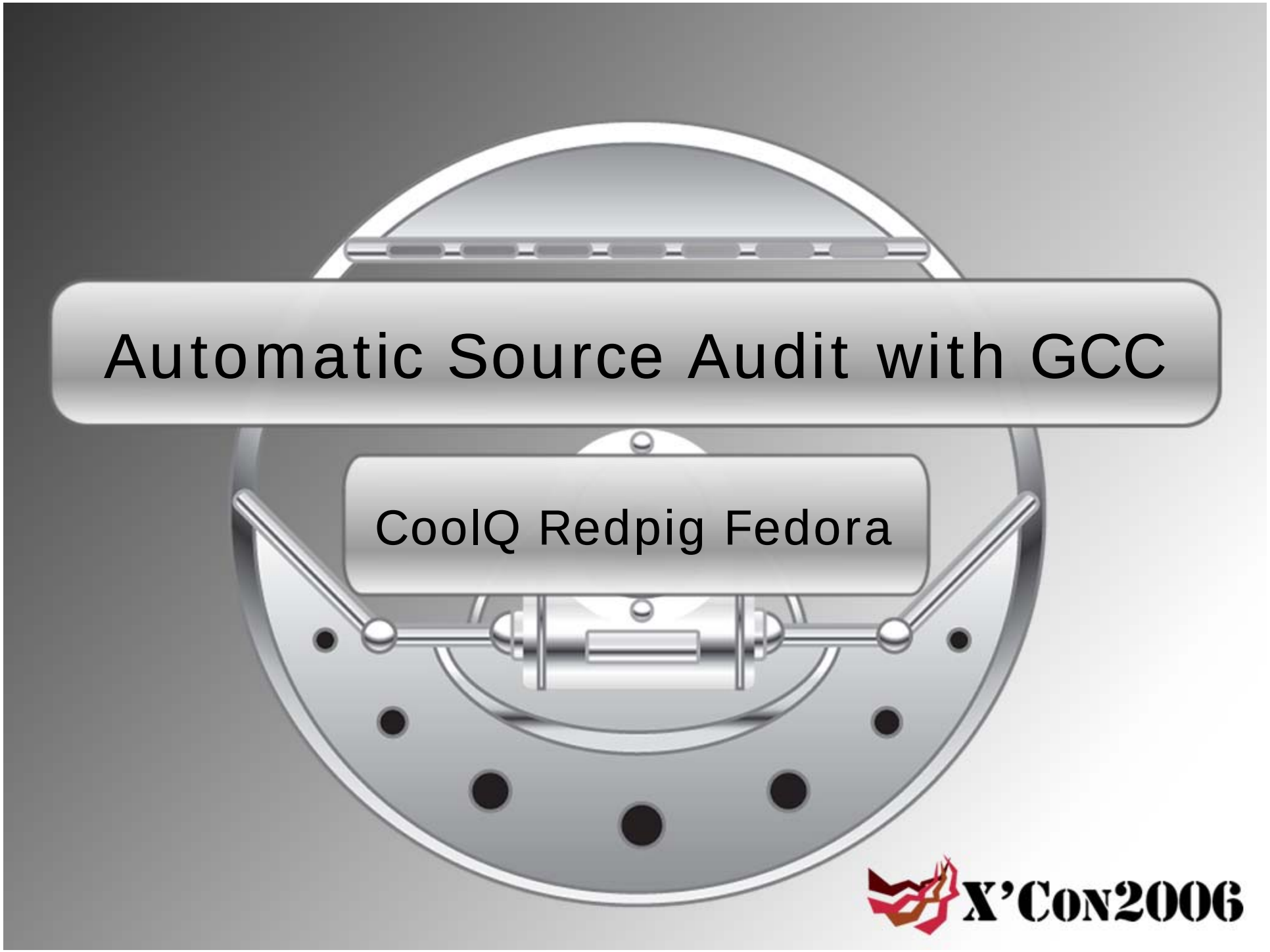
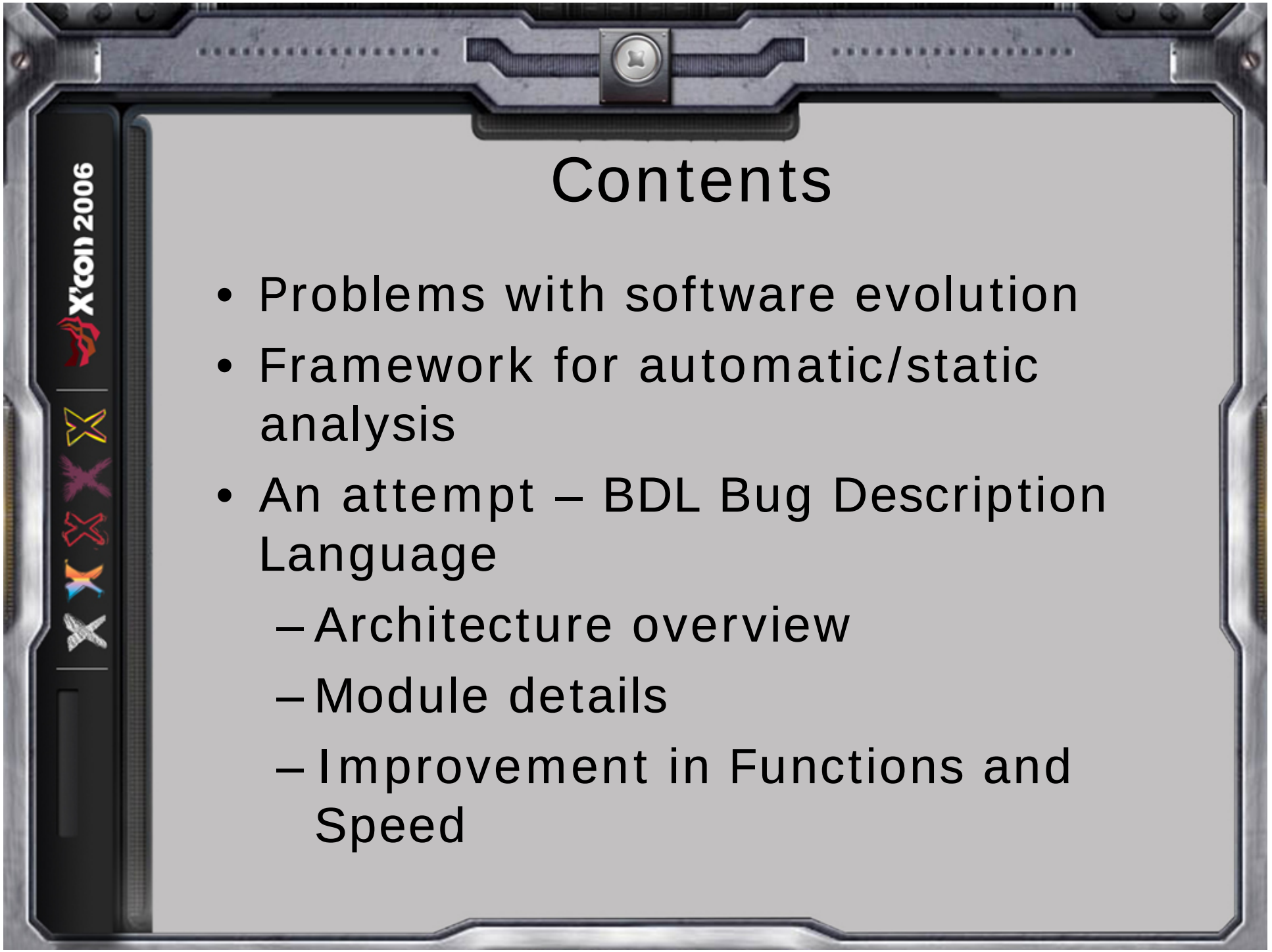
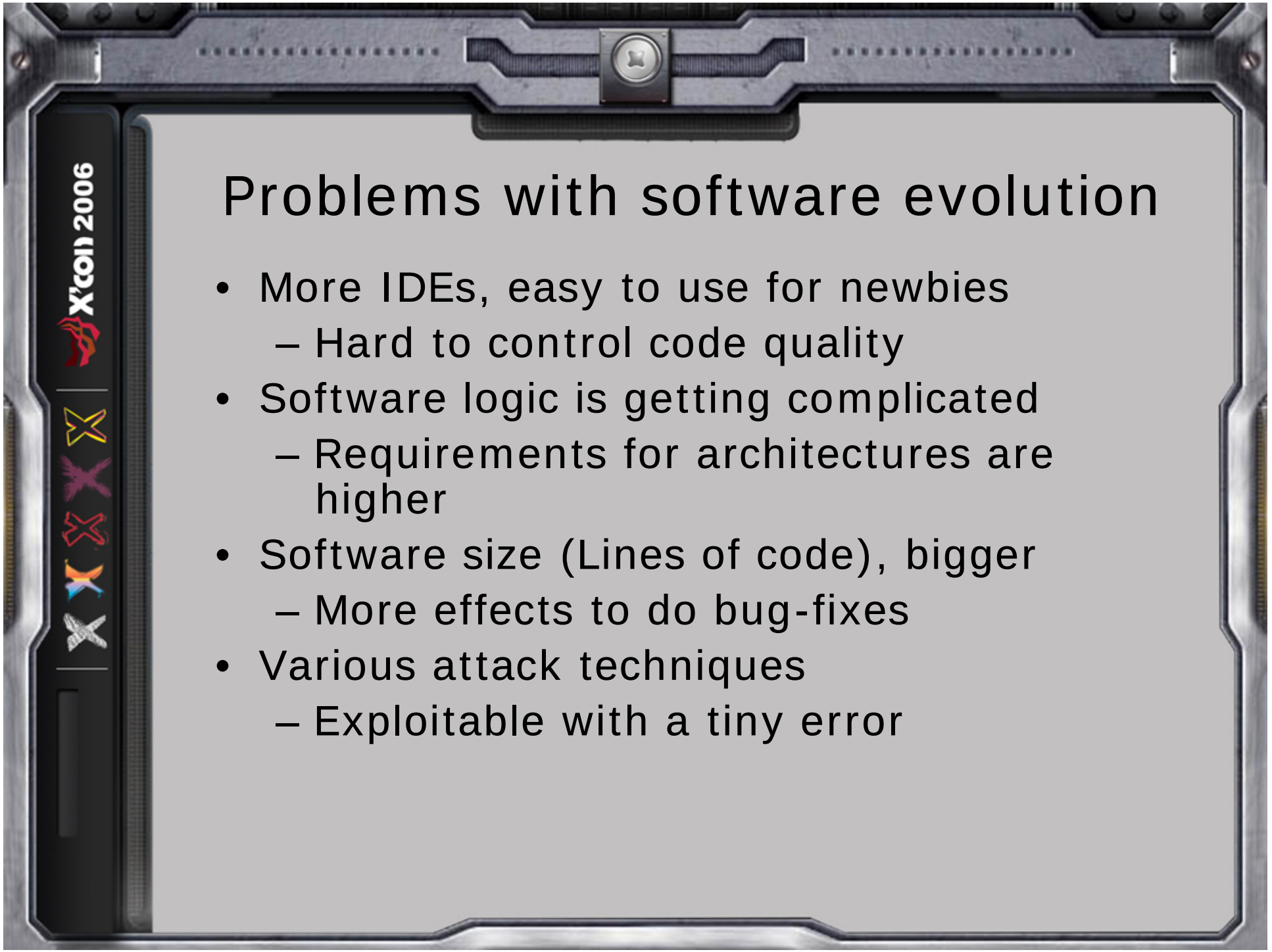

