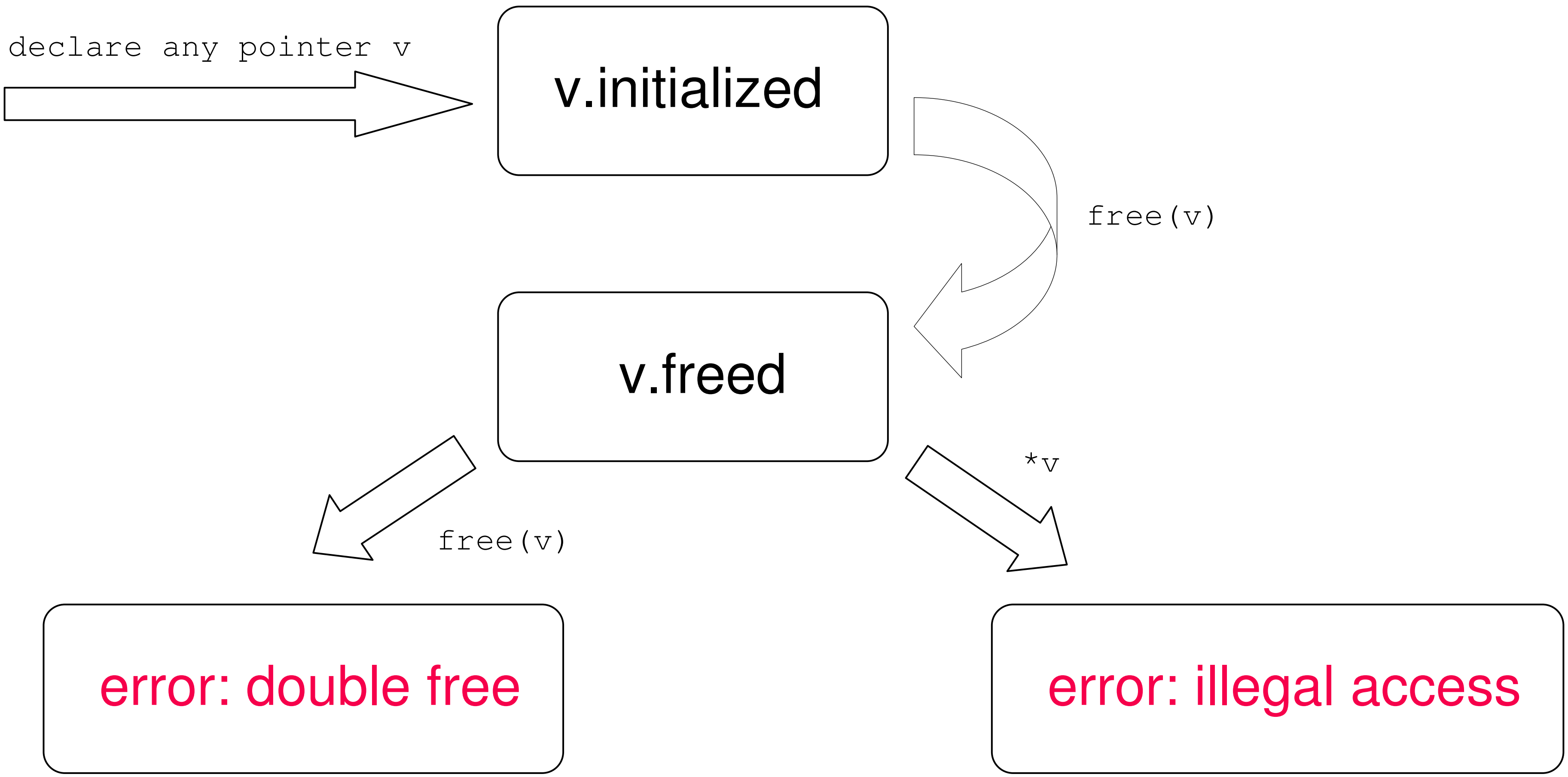
- **XGCC uses finite automatons to track assurance of conditions**
- **These automatons are described in the tool’s language “metal”**

Checking:

- **A control flow graph is generated from the source code**
- **The tool walks through the graph (using depth first search)**
- **The statements in the vertices of the cfg are examined**
  - **Creation of an automation**
  - **Triggering a transition of the automaton → the state of the automaton is updated**
- **If a branch (e.g. an if-statement) in the graph is encountered all automaton which are affected by the branch’s body are duplicated**

Finite Automatons: XGCC (II)

Example: double free errors



Code	Automaton
`int *v;`	v.initialized
`v = malloc(3*sizeof(int));`	
`*v = 3;`	
`free(v);`	v.freed
`if (…){`	v → v1.freed, v2.freed
`  *v = 52;`	v1.error_illegal_access
`} else {`	
`  free(v);`	v2.error_double_free
`}`	

Finite Automatons: MOPS

<u>MO</u>del checking <u>P</u>rograms for <u>S</u>ecurity properties (2002)

- MOPS uses a comparable approach to XGCC
- For every checked property an FSA (finite state automaton) is constructed
- The program is compiled into an PDA (push down automaton)
- The tool checks if the FSA and the PDA intersect using automaton algebra

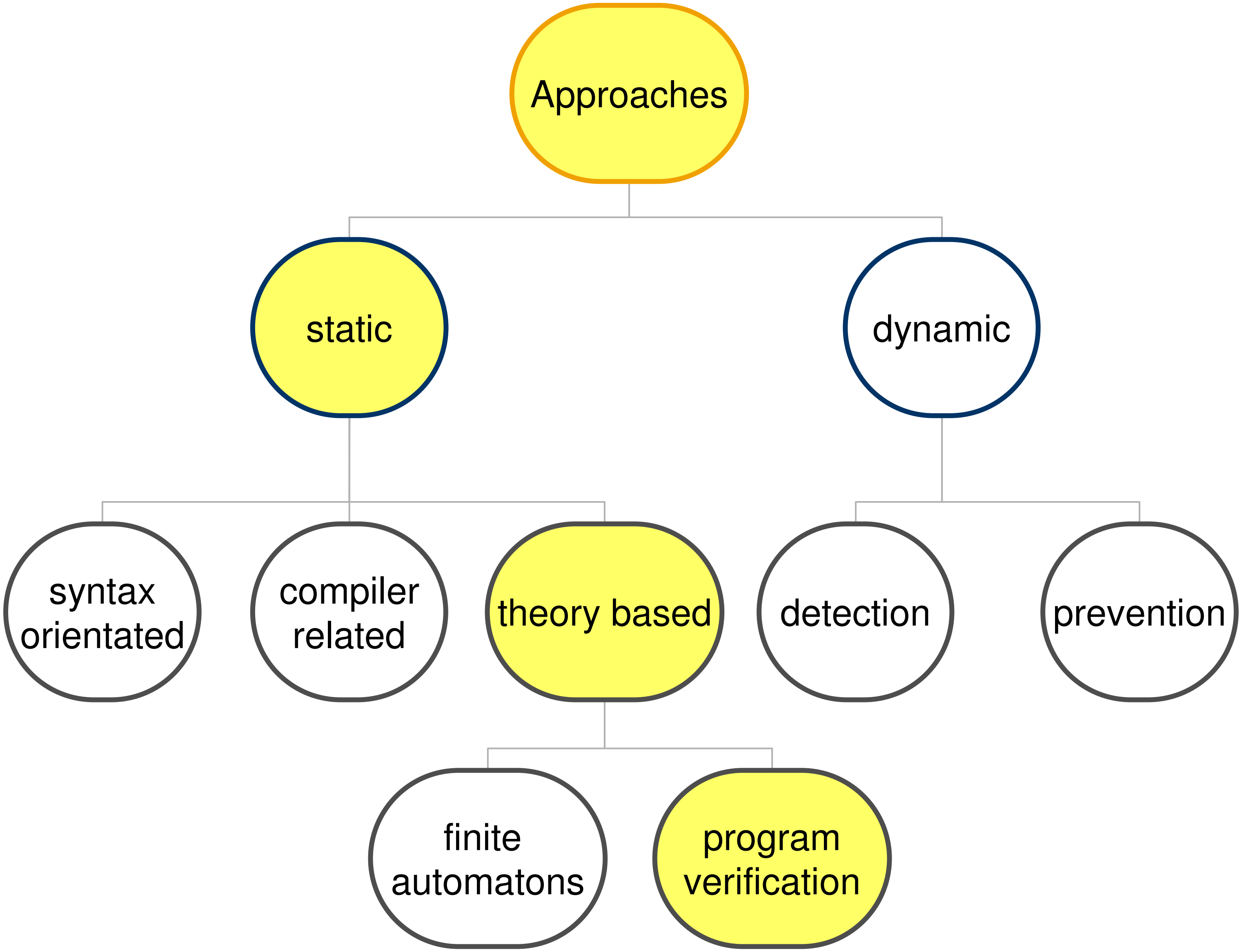
Properties of MOPS:

- Control flow sensitive
- Sound
  - If the automatons are constructed correctly, MOPS is able to guarantee the absence of specified security problems

Problem: constructing automatons is nontrivial

- Unlike XGCC MOPS is unaware of program language specific properties (like pointer initialization)

A classification



Program verification

- preconditions `r`
- postconditions `e`
- Program $P = s_1; s_2; s_3; \ldots; s_n$

$\Big\}$ Specification `S` in first order logic

Wanted:

- Proof: `P` terminates, `{r}P{e}`

Proof:

- $Q_0 = r$ und $Q_n = e$
- Verification: $\{Q_0\} s_1 \{Q_1\} s_2 \{Q_2\} s_3 \ldots \{Q_{n-1}\} s_n \{Q_n\}$
- If $\{Q_{i-1}\} s_i \{Q_i\}$ for all $1 \leq i \leq n$ then follows $\{V\}P\{N\}$

