IV113 Validace a verifikace

Testování se znalostí kódu, automatizace a symbolická exekuce

Jiří Barnat

Funkční testování
(Unit testing)

Funkční testování

Princip techniky

- Oddělené testování malých částí kódu na úrovni vnitřních rozhraní (API).
- Testování jednotlivých funkcí bez využití znalosti jejich implementace (interní black-box testing).
- Vyžaduje modulární kód.

Základní cíl

- Prověřit, že isolované funkce systému fungují správně.
- Odladěná podřešení se lépe kombinují do funkčního celku.

Globální postup

- Identifikují se vnitřní funkce produktu.
- Pro každou identifikovanou funkci se vytvoří test.

Testování funkcí systému v isolaci

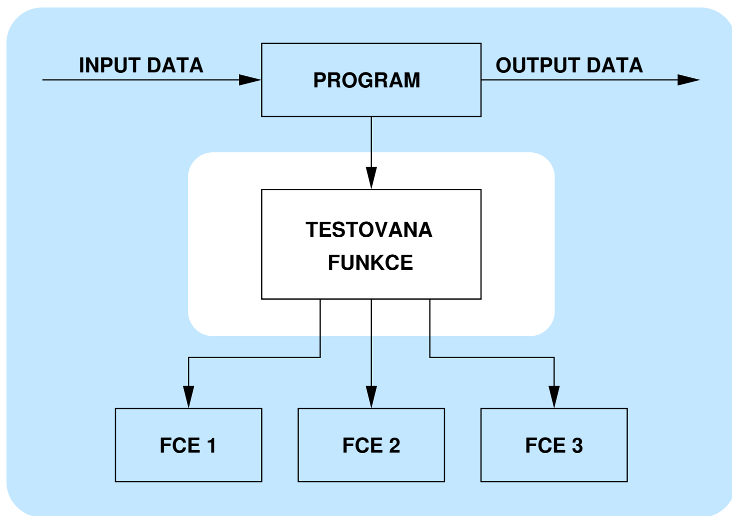
Testovaní isolované funkce

- Funkce isolovaná od systému není samostatně spustitelná.
- Je třeba vyvinout software, který umožní testování funkce.
- **Implementace testovacího prostředí probíhá jako součást implementace funkce.**

Testovací prostředí

- Vzhledem k testované funkci má vnější a vnitřní část.
- **Vnější část** slouží jako hlavní, samostatně spustitelný program, který volá testovanou funkci s danými parametry, sbírá a tiskne relevantní výsledky volání funkce.
- **Vnitřní část** simuluje chování dalších funkcí volaných testovanou funkcí.

Obecné schéma testovací prostředí funkce



Funkční testování – frameworky

Podpora pro funkční testování

- **Java**: jUnit, TestNG, qc4j
- **C++**: CUnit, TUT, QuickCheck++
- **Haskell**: QuickCheck, SmallCheck, HUnit
- **Python**: PyUnit, nose, qc
- **Ruby**: Test::Unit, RushCheck
- ...

Rozcestník

`http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`

Co je to funkce a kde je získat

Funkce

- Funkce je něco, co program může dělat.
- "Features", schopnosti, entity identifikované svými schopnostmi, ...

Identifikace funkcí

- Ze specifikace, či manuálu.
- Z uživatelského rozhraní.
- Z nápovědy v GUI/TUI.
- Prohledáním zdrojového kódu (názvy členských funkcí tříd, texty chybových hlášek, ...).

Seznam funkcí a popis funkcí

Seznam funkcí

- Základní dokument funkčního testování.

Informace obsažené v seznamu funkcí

- Kategorizace funkce, tj. označení skupiny funkcí s podobnou, či související funkcionalitou.
- Vstupy funkce
  - Maximum/Minimum, hraniční případy.
  - Speciální případy
- Výstupy funkce
- Rozsah působnosti funkce (není-li dána kategorizací).
- Možnosti/volby (options) funkce.
- Okolnosti, za kterých se funkce chová odlišně (globální konfigurace programu, verze a typ OS, ...)

Testování funkcí

Orákulum

- Je nutné vědět (nebo mít určeno) jak se pozná, že daný test dané funkce uspěl.
- Orákulum, může být součástí seznamu funkcí.

Neúplnost testování

- Není-li možné otestovat všechny vstupy, je vhodné použít princip doménového testování.