Automatic Source Audit with GCC
CoolQ Redpig Fedora
X'Con2006



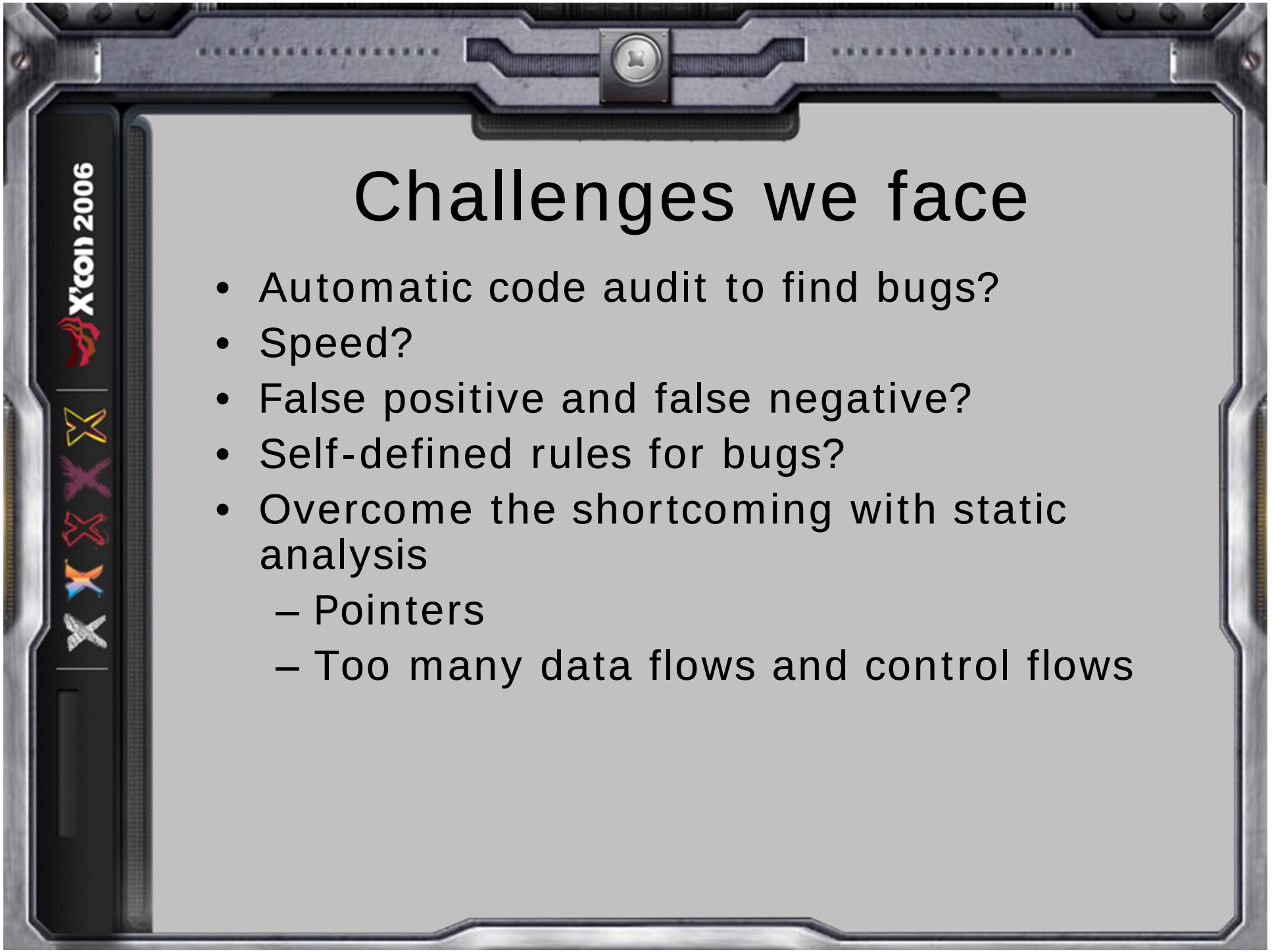
Contents

- Problems with software evolution
- Framework for automatic/static analysis
- An attempt – BDL Bug Description Language
  - Architecture overview
  - Module details
  - Improvement in Functions and Speed



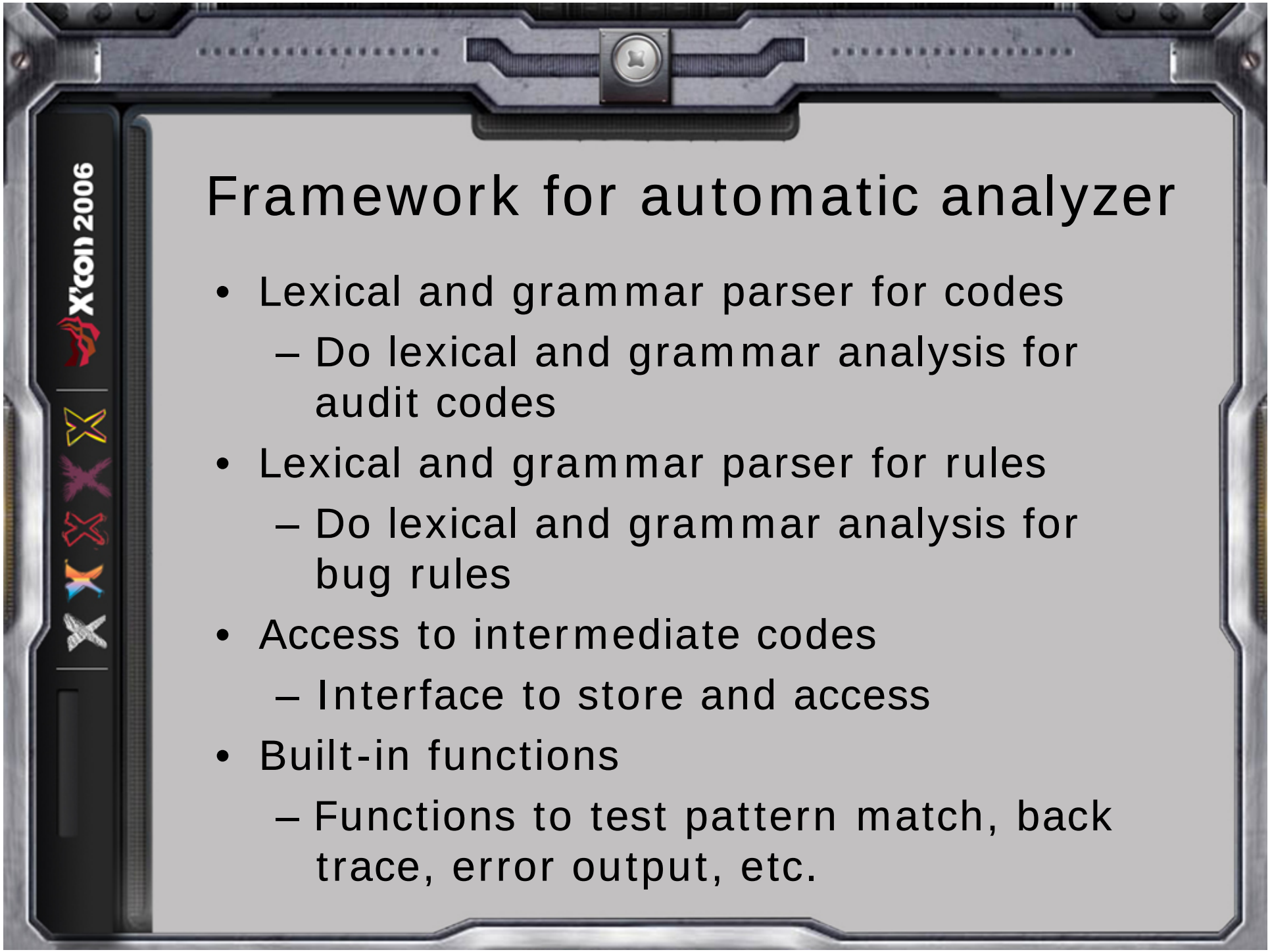
Problems with software evolution

- More IDEs, easy to use for newbies
  - Hard to control code quality
- Software logic is getting complicated
  - Requirements for architectures are higher
- Software size (Lines of code), bigger
  - More effects to do bug-fixes
- Various attack techniques
  - Exploitable with a tiny error