Security tools only define pre- and postconditions for special constructs

- e.g. unsafe library functions

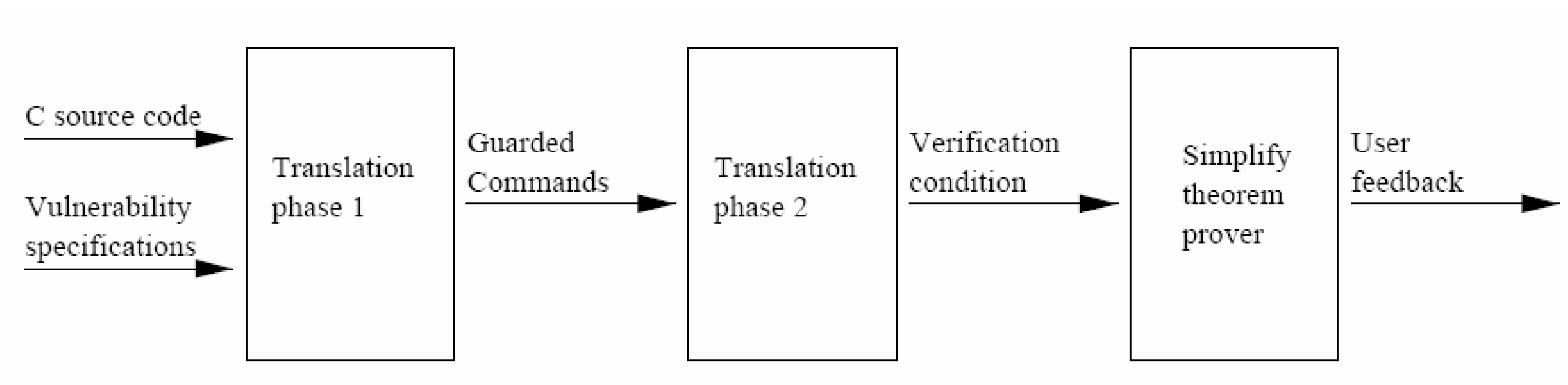
Splint

Splint (LCLint) (2001)

- **Uses pre- and post-conditions for detecting Buffer Overflows**
- **Supports four types of constraints:**
  - **maxSet, minSet: allocated space of a buffer**
  - **maxRead, minRead: used space of a buffer**
- **Pre- and post-conditions have to be added by the programmer:**
  ```
  /*@requires maxSet(dest) > maxRead(src)@*/
  ```
- **Splint is able to deduct postconditions for known code constructs:**
  ```
  char buf[42];
  → ensures minSet(buf) = 0 and maxSet(buf) = 41
  ```
- **The analysis:**
  - **A control flow graph is generated from the source code**
  - **Following the graph, Splint checks, if the pre-conditions can be met**
  - **If a precondition, that can't be verified, is found, the tool emits a warning**

Eau Claire (2002)

- Uses a theorem prover: “Simplify”
- Two step translation process
  - C code to Guarded Commands
  - Guarded Commands to verification condition
- “Guarded Commands” are (roughly) a translation of an instruction into its pre- and post-conditions
- Function calls are translated into the function’s pre- and post-conditions
- Annotated library functions are used for the analysis



Example:

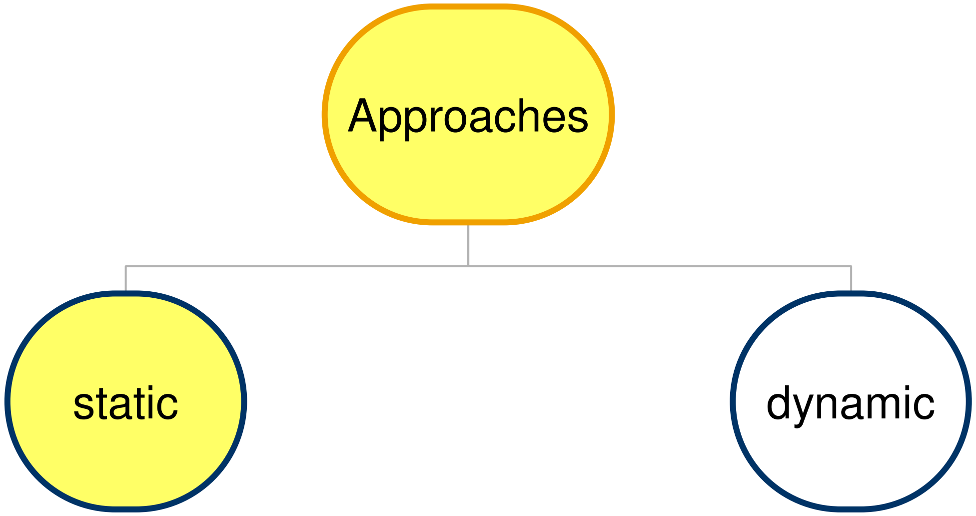
```
/*
spec strcpy(cpTo, cpFrom)
{
  requires $valid(cpTo):
       "the first argument must be a valid pointer"
  requires $string(cpFrom):
       "the second argument must be a valid string"
  requires $length(cpTo) > $string_length(cpFrom):
       "the array must be large enough to hold the entire string"

  modifies $elems(cpTo)

  ensures forall(i) ((i >= 0 && i <= $string_length(cpFrom)) implies
                    $final(cpTo[i]) == cpFrom[i])

  // next item is true but not necessary for the spec
  //ensures $string_length(cpTo) == $string_length(cpFrom)
}
*/
void strcpy(char* cpTo, char* cpFrom);
```

A classification

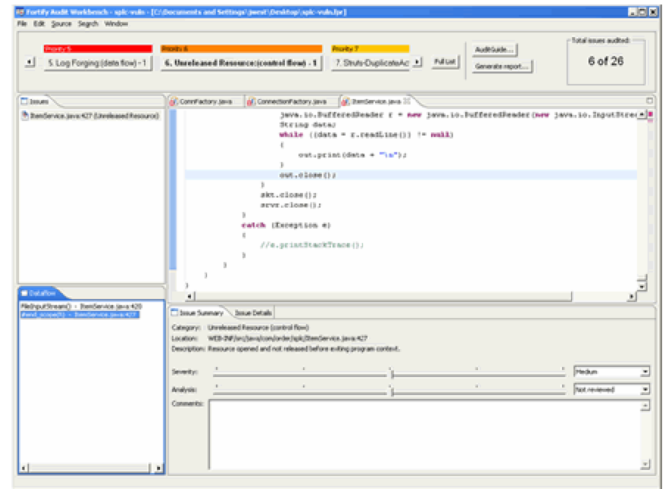


Commercial tools

Commercial Tools: Fortify

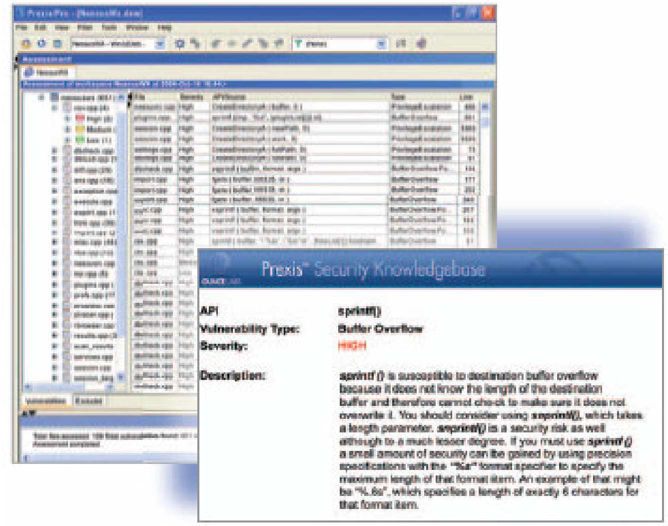
Fortify Source Code Analysis Engine

- Four modules:
  - Data Flow Analyzer
  - Semantic Analyzer
  - Control Flow Analyzer
  - Configuration Analyzer
- Multiple languages are supported
  - C, C++, Java, JSP, PL/SQL, C#, XML
- Supports custom “Rulepacks”
- Provides IDE Plug-Ins
  - Borland JBuilder
  - Eclipse
  - MS Visual Studio



http://www.fortifysoftware.com/products/sca/index.html

Commercial Tools: Prexis



Ounce Labs Prexis

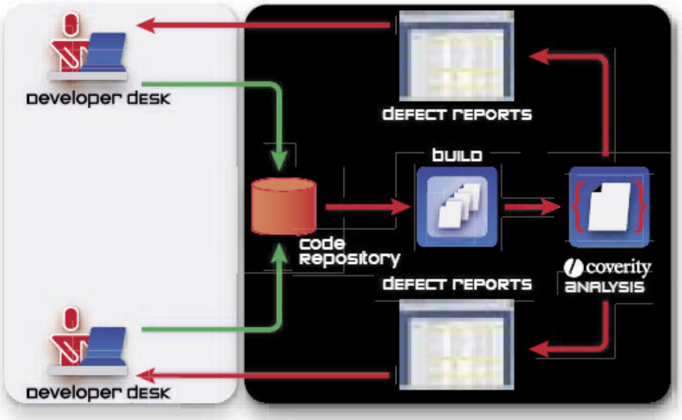
- **Two modules**
  - **CAM++: C/C++ Assessment Module**
  - **JAM: Java Assessment Module**
- **Context Sensitive Analysis**
- **“Multi Dimensional Method”**
  → Some code regions are checked more thoroughly than others
- **Checks for:**
  - Buffer Overflows, Privilege Escalations, Race Conditions, Improper Database Access, Insecure Cryptography, XSS, Insecure Account/Session Management, Command Injection, Insecure Access Control, DOS, Error Handling Problems, Insecure Network Communication, Poor Logging Practices, SQL Injection, Native Code Vulnerabilities, Dynamic Code Vulnerabilities
- **Only loosely integrated in IDEs**

http://www.ouncelabs.com/overview.html

Commercial Tools: Coverity

Coverity Prevent and Coverity Extend

- Based on XGCC
- “Prevent” enforces predefined conditions to eliminate known security problems
- “Extend” is a tool to add custom checks to the process
- Also checks for not security relevant errors
- Integrates in IDEs
- Provides source code Browser