Konfigurace systému a vliv prostředí

- Testy funkce je vhodné opakovat v potenciálně různých podmínkách, při nichž je možné funkci využít.

Testování negativních případů

- Funkce by se měla testovat na to, že dělá to, co dělat má, ale i na to, že nedělá to, co dělat nemá.

Výhody funkčního testování

Pokrytí (coverage)

- Technika vhodná k realizaci úplného pokrytí testy.
- Vhodná technika k budování testovacího plánu, potažmo k měření míry splnění testovacího plánu.

Ranné testování

- Lze testovat už částečně vzniklý kód.
- Vede k rychlému odhalení mnoha chyb.
- Základem pro testy řízený vývoj produktu a další agilní metody vývoje.

Nevýhody funkčního testování

Rizika použití funkčního testování

- Pokud je v projektu přítomno, ochabuje potřeba realizovat jiné metody testování.
- Produkt, který je vystavěn z korektních funkcí, nemusí být korektní.

Nedostatky funkčního testování

- Neuvažuje interakci jednotlivých funkcí na stejné úrovni.
- S rostoucí integrací funkcí "ztrácí sílu".
- Nezachycuje chování funkcí v dlouhodobém běhu.
- Často se zaměřuje na testování schopnosti jako takové, ale ne na testování krajních případů.
- Neřeší otázky typu: Byla funkcí produktu naplněna uživatelova potřeba?

Regresní testování

Regresní testování

Princip techniky

- Opakování vybrané sady úspěšně proběhnuvších testů.

Hlavní důvod použití

- Riziko zanesení chyb změnou kódu.

Další uplatnění

- Potvrzení stability chování/výkonu produktu.
- Nástroj pro prokázání množství odvedené práce klientům.
- Psychologická podpora vývojářů.
  (Vývojáři mohou být odvážnější při změnách kódu.)

Prohřešky odhalené pomocí regresního testování

Oprava chyby je nedostatečná.

- Záplata je neúčinná.
- Záplata odstraní symptomy, ne však chybu samotnou.

Oprava chyby má vedlejší účinky.

- Výskyt nově zanesených chyb.
- Znovu vyvolání opravených chyb.

Produkt nelze sestavit.

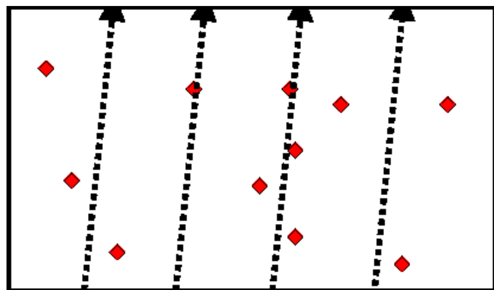
- Typicky ve spojení se systémem pro kontrolu verzí.

Regresní testování a detekce nových chyb

Detekce nových chyb

- Z principu nedetekuje nové chyby, až na ...
- ... chyby závislé na předchozím použití produktu.
- ... chyby, jež se projeví díky přítomnosti nového kódu.

Příklad – Analogie s minovým polem



Aspekty ovlivňující výběr testů

Podstata regresního testování je v opakování testů.

Ptáme se:

- Které testy mají být součástí sady?
- Jaký je důvod pro opakování právě těchto testů?
- Jak přesně se mají jednotlivé testy v sadě vyhodnocovat?

Pozorování

- Podobně jako testování na základě rizika, i regresní testování jde napříč předchozími technikami testování.

Různé pohledy na volbu sady testů

Procedurální

- Opakujeme vybranou, nadále stejnou, sadu stejně vyhodnocovaných testů stejným způsobem.

Ekonomické

- Opakujeme všechny snadno opakovatelné a vyhodnotitelné testy.

Zaměřené na snížení rizika

- Opakujeme existující testy, jež v minulosti odhalily chyby, a volíme testy, jež pokrývají kritické části produktu.

Podpora při vývoji produktu

- Volíme testy, které pomohou rozhodnout, zda je korektní/smysluplné modifikovat produkt navrženým způsobem (sledujeme např. výkon aplikace).

Pozorování

- Produkt se vyvíjí a spolu s ním je nutné vyvíjet i testy, které se mají znovu spustit.
- Udržovat testy v aktuálním spustitelném stavu může být nákladné.
- **Nebrání údržba starých testů ve vývoji nových testů pro dosud neotestované části produktu?**

Regresní testování a funkční testování

Pozorování

- Regresní testování se typicky realizuje pomocí funkčního testování (unit testing).