Challenges we face

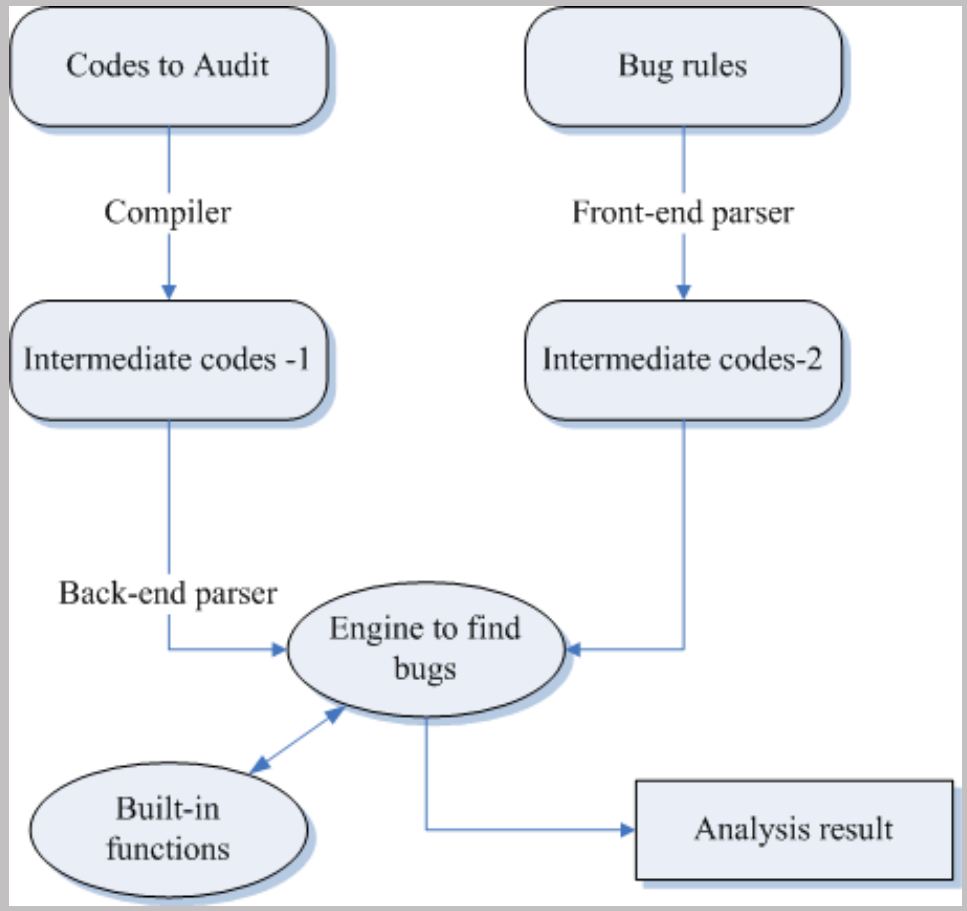
- Automatic code audit to find bugs?
- Speed?
- False positive and false negative?
- Self-defined rules for bugs?
- Overcome the shortcoming with static analysis
  - Pointers
  - Too many data flows and control flows



Framework for automatic analyzer

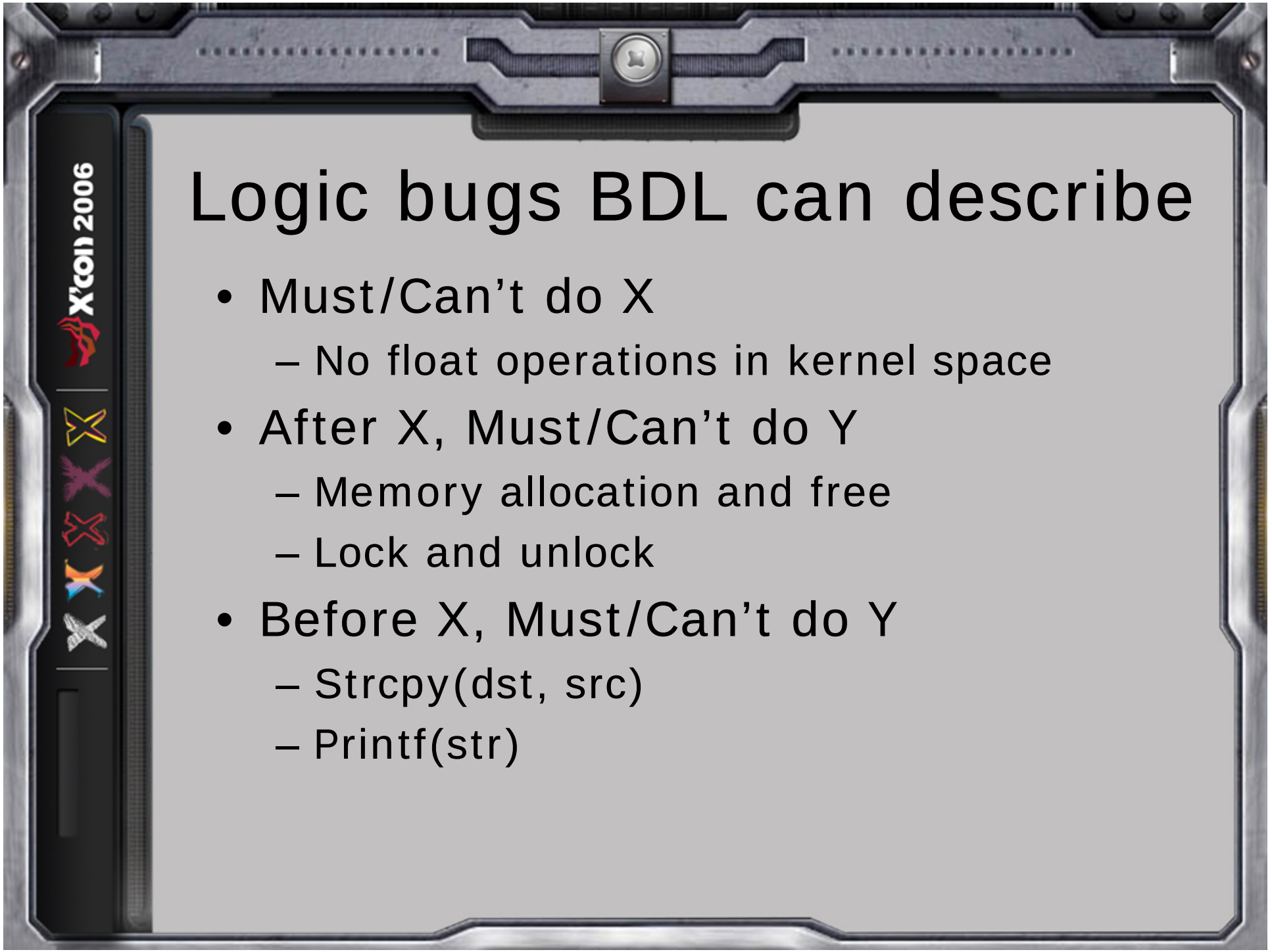
- Lexical and grammar parser for codes
  - Do lexical and grammar analysis for audit codes
- Lexical and grammar parser for rules
  - Do lexical and grammar analysis for bug rules
- Access to intermediate codes
  - Interface to store and access
- Built-in functions
  - Functions to test pattern match, back trace, error output, etc.