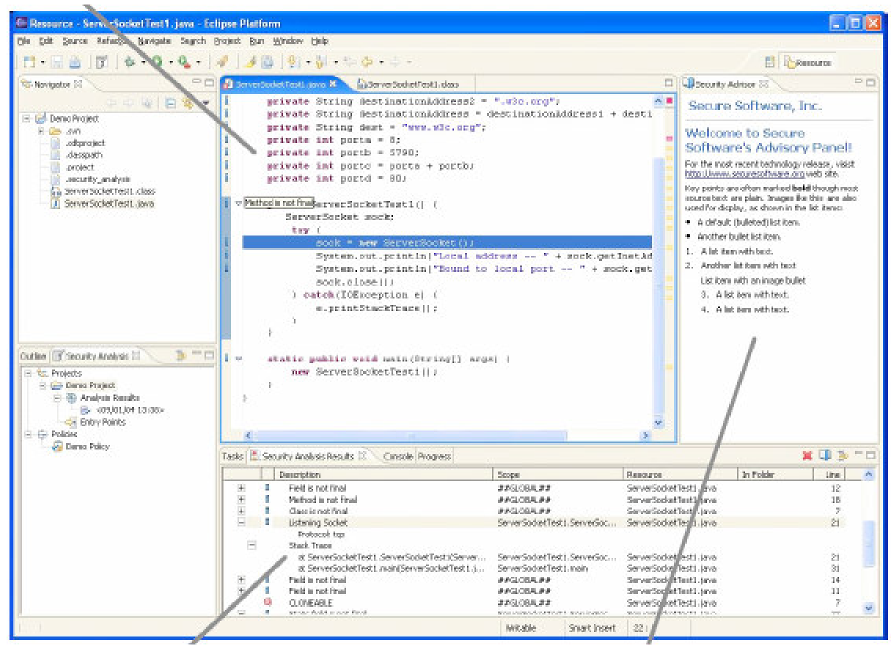


http://www.coverity.com

Commercial Tools: CodeAssure

Secure Software CodeAssure Workbench

- **Assessment of control and data flows**
  - ◆ Integer and string ranges
  - ◆ Function calls
  - ◆ Aliases
- **Knowledge:**
  - ◆ 40 types of vulnerabilities
  - ◆ Thousands of individual examinations and rules
- **Provides severity levels**
- **Language Packs:**
  - ◆ Java
  - ◆ C
  - ◆ C++
- **Integrated in Eclipse**

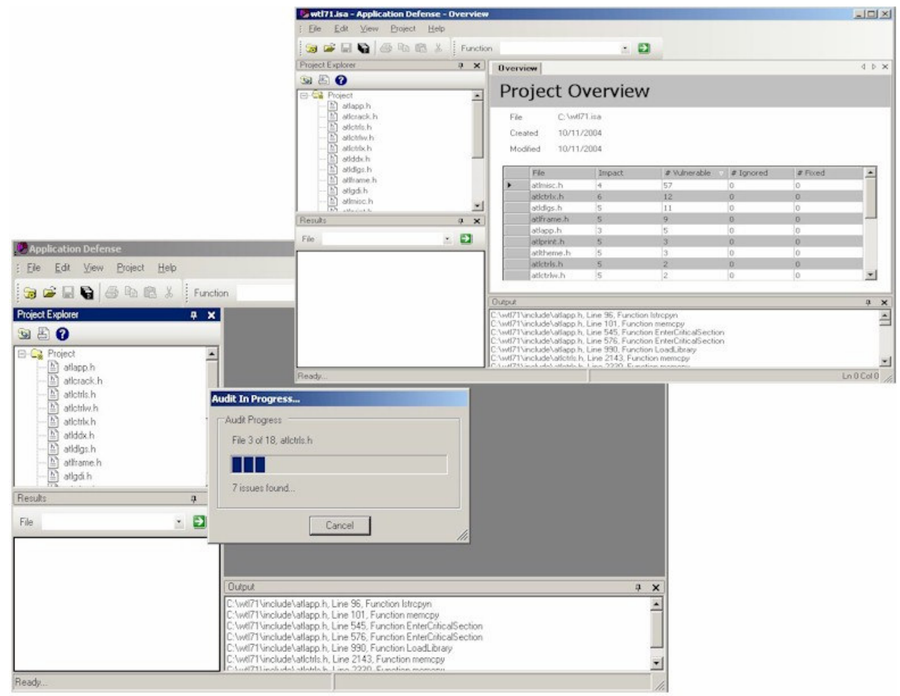


http://www.securesoftware.com/products/source.html

Commercial Tools: AppDefense Developer

Application Defense Developer

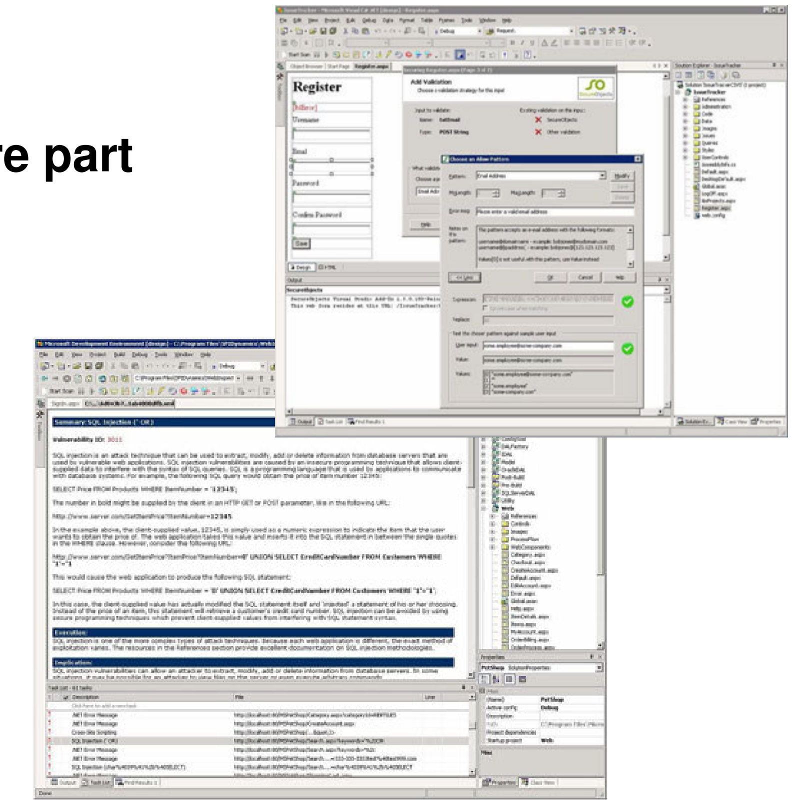
- Supports 13 languages:
  - C, C++, C#, VBScript, VBA, ASP, Jscript, JavaScript, PHP, Python, LISP, ColdFusion, Perl
- Time to check the code < 1 minute
- “proprietary artificial intelligence engine”
- XML Output
- Provides IDE
- Product only available in combination with the company’s other services



http://applicationdefense.com/ApplicationDefense_Products.htm#developer

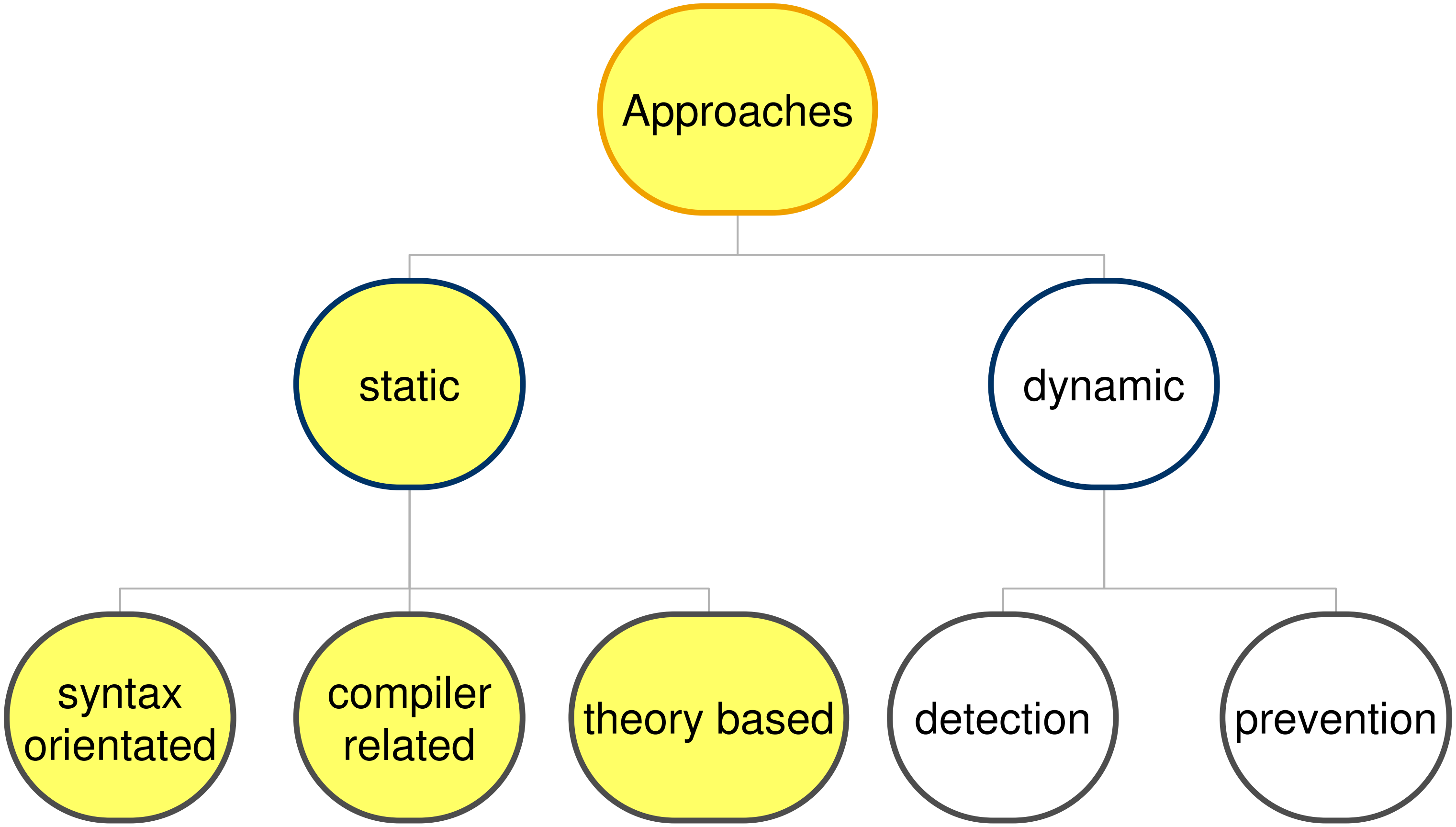
SPI Dynamics DevInspect and SecureObjects

