Software Security

Tools for code security analysis

Most software contain security vulnerabilities
Strong need for tools allowing to:
- detect potential vulnerabilities
- help to evaluate their exploitability / dangerousness

so, several classes of tools:

3 code analysis technique:
- fuzzing (AFL++)
- symbol execution (KLEE / PathCrawler)
- static analysis (Frama-C)

Syntactic vs Semantic

syntactic: check compliance to coding rules / standards
semantic: check for behavioral inconsistencies

Static vs Dynamic

static: check are performed at "compile time" (no concrete code execution)
dynamic: on-line and / or offline checks require code execution steps

Black vs Grey vs White Box

black box: no access required to the target code
white box: full access required to (source ?) target code
grey box: partial access required to the target code

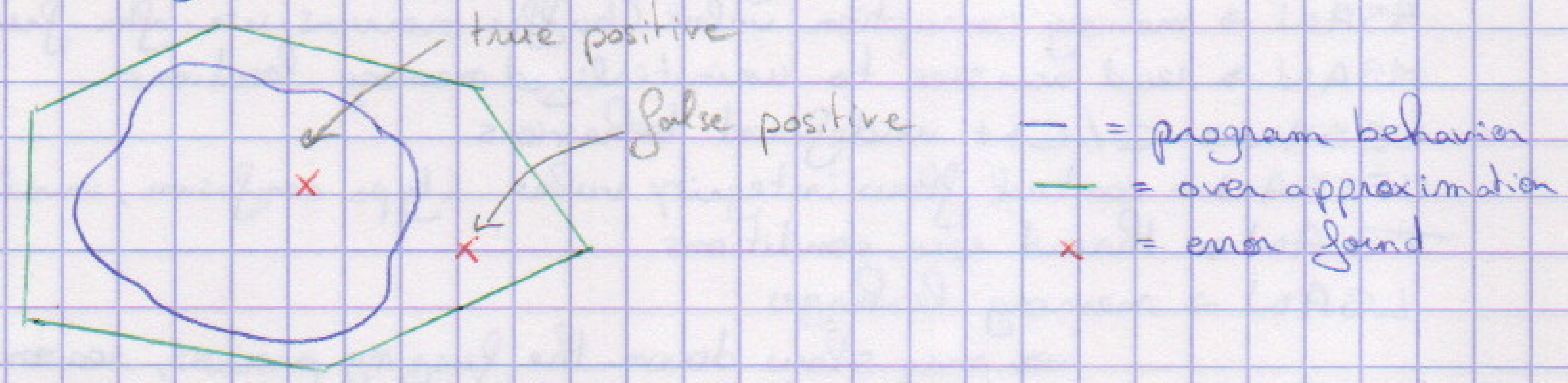
Taking into account the limits of computability, no hope to get a fully automated (powerful) tool

Possible work-arounds:

Approximate enough the program behavior to make the analysis decidable ⇒ the results may be no longuer sound no complete

Use a semi algorithm
if the analysis terminates, then it gives sound and complete results.

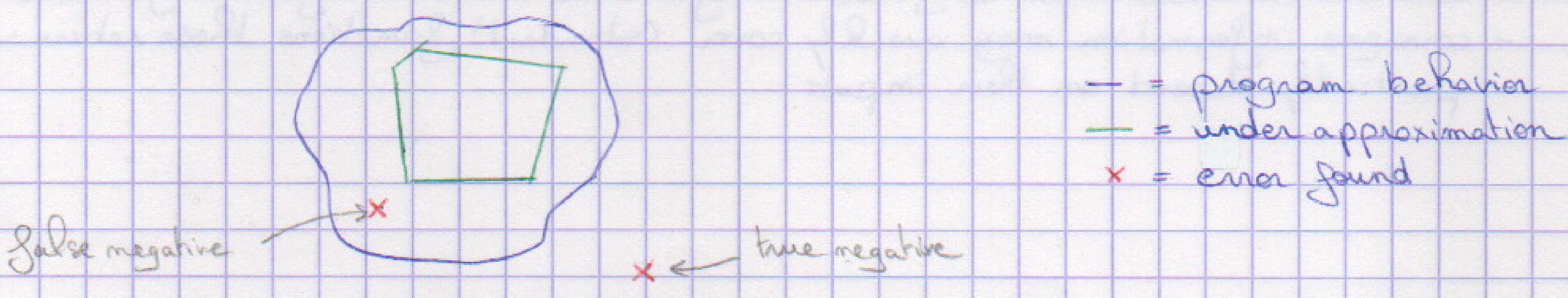
Over-approximation of the program behavior