Framework for automatic analyzer Illustration



An attempt - BDL

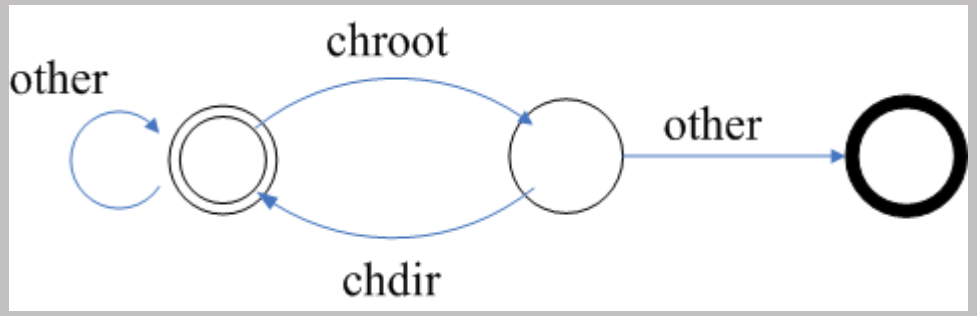
- BDL – Bug description Language
- Based on GCC + Flex + Bison
  - Bug rules are defined by a simple language parsed by Flex and Bison
  - Codes to be audited are parsed by GCC
- The state transformation of single variable is tracked by automata engine
- Support control flows and simple data flows

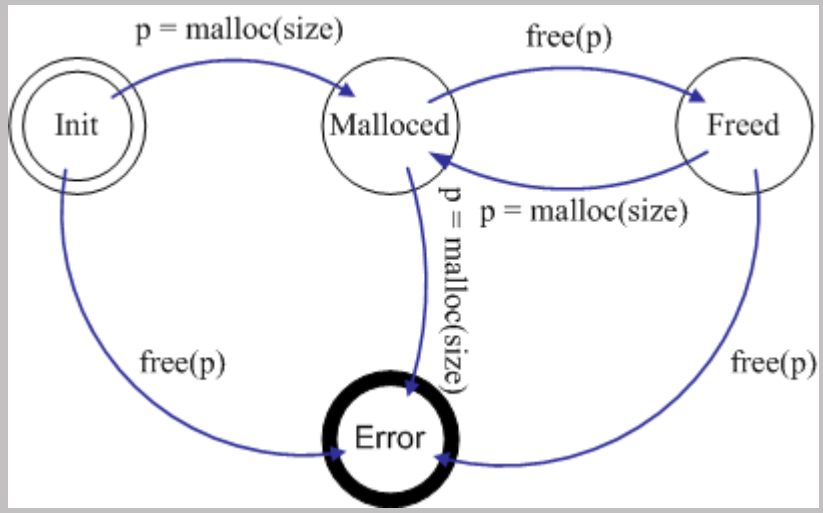


Logic bugs BDL can describe

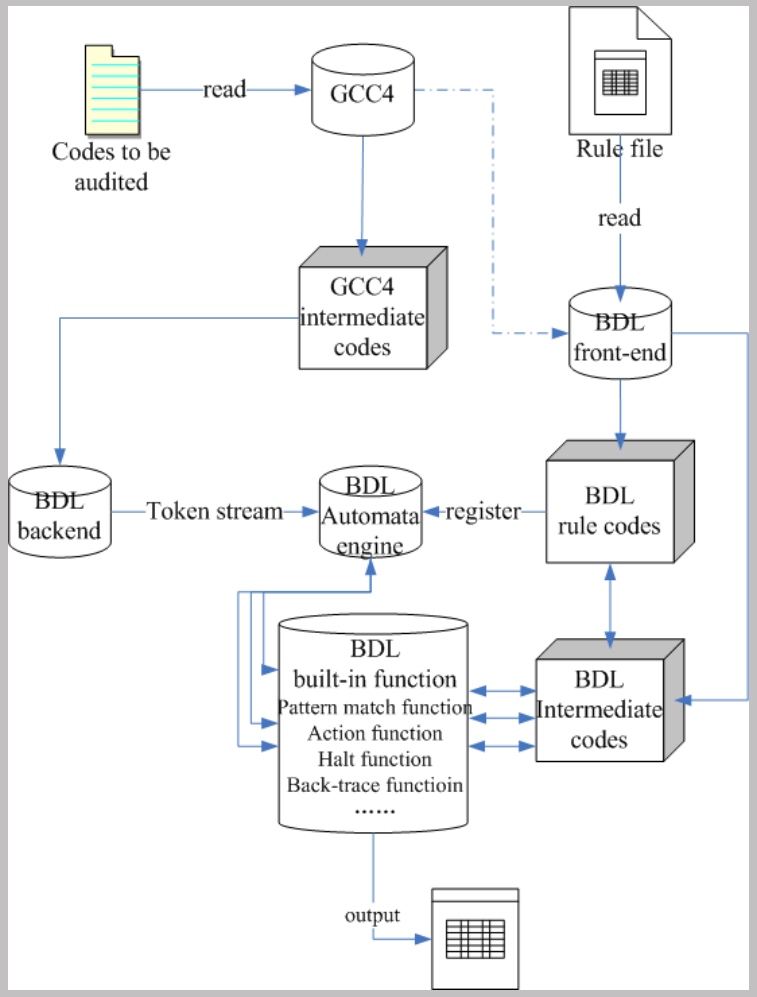
- Must/Can't do X
  - No float operations in kernel space
- After X, Must/Can't do Y
  - Memory allocation and free
  - Lock and unlock
- Before X, Must/Can't do Y
  - Strcpy(dst, src)
  - Printf(str)