- Integrates in Visual Studio.NET
  - Only checks languages, that are part of the .NET framework
- Focuses on Web Applications
  - Input validation
  - XSS
  - SQL injection
- Provides code fragments



http://www.spidynamics.com/products/devinspectso2003/index.html

A classification



A comparison of selected static tools

A comparison

- A comparison test was carried through by J. Wilander and M. Kamkar at the University of Linköping, SE (2002)
- The tested tool were: Flawfinder, ITS4, RATS, Splint and BOON
- For the test a uniform testfile was created, which included:
  - 15 unsafe buffer writings
  - 13 safe buffer writings
  - 8 unsafe format string calls
  - 8 safe format string calls

	Flawfinder	ITS4	RATS	Splint	BOON *
True Positives	22 (96%)	21 (91%)	19 (83%)	7 (30%)	4 (27%)
False Positives	15 (71%)	11 (52%)	14 (67%)	4 (19%)	4 (31%)
True Negatives	6 (29%)	10 (48%)	7 (33%)	17 (81%)	9 (69%)
False Negatives	1 (4%)	2 (9%)	4 (17%)	16 (70%)	11 (73%)

A comparison (II)

Vulnerable Function	Flawfinder		ITS4		RATS		Splint		BOON	
	True	False	True	False	True	False	True	False	True	False
gets()	1	-	1	-	1	-	1	-	1	-
scanf()	1	0	1	0	1	1	0	0	0	0
fscanf()	1	0	1	0	1	1	0	0	0	0
sscanf()	1	0	1	0	1	1	0	0	0	0
vscanf()	1	0	1	0	1	1	0	0	0	0
vsscanf()	1	0	1	0	1	1	0	0	0	0
vfscanf()	1	0	1	0	1	1	0	0	0	0
cuserid()	0	-	1	-	1	-	0	-	0	-
sprintf()	1	1	1	0	1	1	0	0	1	1
strcat()	1	1	1	1	1	1	1	0	1	1
strcpy()	1	1	1	1	1	1	1	0	1	1
streadd()	1	1	1	1	1	0	0	0	0	0
strecpy()	1	1	1	1	1	0	0	0	0	0
vsprintf()	1	1	1	0	1	1	1	1	0	0
strtrns()	1	1	1	1	1	0	0	0	0	0
printf()	1	1	1	1	1	1	1	1	-	-
fprintf()	1	1	1	1	1	1	1	1	-	-
sprintf()	1	1	1	1	1	1	1	1	-	-
snprintf()	1	1	1	1	0	0	0	0	-	-
vprintf()	1	1	0	0	0	0	0	0	-	-
vfprintf()	1	1	0	0	0	0	0	0	-	-
vsprintf()	1	1	1	1	1	1	0	0	-	-
vsnprintf()	1	1	1	1	0	0	0	0	-	-

Dealing with false positives

- It is a given that there will always be a certain ratio of false positives
- Possibilities to handle this problem:
  - Improving the tools to lower the ratio
  - Giving up soundness (danger: false negatives)
  - Weighting the results

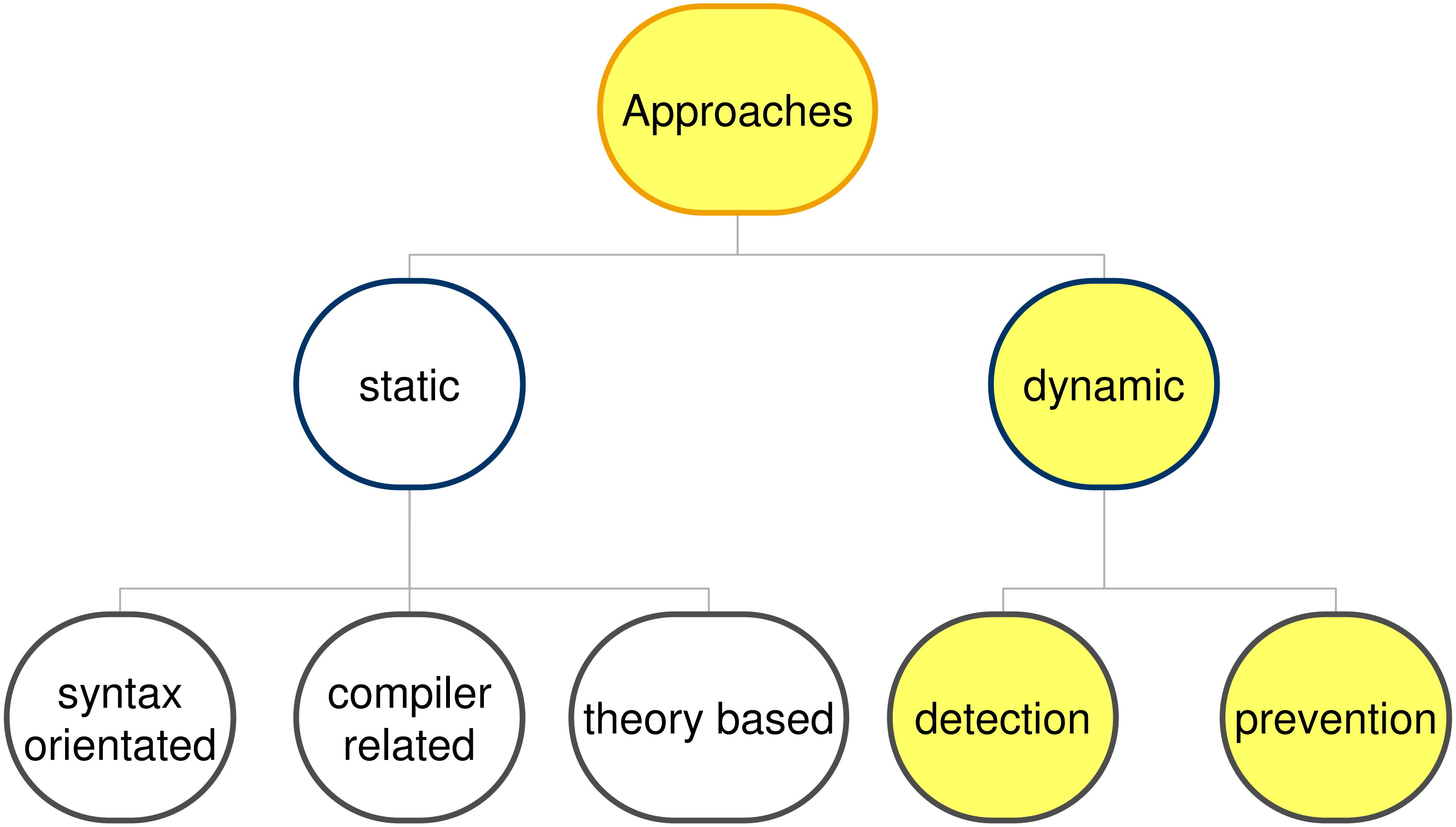
Frontline functions

FLF: Front Line Functions (2003)

- **Hypothesis: The closer a function is to (user) input, the more likely it contains an exploitable vulnerability**

- **The program is examined:**
  - **Functions that receive user input are labeled as *Inputs***
  - **Functions with potential vulnerabilities are labeled as *Targets***
  - **Furthermore a call graph of the program is generated**
- **To weight a given Target, the *FLF density*** k **for each Input is calculated**
  - k = p/m **with**
    p = maximal number of functions on the call graph between the Input and the Target **and**
    m = total number of functions in the program
  - **The largest** k **is chosen as the FLF density**

A classification



Dynamic approaches

Dynamic approaches

- Dynamic approaches work on runtime
- During program execution buffer overflows are detected or prevented
- To achieve this goal extra code is included in the watched program
  - Before compilation: Additional C code is added to the source
  - During compilation:
    - The compilation process is altered
    - Certain statement or mechanisms get translated differently (compared to standard compilation)