Důvody

- Ve větších projektech je typické, že programátor samotný tvoří testy pro část kódu, který vyvíjí.
- Automatizovatelné od raného stádia vývoje.

Rizika

- Po dokončení vývoje modulů, metoda postrádá smysl.
- Sadu testů je nutné obohacovat o integrační testy.
  - Unit testy si tvoří sami vývojáři, jakmile však dochází k integraci, je pro vytvoření testu nutná znalost všech integrovaných částí, což může být na rámec působnosti každého jednotlivého vývojáře.

Automatizované a autonomní testování

Automatizovaná procedura testování

- "Jedním příkazem" se spustí sada testů, ty se provedou, vyhodnotí a na výstupu se zobrazí statistiky, případně identifikují neúspěšné testy.

Autonomní procedura testování

- Spouští se bez explicitního příkazu testera/uživatele.
- Spouštěcí mechanizmy
  - časová periodicita (každou půlnoc)
  - událostmi řízené spouštění
    (commit do systému pro správu verzí)

Buildbot

Připomenutí

- Znovu-spuštění testů – **stroj**
- Vyhodnocení výsledků – **stroj**+člověk

⇒ Regresní testování

BuildBot

- Systém pro podporu automatické kompilace a testování.
- Umožňuje spouštění testů na různých platformách.
- `buildbot.net`

Jiná řešení

- travis-ci, cdash, tinderbox, ...
- `http://nixos.org/hydra/`, ...

Další techniky white-box testování

Testování s použitím modelu

Princip

- Modelování produktu jako konečného automatu.
- Odvozování vlastností a nutné množiny testů na základě modelu.

Motivace

- Přiblížení se formální verifikaci.
- Matematická garance vlastností modelu potažmo produktu samotného.
- Generování minimální množiny testů.

Problémy

- Náročnost budování věrného modelu.

Symbolická exekuce

Problém

- Detekovat chybu, která nastává pouze pro některé vstupy, je obtížné.
- Viz neúplnost testování.

Co bychom chtěli

- Testovat program na všechny možné vstupy.

Symbolická exekuce

Myšlenka

- Vykonávání programu, při němž jsou hodnoty vstupních proměnných označeny symboly a během výpočtu manipulovány symbolicky.

Příklad

Program	Vybrané konkrétní hodnoty	Symbolická reprezentace
`read(A)`		
	$A = 3$	$A = \alpha$
`A = A * 2`		
	$A = 6$	$A = \alpha * 2$
`A = A + 1`		
	$A = 7$	$A = (\alpha * 2) + 1$
`output(A)`		

Větvení v programu a podmínka cesty

Pozorování

- Větvení v kódu programu klade další omezení na možné hodnoty symbolických vstupů.

Příklad

```
1 if (A == 2)
2    then ...
3    else ...
```

$A = (\alpha * 2) + 1$

$(\alpha * 2) + 1 = 2$

$(\alpha * 2) + 1 \neq 2$

Podmínka cesty (Path condition)

- Formule nad symboly označující vstupní hodnoty.
- Kóduje historii výpočtu, tj. kumuluje omezení jež vyplynula z podmínek v místech větvení programu během výpočtu (z počátečního až do aktuálního bodu).
- Iniciálně prázdná (true).

Nerealizovatelné cesty

Pozorování

- Podmínka cesty může být nesplnitelná.
- Tento jev indikuje nerealizovatelnost průchodu programem asociovaného s danou cestou.

Příklad 1

	Kód		
1	`if (A == B)`	$A = \alpha, B = \beta$	
2	`  then`	$\alpha = \beta$	
3	`    if (A == B)`		
4	`      then ...`	$\alpha = \beta \wedge \alpha = \beta$	
5	`      ~~else ...~~`	$\alpha = \beta \wedge \alpha \neq \beta$	*is UNSAT*
6	`  else ...`	$\alpha \neq \beta$	

Příklad 2

% – operace modulo

	Kód		
1	`A=A%2`	$A = \alpha\%2$	
2	`if (A == 3)` ~~`then ...`~~	$\alpha\%2 = 3$	*is UNSAT*
3	`            else ...`	$\alpha\%2 \neq 3$	

Pozorování

- Možné průchody programem lze seskupit a reprezentovat stromovou strukturou – **strom symbolické exekuce**.
- Struktura stromu vzniká rozbalováním grafu toku řízení.