State transformation exapmles

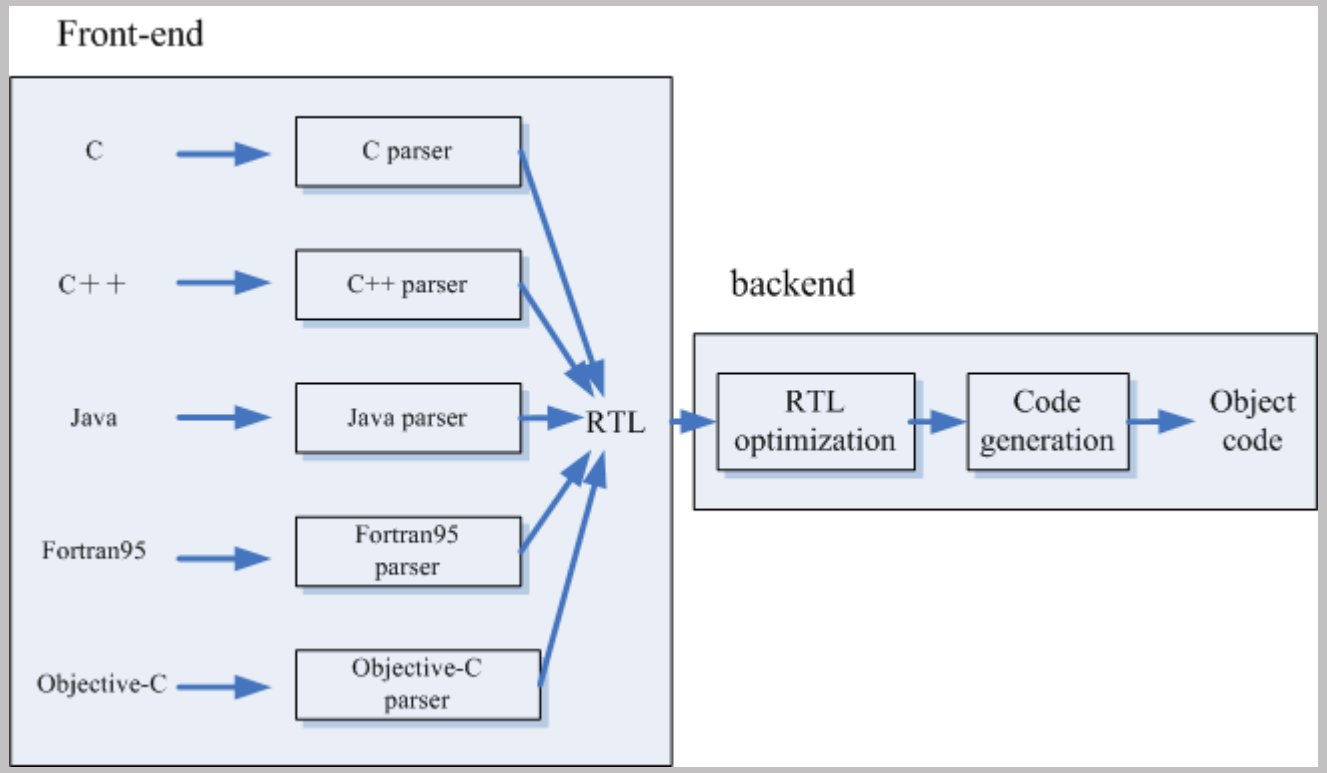




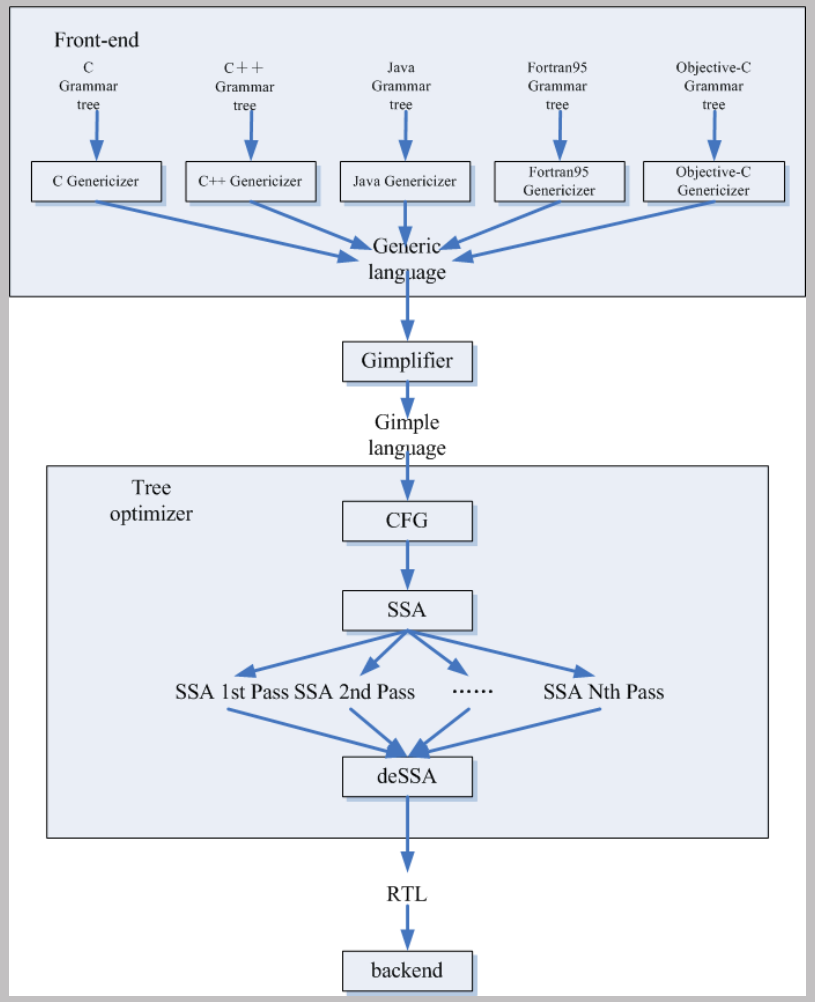
Architecture overview



GCC3 Overview

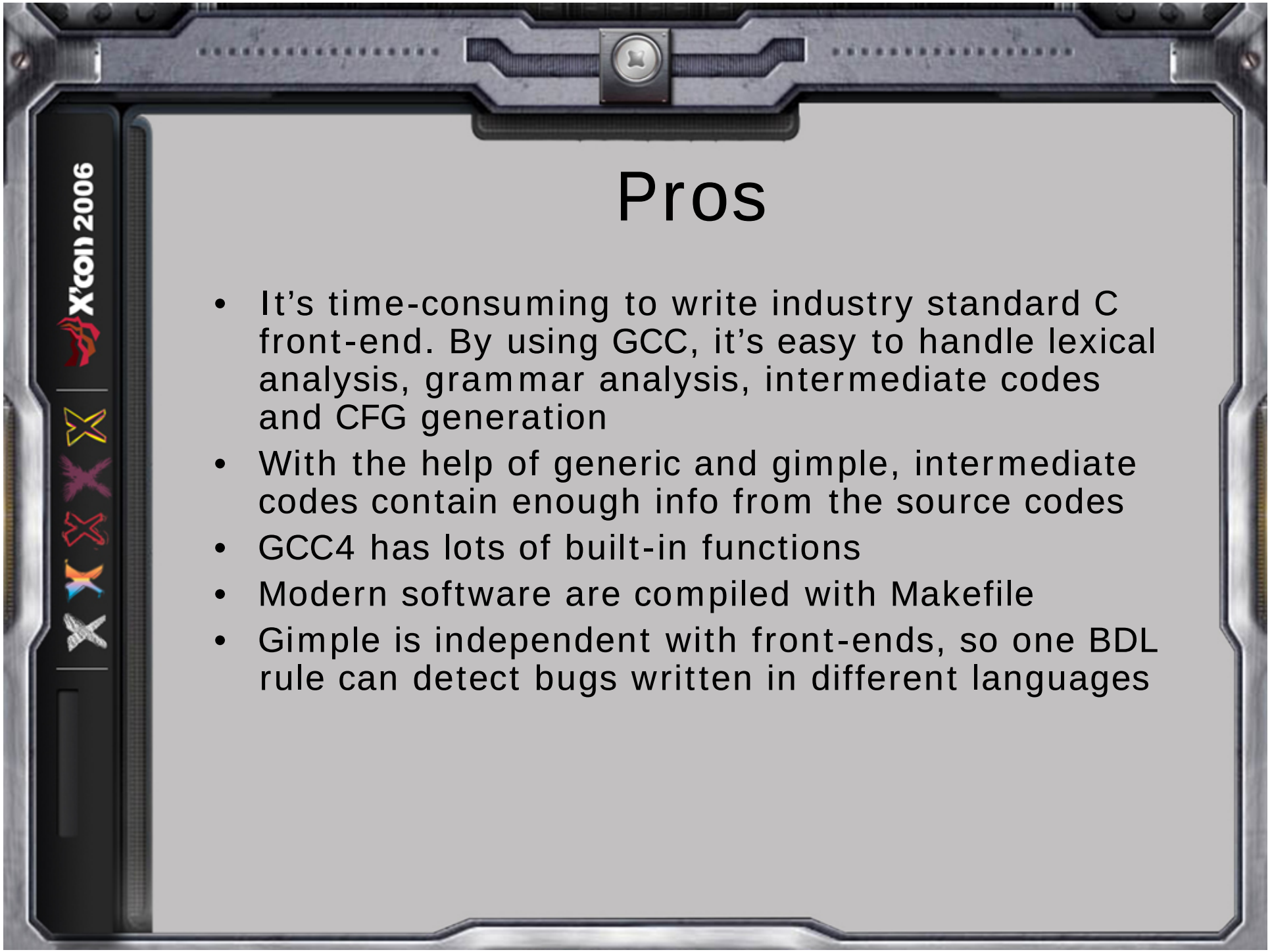


GCC4 Overview



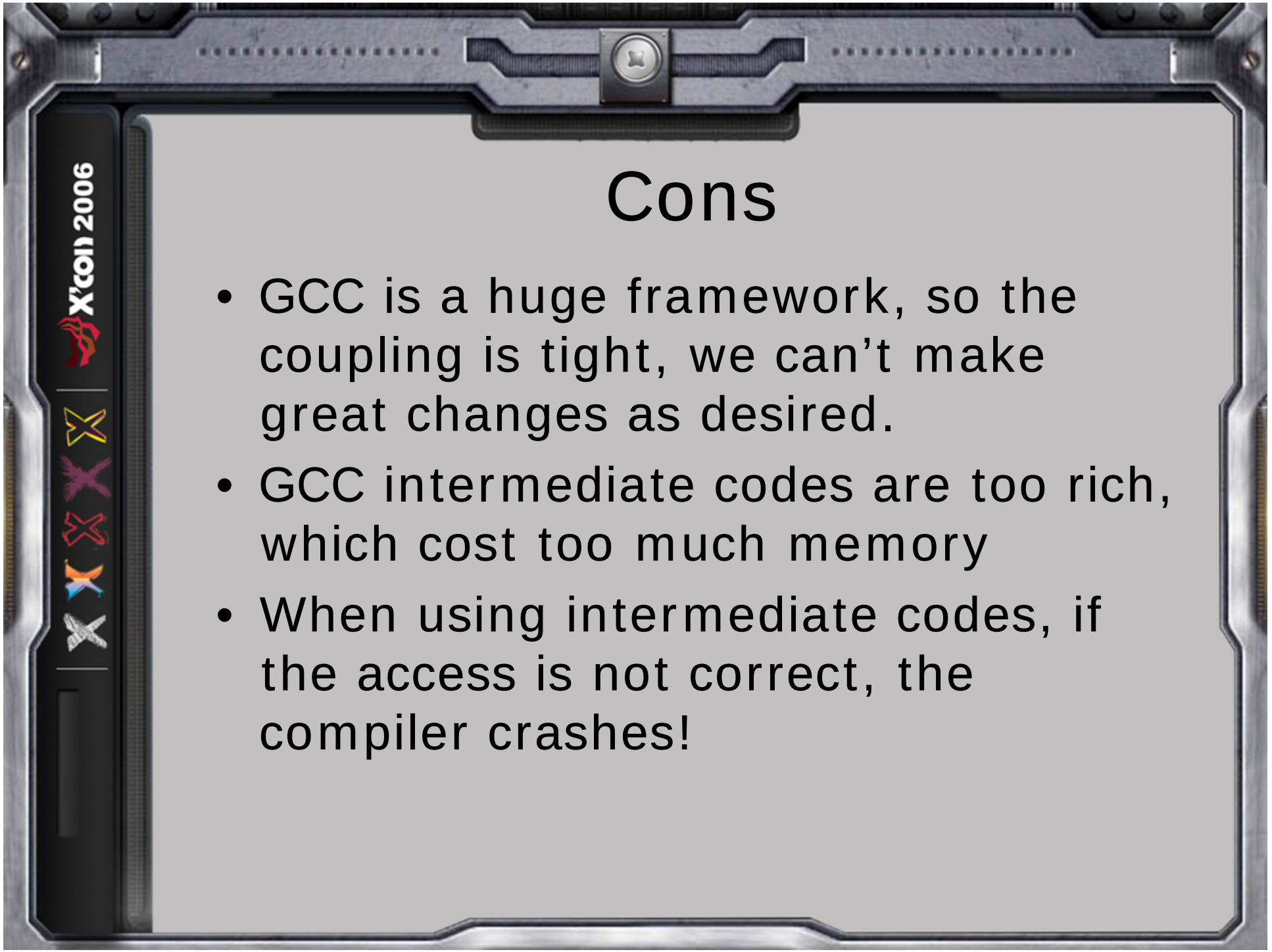
New features in GCC4

- The intermediate codes are independent with language front-ends, so, we can find bugs in many languages
- With the help of generic and gimple, front-end codes convert into simple ones.
- CFG is in Gimple layer, very easy to use



Pros

- It’s time-consuming to write industry standard C front-end. By using GCC, it’s easy to handle lexical analysis, grammar analysis, intermediate codes and CFG generation
- With the help of generic and gimple, intermediate codes contain enough info from the source codes
- GCC4 has lots of built-in functions
- Modern software are compiled with Makefile
- Gimple is independent with front-ends, so one BDL rule can detect bugs written in different languages