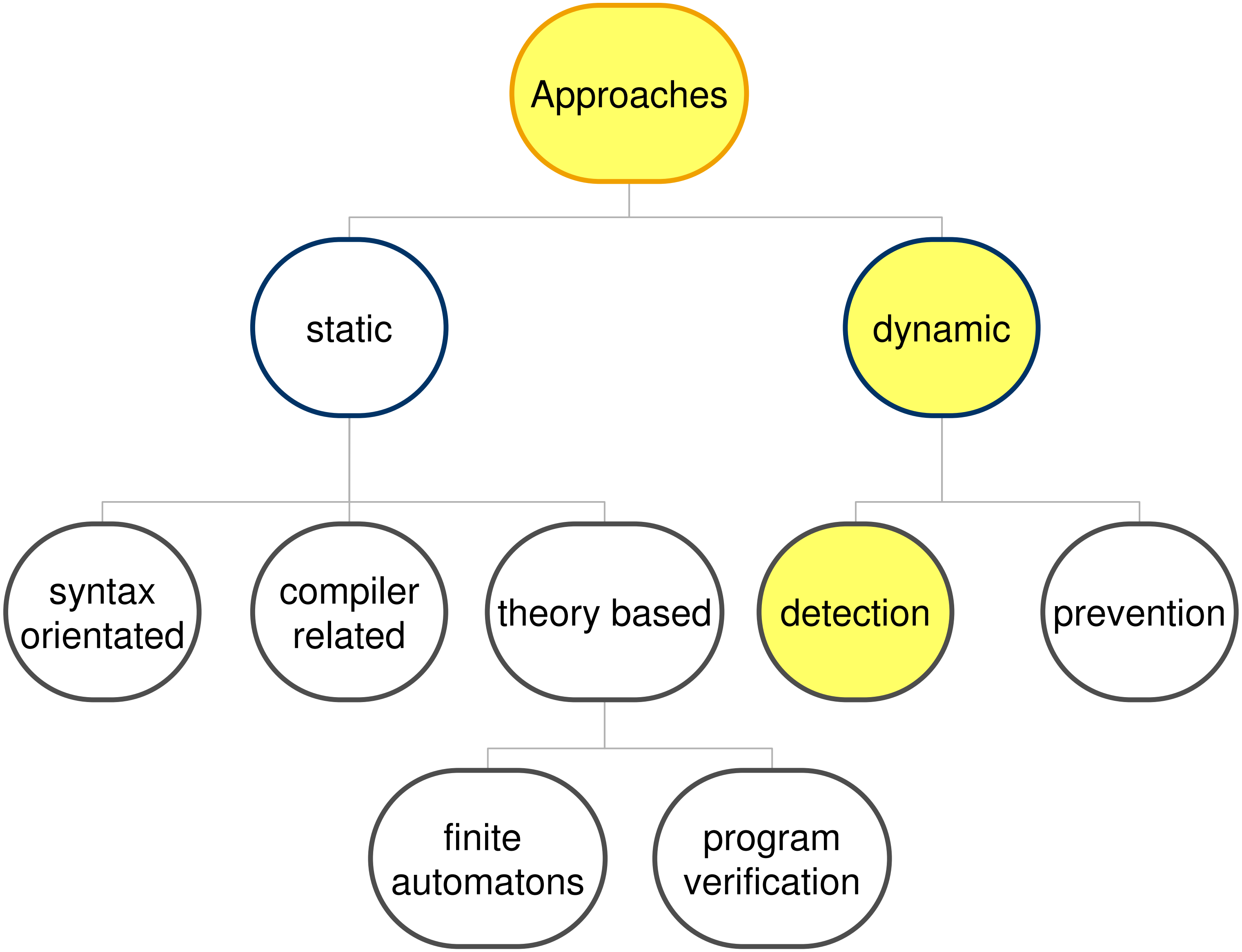
Note:

- In this presentation only tools that check for (stack) buffer overflows are discussed
- Approaches that look for format string exploits or heap corruption exist as well
- (check out Yves Younan's talk)

dynamic versus static approaches

- Sophisticated static tools try to determine runtime conditions of programs before execution
  - Control flow
  - Loop heuristics
  - Data flow
- Dynamic tools evaluate data that occurs when the program is actually run
  - Therefore the tools have no need for approximation/guessing of runtime behavior

A classification



Dynamic detection: STOBO

<u>S</u>ystematic <u>T</u>esting <u>O</u>f <u>B</u>uffer <u>O</u>verflows (2003)

- Dynamic extension of BOON's approach
- Tracks possible buffer length *during* execution
- The tool is supposed to accompany program testing

How it woks:

- The source code is altered before compilation
- The tool keeps track of all buffer sizes in a global table
  - For every variable declaration that declares a buffer a special function call is added
  - All functions and constructs that allocate buffer memory are wrapped
- All functions which could lead to buffer overflows are wrapped
- If a wrapped function detects a potential overflow a warning gets generated

Dynamic detection: STOBO (II)

Source code

```
char buf[100];

char *ptr;
ptr = malloc(20);

strcpy(ptr,buf);
```

STOBO output

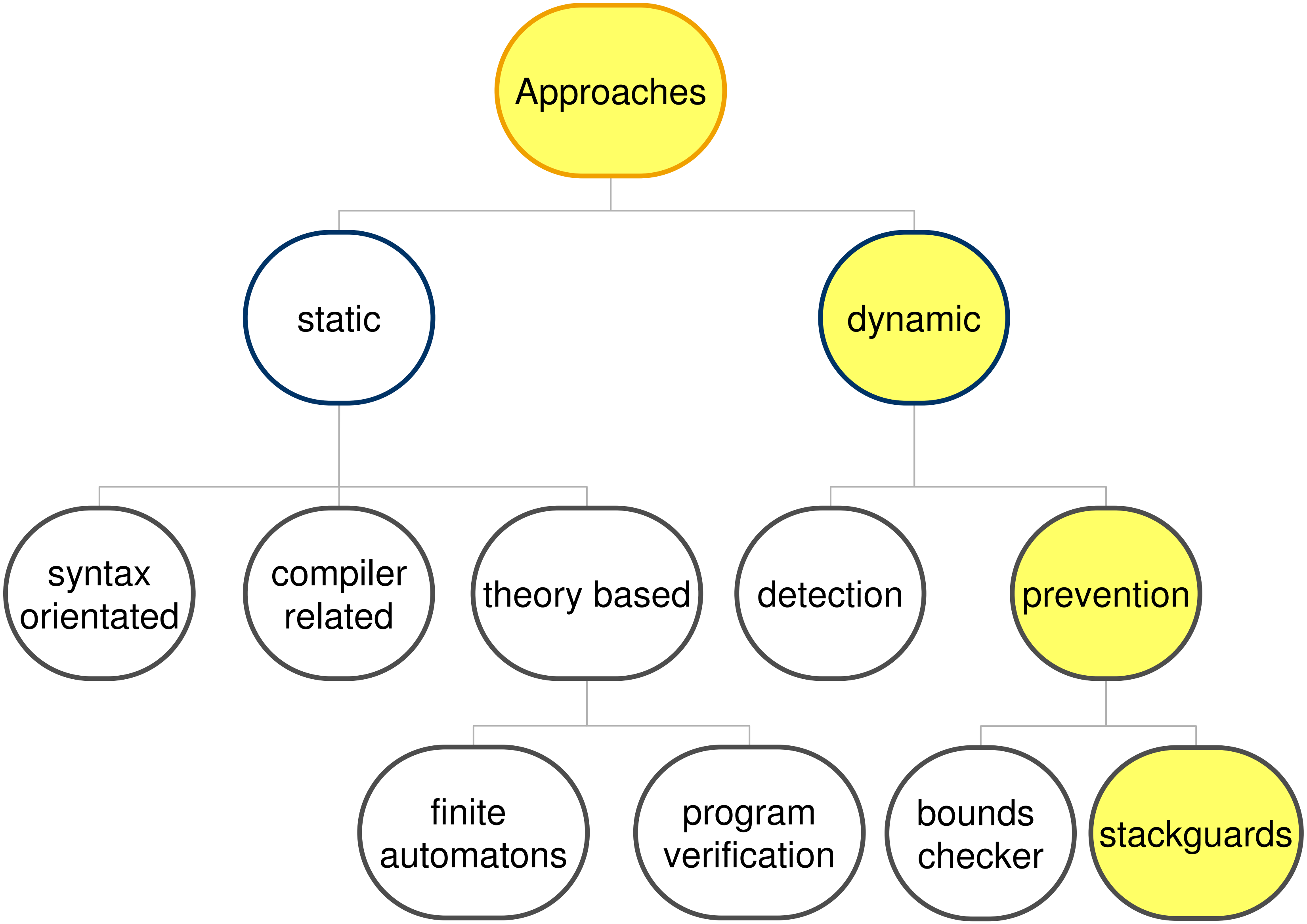
```
char buf[100];
__STOBO_stack_buf(buf,
                    sizeof(buf));

char *ptr;
ptr =
__STOBO_const_mem_malloc(20);

__STOBO_strcpy(ptr,buf);
```

STOBO is able to notice the possible buffer overflow

A classification



Stack protectors

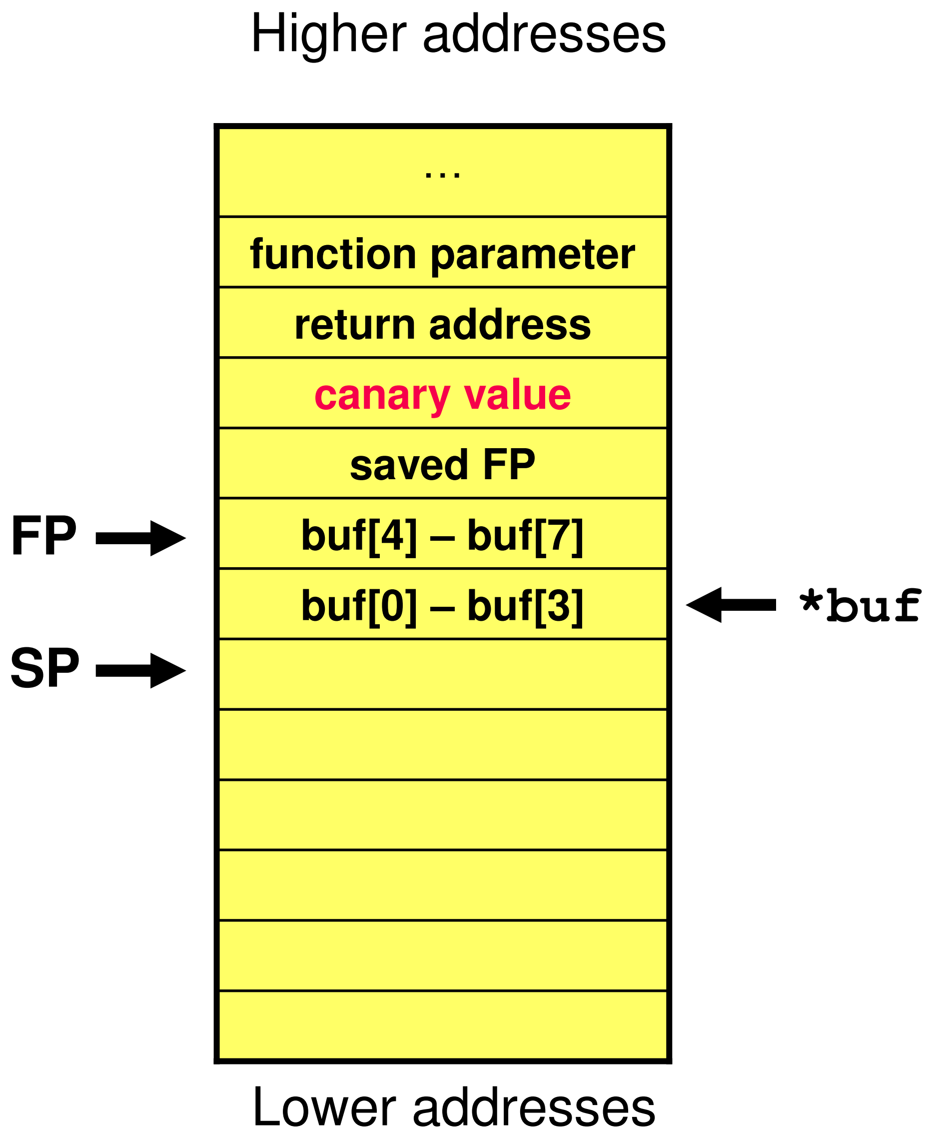
- Stack protectors try to prevent exploitation by altering the underlining program semantics
  - Enhanced function-prolog and -epilogue
  - Separate stack for return addresses
  - Reordering of local variables