Strom symbolické exekuce

- Vrchol stromu je tvořen lokací programu, symbolickou valuací proměnných a podmínkou cesty, například:

lokace	valuace	podmínka cesty
#12	$A = \alpha + 2, B = \alpha + \beta - 2$	$\alpha = 2 * \beta - 1$

- Hrana mezi vrcholy odpovídá vykonání příkazu na dané lokaci s odpovídající aktualizací symbolické valuace.
- Větvení v programu způsobí větvení ve stromové struktuře a aktualizaci podmínek cest.

Příklad stromu symbolické exekuce

Program

```
1 input A,B
2 if (B<0) then
3   return 0
4 else
5   while (B > 0)
6   { B=B-1
7     A=A+B
8   }
9 return A
```

DODO = DOdělej DOma

Exploze cest

Vlastnosti stromu symbolické exekuce

- Nedochází ke spojování vrcholů (dosažení identické trojice nevede k žádné zpětné/křižné hraně).
- Jedna programová lokace se může vyskytovat ve vícero vnitřních uzlech.
- Strom může obsahovat nekonečné cesty.

Path explosion problem

- Pro netriviální programy je počet větví stromu symbolické exekuce obrovský.
- Počet cest roste exponenciálně vzhledem k počtu průchodů větvícími lokacemi programu.

Použití symbolické exekuce ve verifikaci

Analýza stromu symbolické exekuce

- Prohledávání do šířky, strom je potencionálně nekonečný.

Informace získané o programu

- Určení proveditelných a neproveditelných cest.
- Detekce dosažitelnosti dané lokace.
- Detekce chyb (dělení nulou, přístup mimo pole, porušení invariantu lokace, atd.).

Syntéza testovacích dat

- Je-li podmínka cesty pro nějaký symbolický běh splnitelná, modelem této formule jsou konkrétní vstupní hodnoty programu, které si vynutí výpočet programu podle dané cesty.
- Syntéza testů zvyšující pokrytí kódu.

Automatické generování testů

Princip

1. Vygenerujeme náhodné vstupní hodnoty (náhodná cesta).
2. S danými hodnotami provedeme průchod stromem symbolické exekuce a zaznamenáme podmínku cesty.
3. Z podmínky cesty vytvoříme novou podmínku cesty tím, že negujeme formuli vybraného větvení.
4. Najdeme vstupní data vyhovující nové podmínce cesty.
5. Opakujeme od bodu 2 (pokud nové testy zvyšují pokrytí).

Realizace

- Heuristiky pro výběr větvení, jehož podmínka bude negována.
- Augmentace kódu pro zaznamenání podmínky cesty.

Limity symbolické exekuce

Nerozhodnutelnost

- Použití kompletní aritmetiky na neomezených doménách implikuje obecnou nerozhodnutelnost problému splnitelnosti.
- Strom symbolické exekuce může být nekonečný (rozbalování cyklů s dynamickým počtem opakování).

Výpočetní náročnost

- Exploze cest.
- Algoritmy pro splnitelnost formulí na omezených doménách.

Omezení

- Jak realizovat operace nad nečíselnými proměnnými?
- Jak reprezentovat dynamické datové struktury?
- Jak symbolicky vyhodnotit volání externí funkce?

Automatizace testu splnitelnosti

Problém SAT

Problém splnitelnosti – SAT

- Problém rozhodnout, zda existuje valuace boolovských proměnných formule výrokové logiky taková, že formule je v této valuaci pravdivá.

Vlastnosti

- Jeden z nejznámějších NP-úplných problémů.
- Pro daný problém není znám polynomiální algoritmus.
- Existující řešiče SAT jsou velmi efektivní a díky mnohým heuristikám zvládají řešit překvapivě velké instance problému.

Nástroj Z3

ZZZ aka **Z3**

- Nástroj vyvíjený v Microsoft Research.
- Řešič instancí problémů SAT a SMT.
- WWW interface — `http://www.rise4fun.com/Z3`
- Binární API pro použití v jiných aplikacích.

Rozhodněte pomocí Z3

- Je splnitelná formule $(a \vee \neg b) \wedge (\neg a \vee b)$?