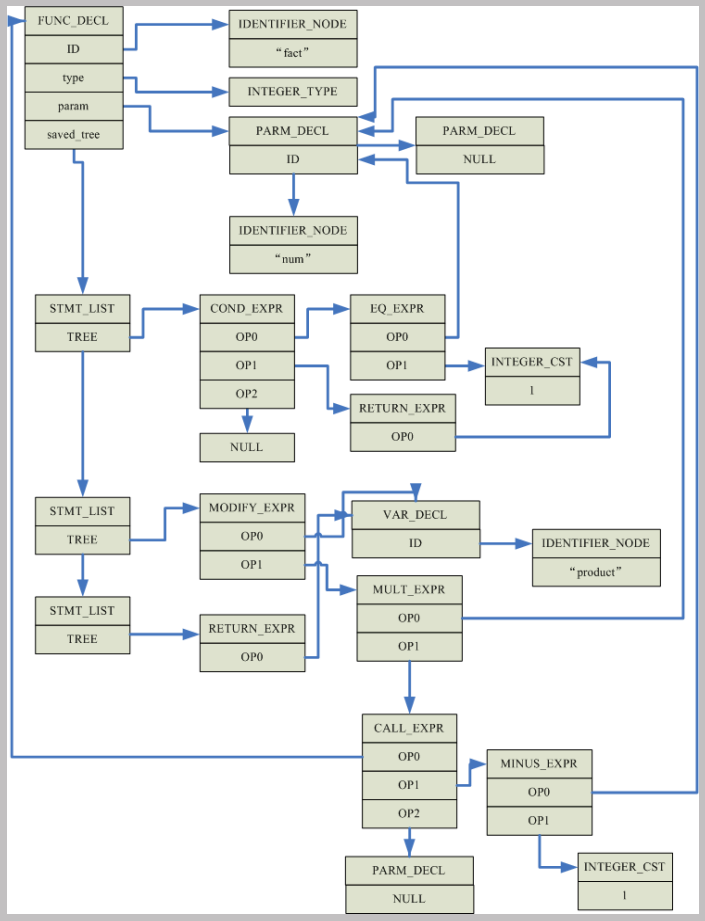


Cons

- GCC is a huge framework, so the coupling is tight, we can’t make great changes as desired.
- GCC intermediate codes are too rich, which cost too much memory
- When using intermediate codes, if the access is not correct, the compiler crashes!

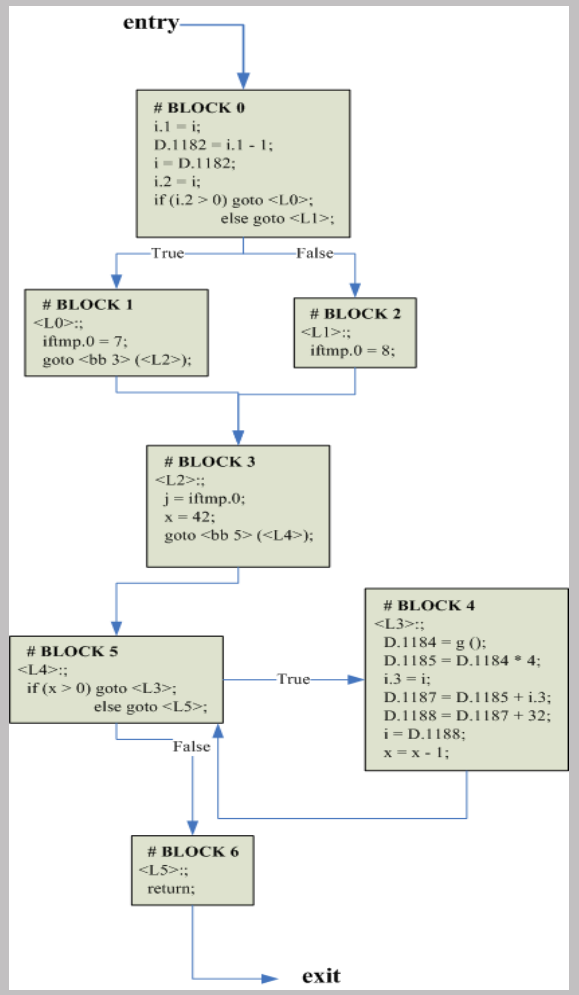
Grammar tree and data structures in GCC4

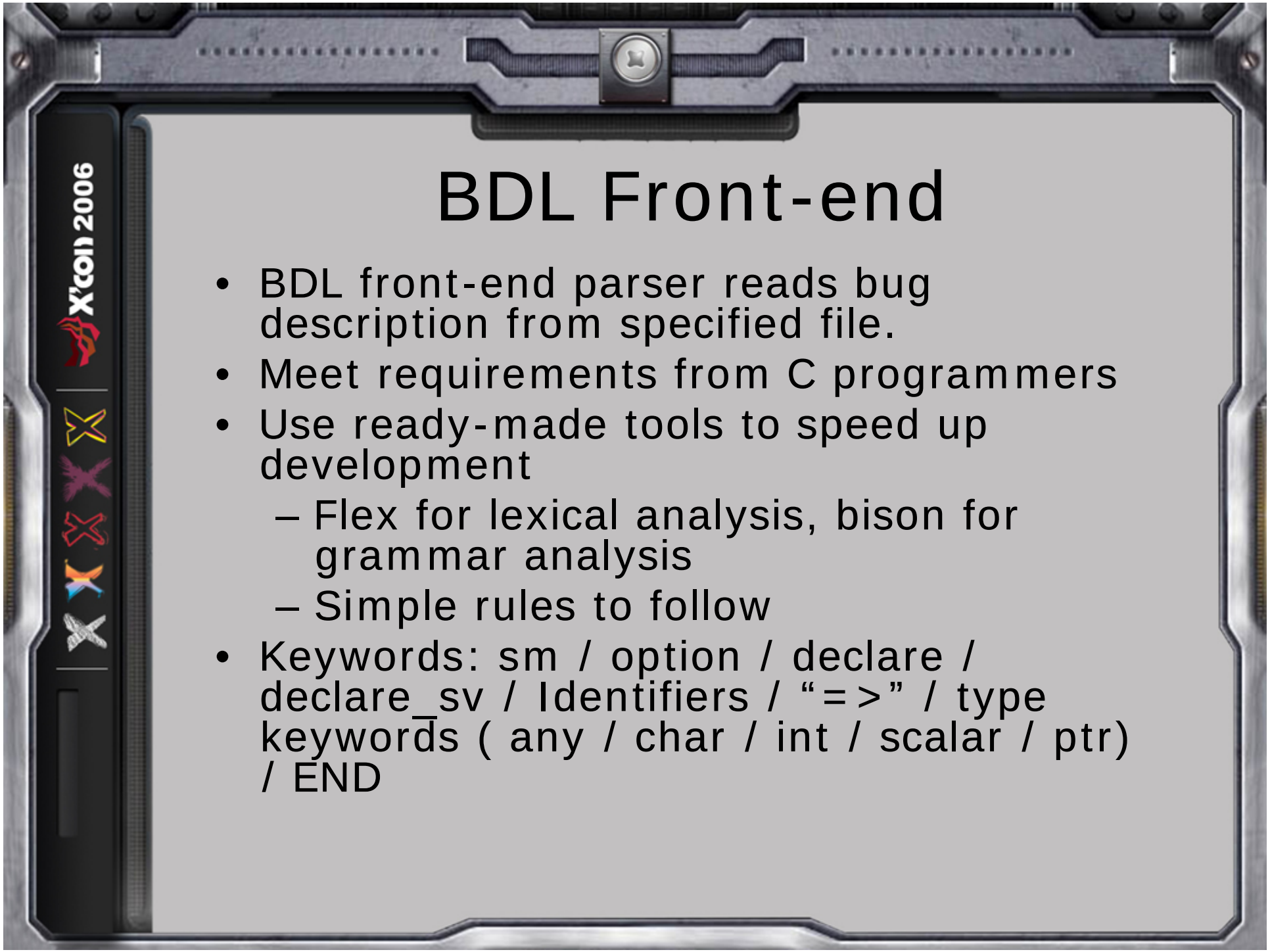
```
unsigned int fact
(unsigned int num)
{
        unsigned int product;
        if(num = = 1)
                return 1;
        product = num *
                        fact(num – 1);
        return product;
}
```