Stack protectors: Canaries

Protection of the return address through a canary value

- **The function prologue and epilogue get enhanced on compile time**
- **The function prologue adds a canary value on the stack before the return address**
- **If a vulnerable buffer is overflown in order to overwrite the return address, the canary value is overwritten as well**
- **The function epilogue checks, if the canary value is unaltered, before the return from the function is excuted**

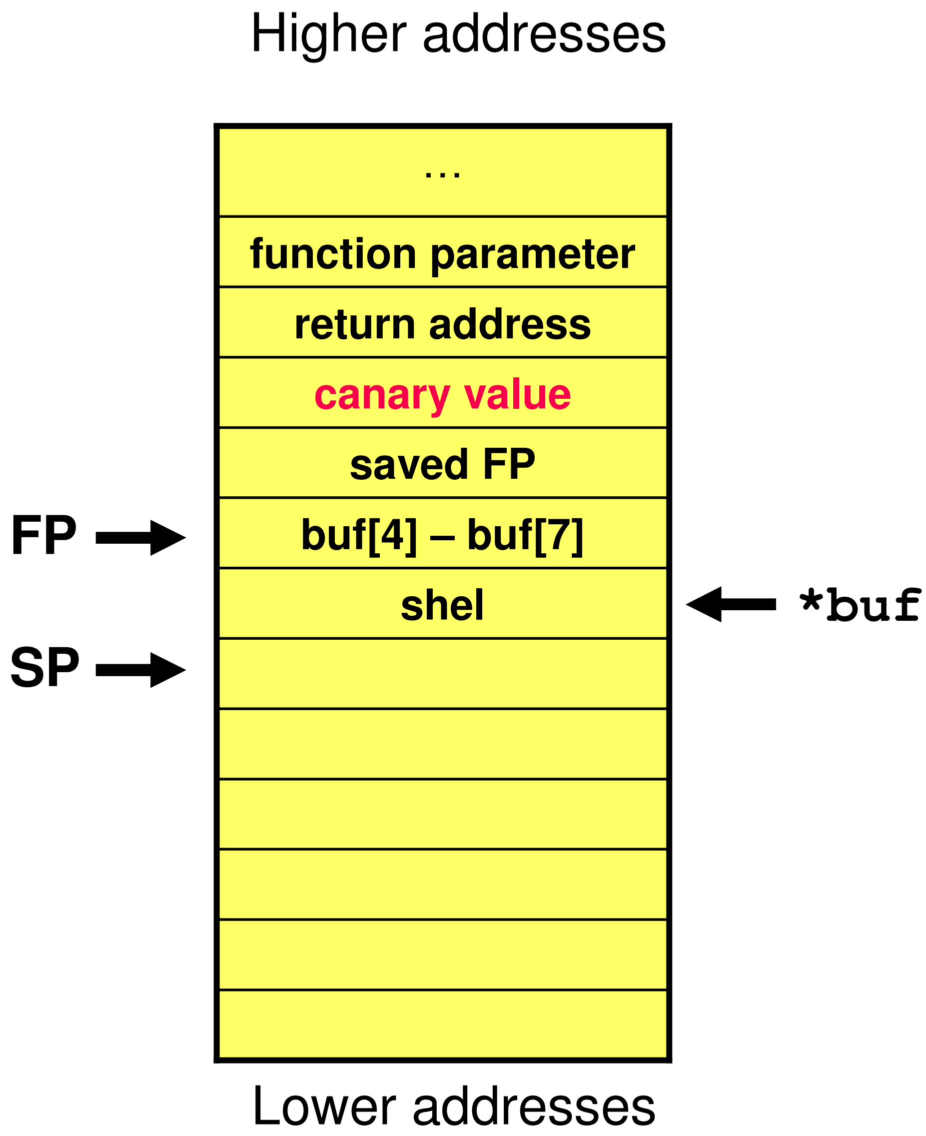
Stack protectors: Canaries (II)




```
int something(int para){
   char buf[8];
   ...
   strcpy(buf,
   “shellcodeshellcode”);

}
```

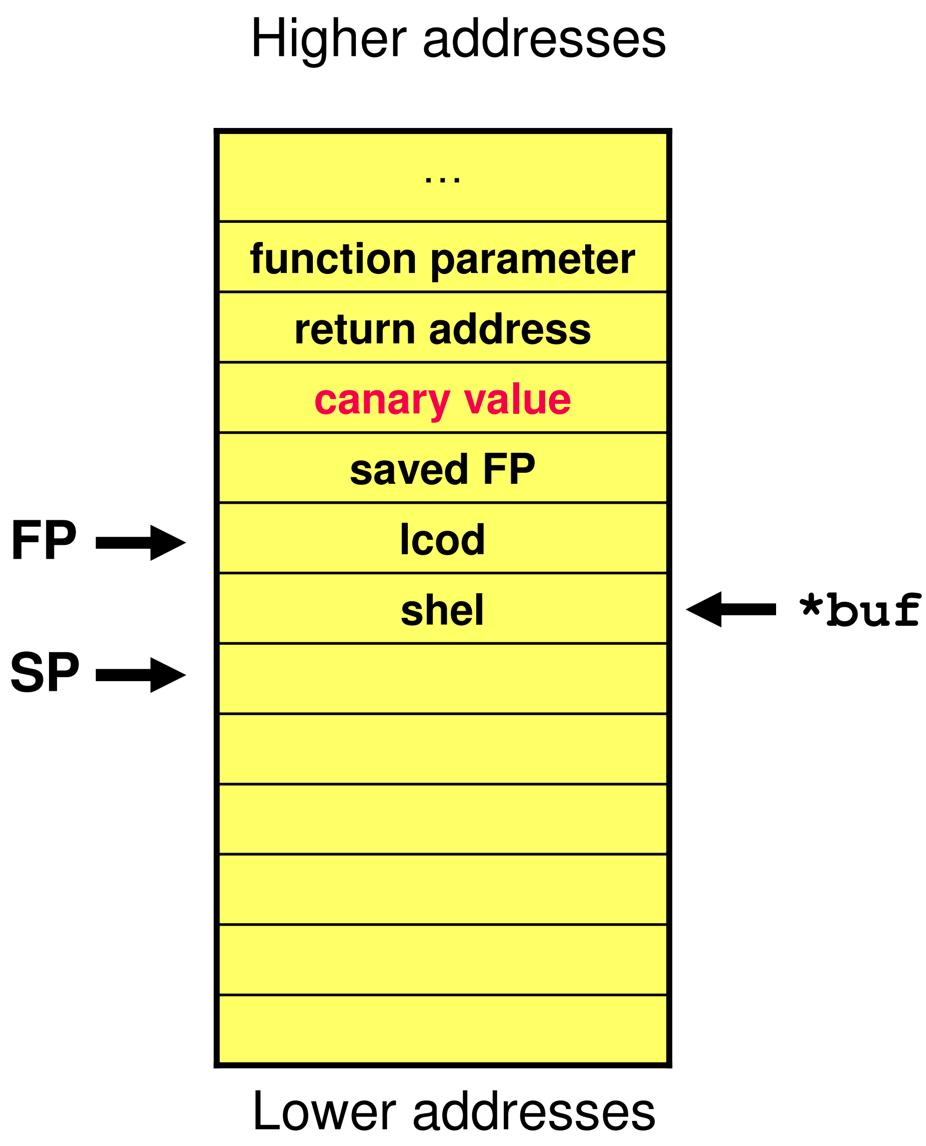
Stack protectors: Canaries (II)




```
int something(int para){
  char buf[8];
  ...
  strcpy(buf,
  "shellcodeshellcode");

}
```

Stack protectors: Canaries (II)