Sound: "correct" verdicts are always reliable (never miss an "incorrect" execution)
Not complete: may reject correct programs

Possible false positive → Frama-C

Under-approximation of the program behavior



Unsound: "correct" verdicts are not reliable (may miss "incorrect" executions)
Complete: never reject correct programs ("incorrect" execution reported are real ones)

Dynamic approaches: fuzzing / (dynamic) symbolic execution
Static approaches: value set analysis / code pattern based vulnerability detection.

Fuzzing ← vulnerability detection

→ Run the program in order to detect "unsecure behaviors" (from simple crashes to complex security property violations)

black box vs white box fuzzing, public vs unknown input format...

* **Random** (or brute-force or blind) **fuzzing**

pros: very efficient generation scheme! / no initial knowledge required / pure black box

cons: no control over the execution sequences produced... / easily stuck by checksums, robust parsers

* **Grammar-based fuzzing** ← drive the input generation using a grammar

pros: may cover complex input domains (file format, protocol)
may overcome checksums and first-level parsing barriers

cons: required some knowledge about the nominal pgm inputs (publicly available...)
how much "unexpected" are the input produced?

* **Genetic-based fuzzing** ← use a fitness function to measure execution "relevance"

pros: a mix between random and controled fuzzing
still an efficient generation scheme

cons: needs to design a good fitness function w.r.t. intended objective (coverage, pattern oriented, property oriented, ...)
some code instrumention usually required (for the fitness function)
may still be stuck by checksums, robust parsers, ...

→ Concrete fuzzer example: **AFL++** (American Fuzzy Loop)

mutation strategy:

→ **deterministic** (sequentially): flip bits / add or substract small integers / insert known interesting integers → 0, 1, INT_MAX...)

→ **non-deterministic**: insertion, deletion, arithmetics...

take a file with possible inputs as input to start the fuzzing

useful metrics:
- queue: test cases for every distinctive execution path
- crash: (unique) test cases that produced a fatal signal
- hangs: (unique) test cases that caused a time out
- cycle: a complete "queue pass"
- stability: consistency of observered traces

AFL can be used in **conjonction with (dynamic) sanitizers**:

ASAN → memory corruption vulns (buffer overruns, use-after-free, ...)
MSAN → read accesses to uninitialized memory locations
UBSAN → C/C++ undefined behaviors
CFISAN → control flow integrity vulns (type confusion, invalid return addr.
TSAN → thread race conditions
LSAN → memory leakages

⇒ may slow down the fuzzing process, reasonnable trade-off!?

Limitations of AFL:
- may only fuzz standalone applications (no libraries, APIs or even parts of code)
- fuzzer inputs should be provided from a single input file
- a (new) external input is generated for each execution of the target code
- coverage information may weakly cover (standard) functions those return values partially depend on their inputs

Symbolic execution

run a program paths (as in test execution) but mapping variables to symbolic values (instead of concrete ones)
- each symbolic execution allows to reason on a set of concrete executions (all the ones following the same path in the CFG)
- allow to decide if a CFG path is feasible or not (and with which input values?)
- allow to explore a finite set of paths in the CFG

CFG = Control Flow Graph

Associate a path predicate φ_σ to each σ of the CFG:
$(\exists$ a variable valuation v s.t. $v \models \varphi_\sigma) \iff (v$ covers $\sigma)$
(φ_σ is the conjunction of all boolean conditions associated to σ in the CFG)
- solving φ_σ indicates if σ is feasible
- iterate over a finite subset of the CFG paths...