Gimple and CFG in GCC4

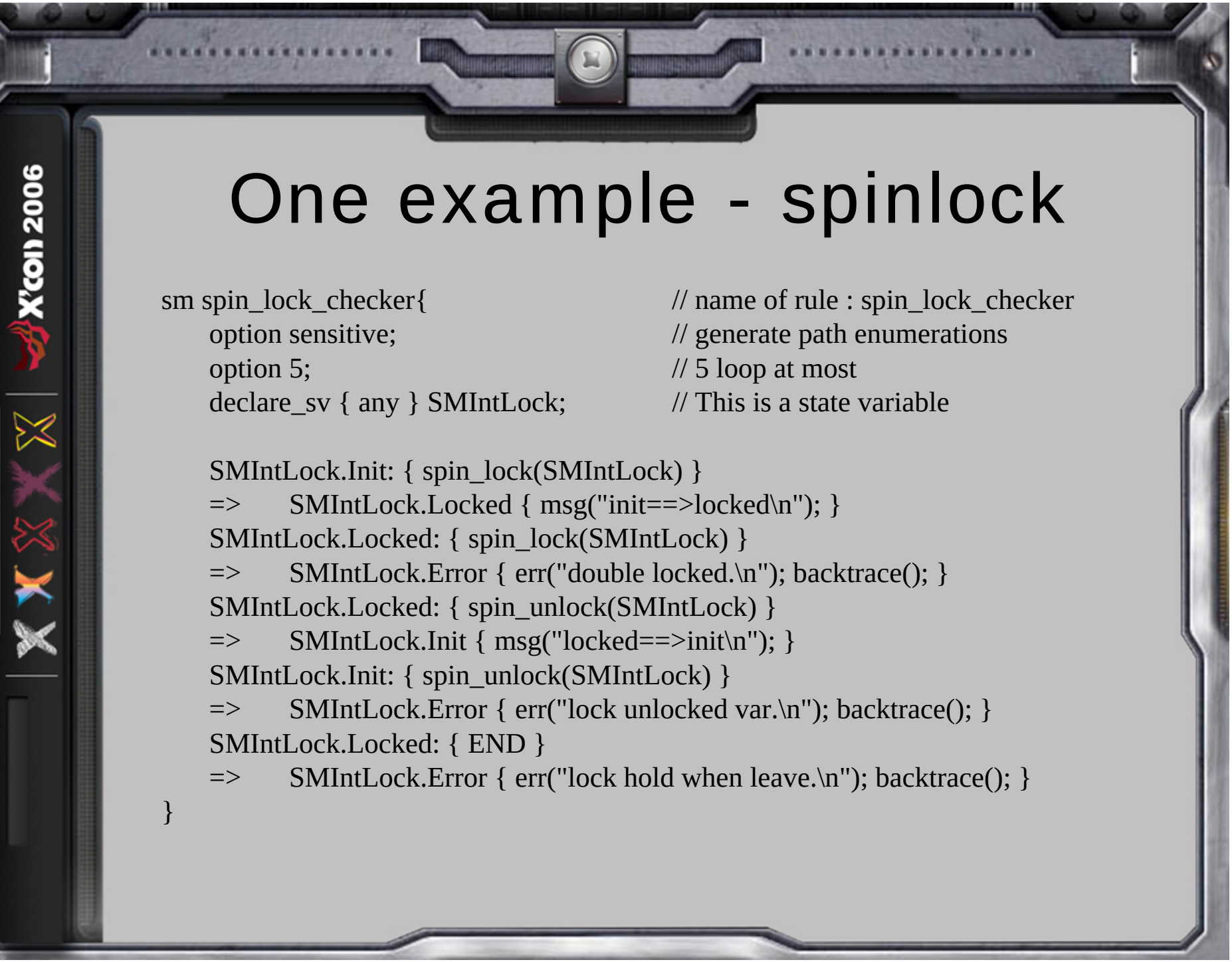
```
int i;

int g(void);

void foo(void)

{

    int j = (--i, i > 0? 7: 8);

    for(int x = 42; x > 0; --x)

        i += g() * 4 + 32;

}
```





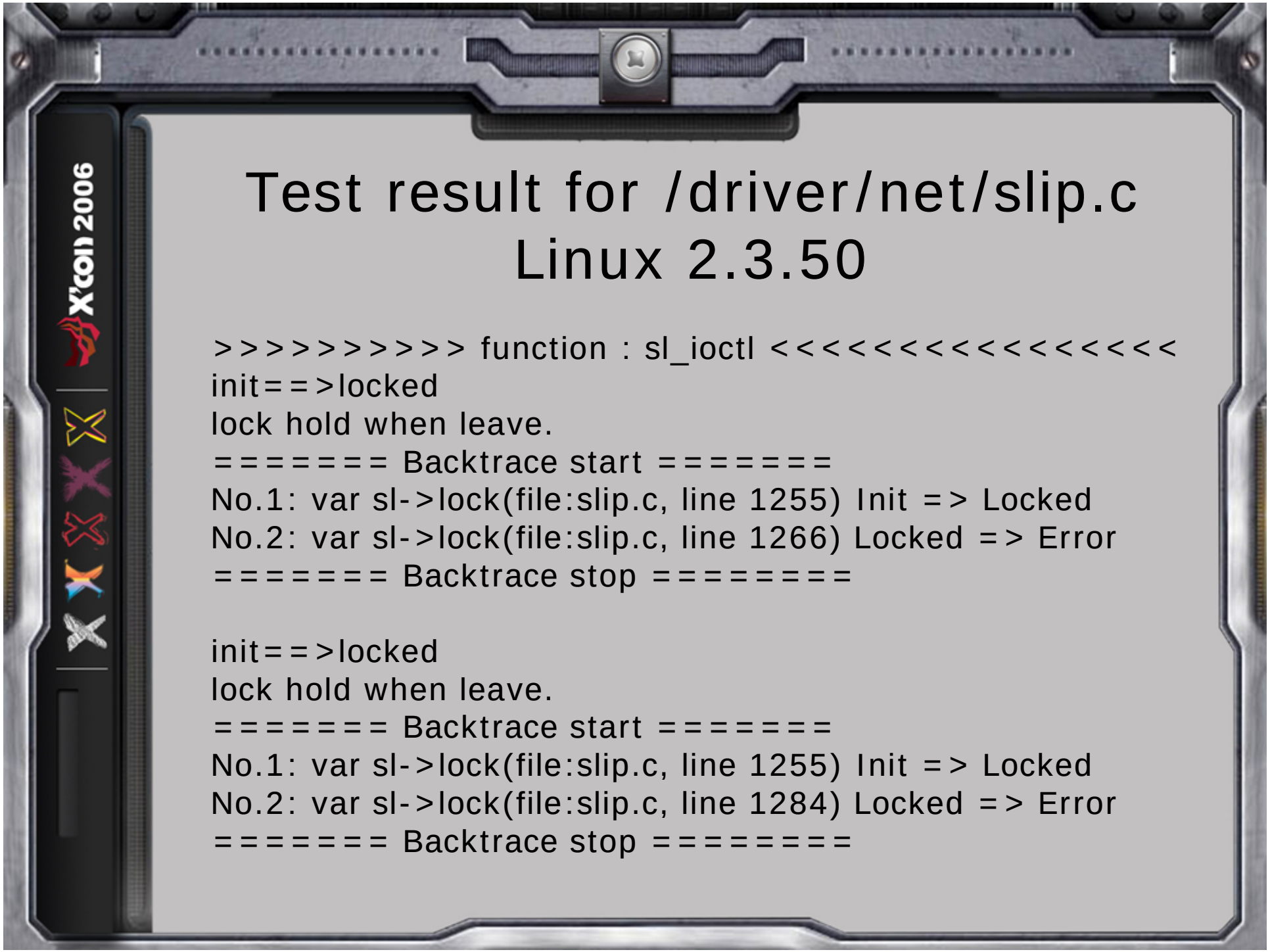
BDL Front-end

- BDL front-end parser reads bug description from specified file.
- Meet requirements from C programmers
- Use ready-made tools to speed up development
  - Flex for lexical analysis, bison for grammar analysis
  - Simple rules to follow
- Keywords: sm / option / declare / declare_sv / Identifiers / “=>” / type keywords (any / char / int / scalar / ptr) / END