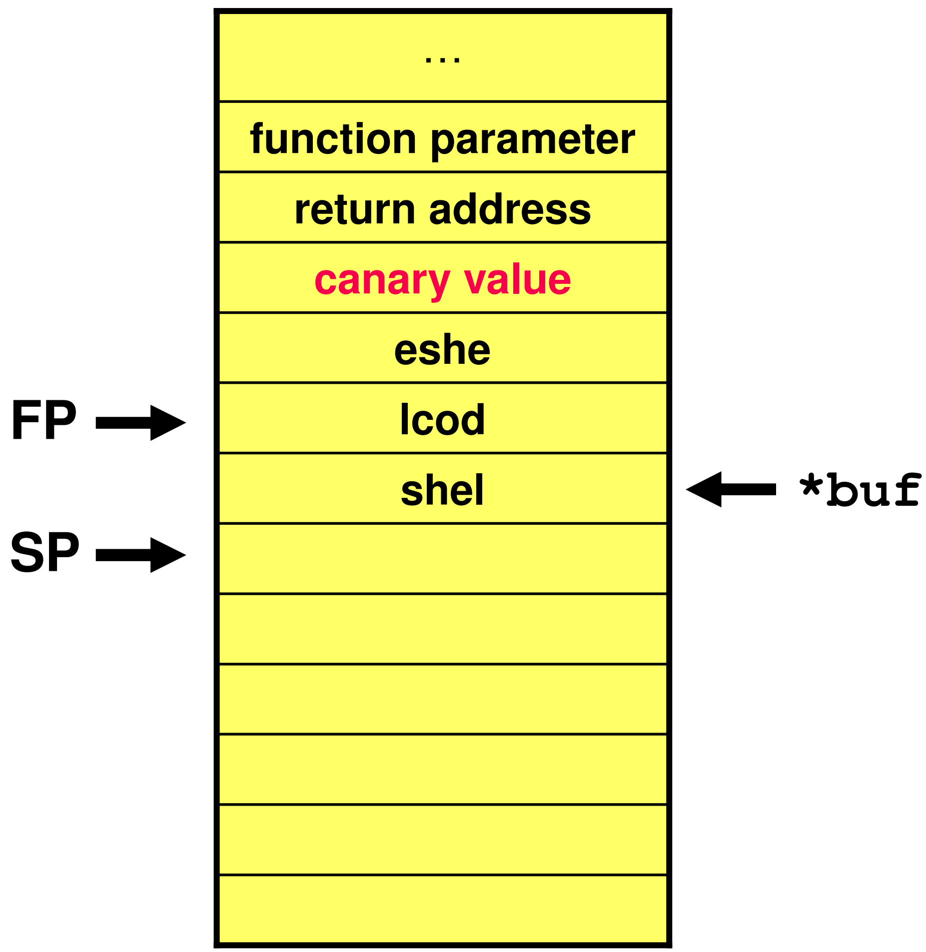



```
int something(int para){
  char buf[8];
  ...
  strcpy(buf,
  "shellcodeshellcode");

}
```

Stack protectors: Canaries (II)

Higher addresses

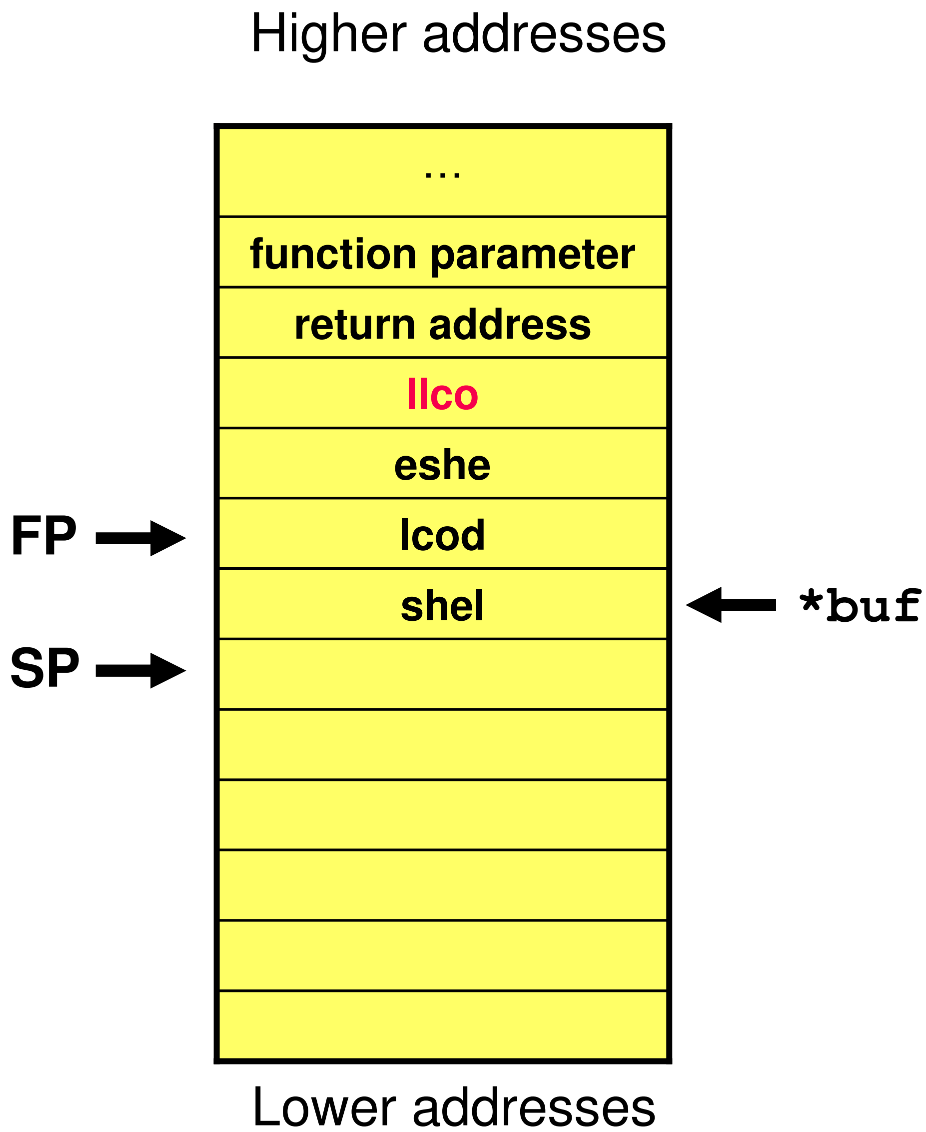


Lower addresses

```
int something(int para){
   char buf[8];
   ...
   strcpy(buf,
   “shellcodeshellcode”);

}
```

Stack protectors: Canaries (II)




```
int something(int para){
   char buf[8];
   ...
   strcpy(buf,
   "shellcodeshellcode");

}
```

- **The canary value gets overwritten.**
- **The attack is detected**

Stack protectors: Canaries (III)

Types of canaries:

- **Random canary**
- **Random XOR canary**
- **Null-canary** `(“\0\0\0\0”)`
- **Terminator canary** `(“\0\n\r\ff”)`

Limitations

- **Stack canaries protect only against Buffer Overflow that try to overwrite the return address**
- **No protection against**
  - **Heap overflows**
  - **Formatstring exploits**
  - **Function pointer overwriting**
  - **Alteration of local variables**

Stack protectors: Canaries (IV)

Tools that use stack canaries:

- **StackGuard**
- **Microsoft VisualStudio.NET**
- **ProPolice/GCC (with some enhancements)**

Additional problem

- **When a canary violation is detected, the program usually exits**
- **This turns the vulnerability into a denial of service opportunity**

Stack protectors: ProPolice SSP

ProPolice Stack-Smashing Protection (2000) by IBM Research

- **Uses stack canary values**
- **Additionally ProPolice reorders the values on the stack**

…
function arguments
return address
saved frame pointer
guard value
local buffers
local variables and pointers
…

Stack protectors: ProPolice SSP (II)

- **Vulnerable function arguments are protected through local copies**

```
void bar(void (*func)()){
  char buf[128];
  ...


  strcpy(buf,getenv("HOME"));
  (*func)();
}
```

```
void bar(void (*func)()){
  char buf[128];
  void (*local_func)();
  local_func = func;
  ...
  strcpy(buf,getenv("HOME"));
  (*local_func)();
}
```

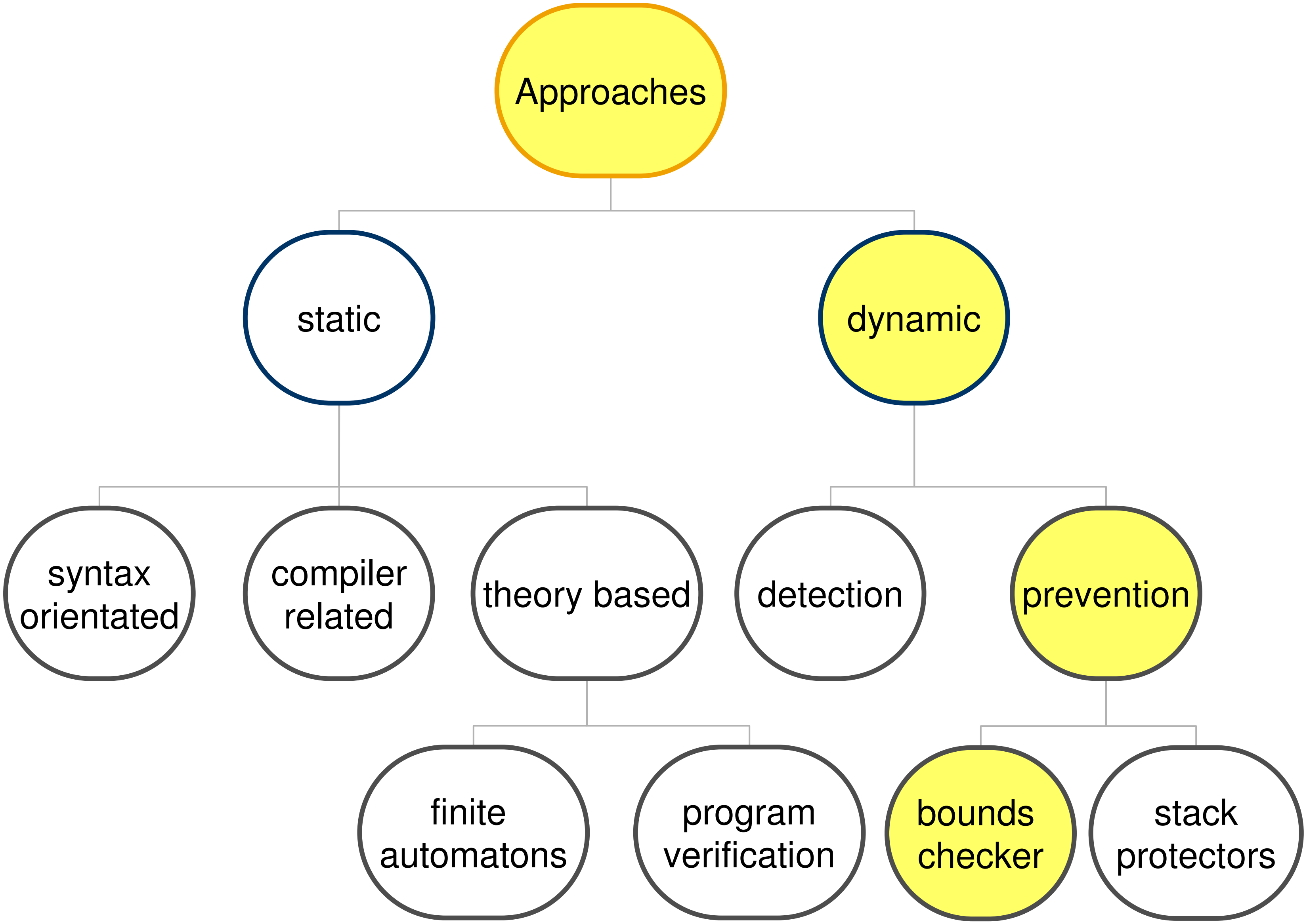
Disadvantages:
- **No reordering of struct-elements**
- **No protection of pointer arrays**