Použití Z3 – SAT

Reformulace formule pro Z3

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

```
(declare-const a Bool)
(declare-const b Bool)
(assert (and (or a (not b)) (or (not a) b)))
(check-sat)
(get-model)
```

Odpověď od Z3

```
sat
(model
  (define-fun b () Bool
    false)
  (define-fun a () Bool
    false)
)
```

Problém splnitelnosti v dané teorii – SMT

Satisfiability Modulo Theory – SMT

- Problém rozhodnout splnitelnost formule prvořádové logiky s rovností, predikáty a funkčními symboly kódující jednu či více zvolených teorií.
- Typicky používané teorie
  - Aritmetika celých a desetinných čísel.
  - Teorie datových struktur (seznamy, pole, bitové vektory, ...).

Jiný pohled (převzato z Wikipedie)

- Na SMT lze také nahlížet jako na jistou formu hledání řešení vyhovující sadě daných omezení, tudíž lze to také chápat jako jistý formalizovaný přístup k oblasti programování s omezeními (constraint programming).

Příklady Z3 – SMT

Řešte pomocí Z3 `http://rise4fun.com/Z3/tutorial/guide`

- Existují celá nenulová čísla `x` a `y` taková, že `y=x*(x-y)`?

```
(declare-const y Int)
(declare-const x Int)
(assert (= y (* x (- x y))))
(assert (not (= y 0)))
(check-sat)
(get-model)
```

- Existují celá nenulová čísla `x` a `y` taková, že `y=x*(x-(y*y))`?

```
(declare-const y Int)
(declare-const x Int)
(assert (= y (* x (- x (* y y)))))
(assert (not (= x 0)))
(check-sat)
```

Splnitelnost a platnost

Pozorování

- Formule je platná právě když její negace není splnitelná.

Důsledek

- Řešiče SAT a SMT lze využít jako nástroje pro dokazování platnosti formulovaných tvrzení.

Syntéza modelu

- Řešiče SAT nejen rozhodují splnitelnost formulí, ale v případě splnitelnosti vrací požadovanou valuaci proměnných, pro níž je formule pravdivá.
- Na rozdíl od dokazovacích nástrojů tak poskytují "protipříklad" v případě neplatnosti dokazovaného tvrzení.

Konkolické Testování

Motivace

Problém

- Principielní nerozhodnutelnost proveditelnosti cesty.
- V praxi typicky nerozhodnutelnost znamená nesplnitelnost.
- Vynecháním těchto cest můžeme minout chybu.
- Provedením těchto cest můžeme nalézt nereálnou chybu.

Částečné řešení

- Současné použití konkrétních a symbolických hodnot vstupních proměnných a využití konkrétních hodnot pro rozhodnutí jinak nerozhodnutelné instance splnitelnosti.
- Heuristika.
- Zajímavý případ (korektní): UNKNOWN $\Longrightarrow$ SAT
- **Con**crete and Symb**olic Testing** = **Concolic Testing**

Hypotetická ukázka konkolického testování

Program

```
1 input A,B
2 if (A==(B*B)%30) then
3   ERROR
4 else
5   return A
```

Konkolické testování

1. A=22, B=7 (náhodné hodnoty)
2. (22==(7*7)%30) je *False*, podmínka cesty: $\alpha \neq (\beta * \beta)\%30$
3. Test dopadl OK
4. Syntéza dat z negace PC: $\alpha = (\beta * \beta)\%30$ – **UNKNOWN**
5. Využití konkrétních hodnot: $\alpha = (7 * 7)\%30$ – **SAT**, $\alpha = 19$
6. A=19, B=7
7. Test odhalil chybovou lokaci na řádku 3.

Nástroj SAGE

Příběh nástroje SAGE

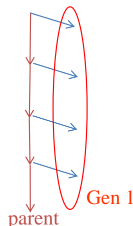
Systematic Testing for Security: Whitebox Fuzzing

Patrice Godefroid
Michael Y. Levin and David Molnar

http://research.microsoft.com/projects/atg/
Microsoft Research

Whitebox Fuzzing (SAGE tool)

- Start with a well-formed input (not random)
- Combine with a generational search (not DFS)
  - Negate 1-by-1 each constraint in a path constraint
  - Generate many children for each parent run
  - Challenge all the layers of the application sooner
  - Leverage expensive symbolic execution



- Search spaces are huge, the search is partial… yet effective at finding bugs !