In practice: express φ_σ in a decidable logic fragment

What can we do if:
- the path predicate cannot be expressed in a decidable logic? (non linear operations)
- the program contains conditions on non-reversible functions? (if (x = hash(y)))
- part of the program code is not available (library function)

⇒ combine symbolic and concrete executions: concolic execution (or Dynamic Symbolic Exec) DSE

⇒ Trade-off between:
- tractability: keep decidable decision procedures over path predicates
- scalability: concrete execution faster than symbolic reasoning
- completness: concretization ⇒ loss of execution paths

DSE for vulnerability analysis
an effective and flexible test generation & execution technique
- can be used on "arbitrary" code (dynamic allocat°, complex math functions, binary code)
- trade-off between correctness, completeness and efficiency (ratio b/ symbolic & concrete values)
- can be used in a coverage-oriented (bug finding) or goal-oriented.

numerous existing tools
- source level: KLEE (C/C++), JPF (Java)
- binary level: Sage, Angr

However, not all problems solved:
- "path explosion" problem on large codes
- can be rather slow (compared with fuzzing)

PathCrawler

Comparaison entre AFL et KLEE:
Tout d'abord, AFL créé des inputs par mutation : si l'input créé provoque un crash, AFL continuera d'utiliser cet input comme seed. En revanche, si l'input ne provoque pas de crash pendant un certain temps, AFL changera de seed.
KLEE en revanche utilise l'exécution symbolique, il vérifie d'abord dans le programme les conditions pour générer des inputs qui passent par tous les chemins et peut-être rencontrer des erreurs.
Ensuite, KLEE apporte des erreurs qui n'ont pas les mêmes causes là où AFL nous donne toutes les erreurs qu'il rencontre, il est donc normal que nous ayons plus d'erreurs avec AFL.
Aussi, KLEE s'aide du fichier source pour générer des inputs qui provoqueront des crashs, là où AFL produit les inputs grâce à un fichier donné. Dans notre cas il a réussi à trouver les backspaces très vite malgré que nous ne lui ayons donné qu'un seul fichier texte avec un seul input "toto".
Par contre, KLEE créé beaucoup de dossiers (un pour chaque essai) et si il y a ~~un~~ souci de boucle infini dans le programme on peut vite avoir des overflows de mémoire, là où nous n'avons pas eu de problème avec AFL sur ce point.

Static Analysis ← over approximation ou under approximation du programme

Méthode pour examiner le code source d'un programme (pas d'exécutions). Objectif : repérer des problèmes avant même que le programme ne soit exécuté, afin de réduire les coûts et les risques liés aux bugs ou aux vulnérabilités en production.

→ non terminating analysis → if the analysis terminates, then the result is sound and complete

To don't test on every iteration of a loop

↳ over-approximate the program behavior

1. propagate an abstract state
2. safely merge memory abstract states produced from ≠ paths
3. make loop iterations always finite

→ accuracy vs scalability trade-offs

widening : on voit que x augmente, au lieu de le remplacer par [1, 1], on remplace [1, +∞[
→ permet d'éviter 1000 calculs

narrowing : ramener un nombre infini à la limite supérieur

EVA = Evolved Value Analysis

→ to detect illegal memory access / array overflows / uninitialized values.
provide an approximation of possible values for each variable at any program point
⟹ may signals false positive ← à cause des approximations.

RTE = Runtime Errors

→ to detect common runtime errors ⇒ Division by 0 / integer overflows / array overflow or illegal accesses / Deferencing null pointers.

⇒ needs to be combined with other plugins (WP or EVA) to prove inserted assertions
⇒ May increase code complexity with added assertions.

WP = Weakest Preconditions

→ formally prove properties or behaviors of the program such as
- a function always return a valid result
- no overflows or illegal acceses are possible
- a given ACSL specification is respected. (guarantee the functional correctness of the program)

⇒ Can be challenging to use on complex or poorly structured pgm.
⇒ Some cases may not be provable automatically