Stack protectors: Stack Shield

Stack Shield (2001)

- Has two options for return address protection
- Global Ret Stack
  - An additional global stack containing the return address is maintained
  - In the function prologue the return address is written on the global stack
  - In the function epilogue the saved return address is copied back onto the stack
  - → No detection of stack smashes, the program continues to run
- Ret Range Check
  - Using a global variable, the beginning address of the .data segment is calculated
  - Every return value is compare with this address
  - If the return address is smaller it points to the .text segment and is therefore probably safe
  - This approach is vulnerable against return-to-libC attacks

A classification

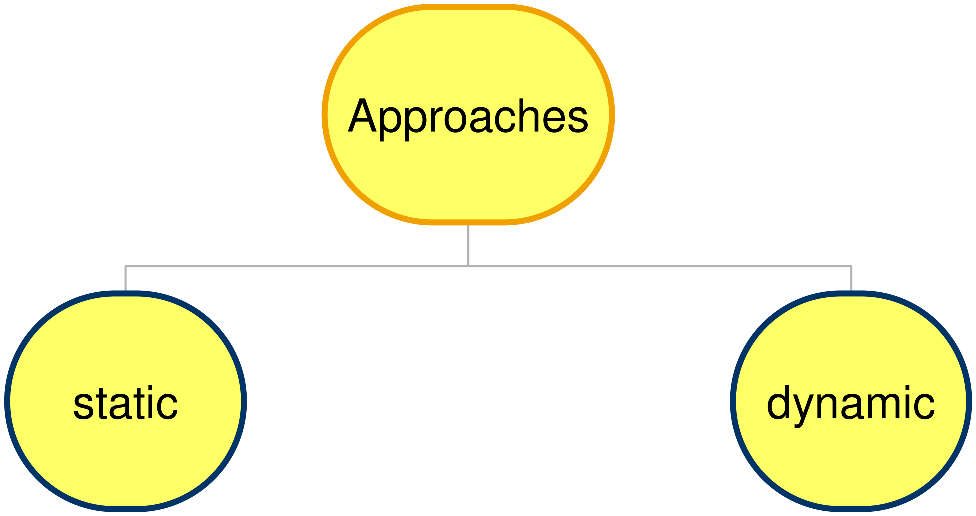


Bounds checker

CRED: C RangE Detector (2004)

- **CRED uses an global object table to track memory regions**
- **For every memory reference that is created during program execution an entry in the object table is created**
- **An entry consists of a base address and the amount of allocated memory**
- **Whenever a pointer is dereferenced, the object table is used to determine, if the access is correct**
- **To achieve this** `malloc()`**,** `free()` **and code that dereferences pointers is replaced**
- **Out of bounds pointers are legal, as long they aren't dereferenced**
- **With this method all memory regions are protected against overflows, not only stack buffers**
- **Memory regions that were allocated by external libraries aren't protected**
- **The (experimental) determined runtime overhead is between 26% and 150%**

A classification

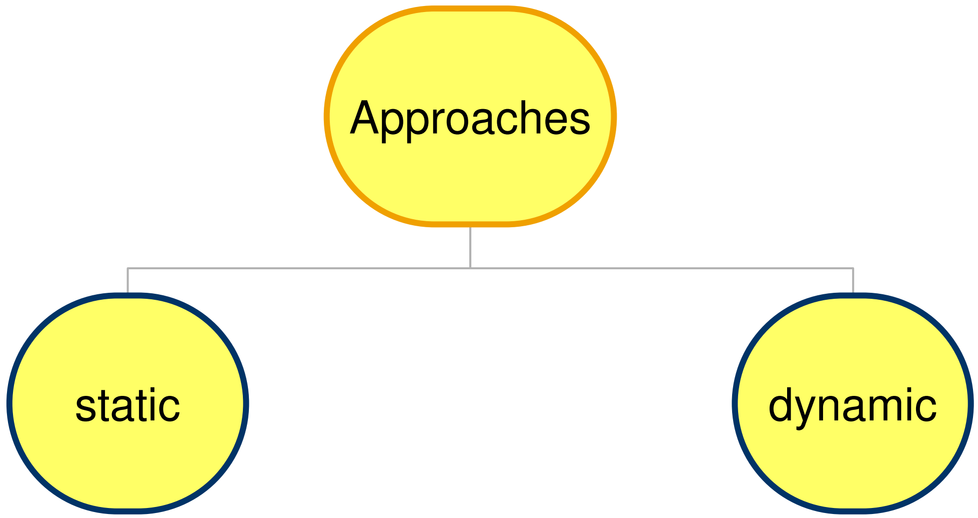


Combining static and dynamic techniques

Combining static and dynamic approaches

CCured (2002)

- **Combines type inference and runtime checking**
- **Pointers are separated according to their usage**
  - **SEQ: Sequence pointers – pointer may be subject to pointer arithmetic**
  - **SAFE: Pointer remains unaltered on program execution**
  - **DYNAMIC: Pointer may be subject to type casting**
- **Code handling “unsafe” pointers is instrumented with run time checks**



Conclusion

Conclusion

- Current limitations are unknown
  - Which kind of vulnerabilities are detectable today?
  - Which kind of vulnerabilities are still not covered?
- What is the actual ratio between real vulnerabilities and false positives
  - How should be dealt with potential false positives?
- Static tools
  - Ignorance of C++ (and/or object orientated programming)
  - Narrow focus (bounds checking, format strings, toctou,…)
  - To check for all potential security flaws, we would have to combine different approaches
    → More false positives
- Dynamic tools
  - Only defend against known attack vectors
  - If new/different ways of exploitation are discovered, many dynamic tools are helpless

The end

Thanks for listening…

Bibliography (static tools)

- **BOON:** D.Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of NDSS 2000*, 2000. http://www.cs.berkeley.edu/~daw/papers/overruns-ndss00.ps
- **CQUAL:** U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In Proceedings of the 10th USENIX Security Symposium, 2001.
  http://www.cs.berkeley.edu/~ushankar/research/percents/percents.pdf
- **Flawfinder:** David A. Wheeler. Flawnder. Web page http://www.dwheeler.com/flawfinder/, May 2001.
- **ITS4:** John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In Proceedings of the 16th Annual Computer Security Applications Conference, December 2000
  http://www.cigital.com/papers/download/its4.pdf
- **MOPS**: H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 235–244, November 2002.
  http://www.cs.berkeley.edu/~daw/papers/mops-ccs02.ps
- **Splint:** D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilites. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.
  http://lclint.cs.virginia.edu/usenix01.pdf
- **XGCC:** Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific static analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69--82, Berlin, Germany, June 2002
  http://www.stanford.edu/~engler/p27hallem.ps

Bibliography (dynamic tools)

- **STOBO:** E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," *Proceedings of the 2003 Symposium on Networked and Distributed System Security* (Feb. 2003). http://nob.cs.ucdavis.edu/~bishop/papers/2003-bufov/index.html
- **CRED**: O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In Proceedings of the Network and Distributed System Security (NDSS) Symposium, pages 159--169, February 2004. http://suif.stanford.edu/papers/tunji04.pdf
- **StackGuard**: C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proceedings of the 7th USENIX Security Symposium, pages 63--78, San Antonio, TX, January 1998. http://citeseer.ist.psu.edu/cowan98stackguard.html
- **StackShield**: http://www.angelfire.com/sk/stackshield/
- **ProPolice SSP**: http://www.research.ibm.com/trl/projects/security/ssp/
- **CCured**: G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In Twenty-Ninth ACM Symposium on Principles of Programming Languages, Portland, OR, Jan. 2002. http://manju.cs.berkeley.edu/ccured/popl02.pdf

Bibliography (Comparisons)

- J. Wilander, M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, San Diego, CA, February 2003, pp. 149-162 http://www.ida.liu.se/~johwi/research_publications/paper_ndss2003_john_wilander.pdf
- J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion detection. In *Proceedings of the Nordic Workshop on Secure IT Systems*, pages 68--84, November 2002. http://www.ida.liu.se/~johwi/research_publications/paper_nordsec2002_john_wilander.pdf
- Peter Silberman, Richard Johnson (iDefense). A Comparison of Buffer Overflow Prevention Implementations and Weaknesses. *Black Hat USA 2004 Briefings & Training,* 2004 http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf
- Yves Younan, Wouter Joosen and Frank Piessens. Code injection in C and C++: A Survey of Vulnerabilities and Countermeasures. *Technical Report CW386*, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004. http://fort-knox.org/CW386.pdf