Example: Dynamic Test Generation

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == ‘b’) cnt++;
   if (input[1] == ‘a’) cnt++;
   if (input[2] == ‘d’) cnt++;
   if (input[3] == ‘!’) cnt++;
   if (cnt > 3) crash();
}
```

input = “good”

Dynamic Test Generation

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt > 3) crash();
}
```

input = "good"

Path constraint:

I_0 != 'b'

I_1 != 'a'

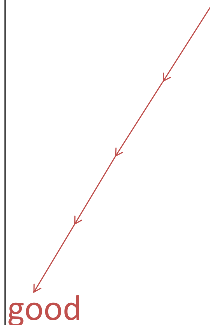
I_2 != 'd'

I_3 != '!'

Negate a condition in path constraint

Solve new constraint → new input

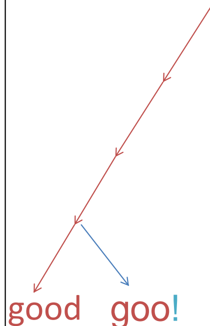
Depth-First Search



input = "good"

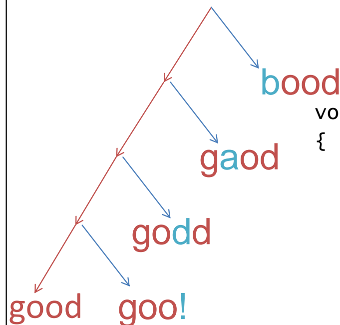
```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;   I0 != 'b'
    if (input[1] == 'a') cnt++;   I1 != 'a'
    if (input[2] == 'd') cnt++;   I2 != 'd'
    if (input[3] == '!') cnt++;   I3 != '!'
    if (cnt > 3) crash();
}
```

Depth-First Search




```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;   I0 != 'b'
    if (input[1] == 'a') cnt++;   I1 != 'a'
    if (input[2] == 'd') cnt++;   I2 != 'd'
    if (input[3] == '!') cnt++;   I3 == '!'
    if (cnt > 3) crash();
}
```

Generational Search



Four “Generation 1” test cases !

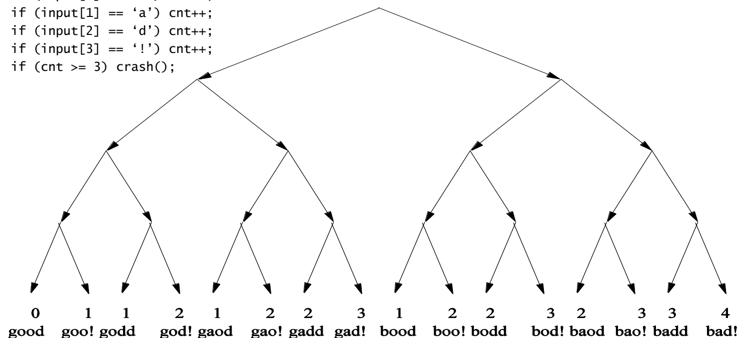
```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

I_0 == 'b'
I_1 == 'a'
I_2 == 'd'
I_3 == '!'

Příběh nástroje SAGE

The Search Space

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == ‘b’) cnt++;
   if (input[1] == ‘a’) cnt++;
   if (input[2] == ‘d’) cnt++;
   if (input[3] == ‘!’) cnt++;
   if (cnt >= 3) crash();
}
```



Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 0 – seed file

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 00 00 00 00 00 00 00 00 00 00 00 00 ; RIFF............
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 1

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 00 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF....*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 2

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 3

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 00 00 00 00 ; ....strh........
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 4

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 5

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 00 00 00 00 ; ....strf........
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 6

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 7

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 C9 9D E4 4E ; ............É•äN
00000060h: 00 00 00 00                                     ; ....
```

Generation 8

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 9

Zero to Crash in 10 Generations

- Starting with 100 zero bytes ...
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 B2 75 76 3A 28 00 00 00 ; ....strf²uv:(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 10 – crash bucket 1212954973!

Příběh nástroje SAGE

Initial Experiences with SAGE

- Since 1st internal release in April'07: tens of new security bugs found
- Apps: image processors, media players, file decoders,... Confidential !
- Bugs: Write A/Vs, Read A/Vs, Crashes,... Confidential !
- Many bugs found triaged as "security critical, severity 1, priority 1"