One example - spinlock

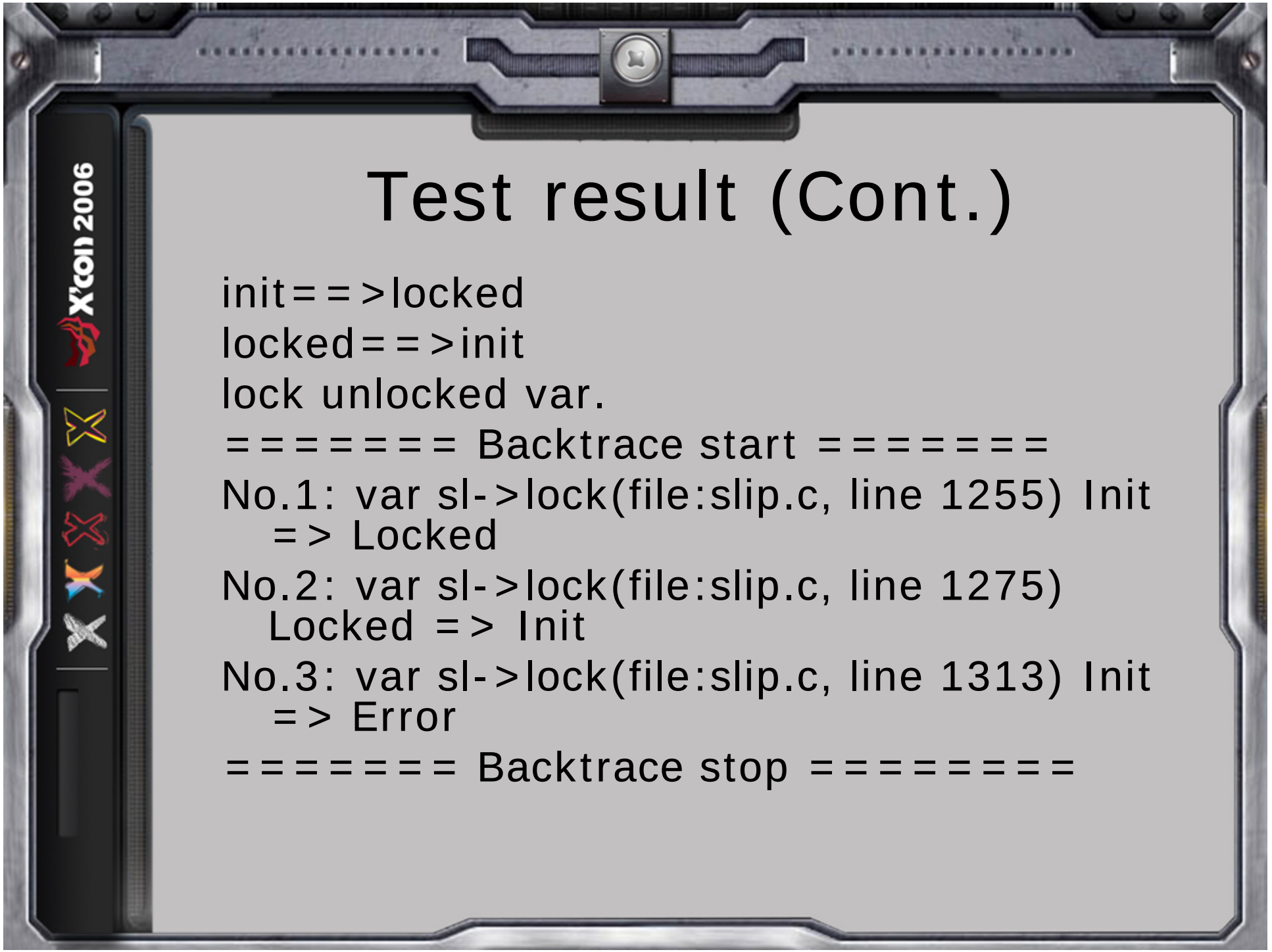
```
sm spin_lock_checker{                          // name of rule : spin_lock_checker
      option sensitive;                        // generate path enumerations
      option 5;                                // 5 loop at most
      declare_sv { any } SMIntLock;            // This is a state variable

      SMIntLock.Init: { spin_lock(SMIntLock) }
      =>      SMIntLock.Locked { msg("init==>locked\n"); }
      SMIntLock.Locked: { spin_lock(SMIntLock) }
      =>      SMIntLock.Error { err("double locked.\n"); backtrace(); }
      SMIntLock.Locked: { spin_unlock(SMIntLock) }
      =>      SMIntLock.Init { msg("locked==>init\n"); }
      SMIntLock.Init: { spin_unlock(SMIntLock) }
      =>      SMIntLock.Error { err("lock unlocked var.\n"); backtrace(); }
      SMIntLock.Locked: { END }
      =>      SMIntLock.Error { err("lock hold when leave.\n"); backtrace(); }
}
```



Test result for /driver/net/slip.c Linux 2.3.50

>>>>>>>>>> function : sl_ioctl <<<<<<<<<<<<<<<<<<<<
init==>locked
lock hold when leave.
======== Backtrace start ========
No.1: var sl->lock(file:slip.c, line 1255) Init => Locked
No.2: var sl->lock(file:slip.c, line 1266) Locked => Error
======== Backtrace stop ========

init==>locked
lock hold when leave.
======== Backtrace start ========
No.1: var sl->lock(file:slip.c, line 1255) Init => Locked
No.2: var sl->lock(file:slip.c, line 1284) Locked => Error
======== Backtrace stop ========



Test result (Cont.)

init==>locked

locked==>init

lock unlocked var.

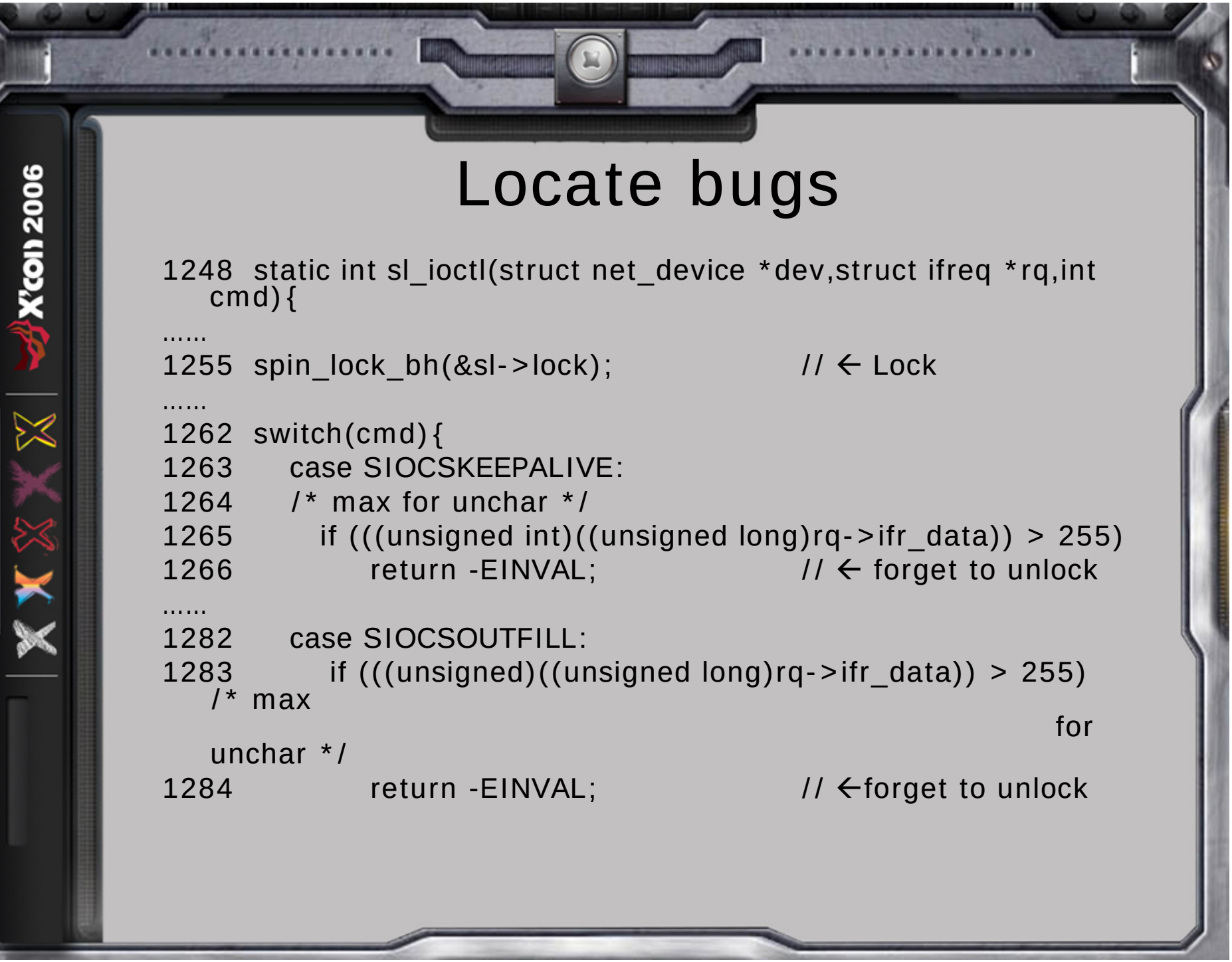
======== Backtrace start ========

No.1: var sl->lock(file:slip.c, line 1255) Init => Locked

No.2: var sl->lock(file:slip.c, line 1275) Locked => Init

No.3: var sl->lock(file:slip.c, line 1313) Init => Error

======== Backtrace stop ========



Locate bugs

```
1248  static int sl_ioctl(struct net_device *dev,struct ifreq *rq,int
    cmd){
......
1255  spin_lock_bh(&sl->lock);                  // ← Lock
......
1262  switch(cmd){
1263     case SIOCSKEEPALIVE:
1264     /* max for unchar */
1265       if (((unsigned int)((unsigned long)rq->ifr_data)) > 255)
1266           return -EINVAL;                   // ← forget to unlock
......
1282     case SIOCSOUTFILL:
1283        if (((unsigned)((unsigned long)rq->ifr_data)) > 255)
    /* max
                                                          for
    unchar */
1284           return -EINVAL;                   // ←forget to unlock
```

Locate bugs (Cont.)

```
1248  static int sl_ioctl(struct net_device *dev,struct ifreq *rq,int
      cmd){
          ……
1255  spin_lock_bh(&sl->lock);                  //  ← lock
          ……
1262  switch(cmd){
1263      case SIOCSKEEPALIVE:
          ……
1275          spin_unlock_bh(&sl->lock);        // ← unlock
1276          break;
          ……
1312  };
1313  spin_unlock_bh(&sl->lock);                // ← unlock again
1314  return 0;
1315 }
```

Another example – Mutex lock

- The kernel source code of FreeBSD6.0
- report:

Exit with lock hold.
======== Backtrace start ========
No.1: var Giant(file:../../../cam/scsi/scsi_cd.c, line 584) init => locked
No.2: var Giant(file:../../../cam/scsi/scsi_cd.c, line 594) locked => error
======== Backtrace stop ========
Exit with lock hold.
======== Backtrace start ========
No.1: var Giant(file:../../../cam/scsi/scsi_da.c, line 996) init => locked
No.2: var Giant(file:../../../cam/scsi/scsi_da.c, line 1004) locked => error
======== Backtrace stop ========
Exit with lock hold.
======== Backtrace start ========
No.1: var sc->aac_io_lock(file:../../../dev/aac/aac_cam.c, line 309) init => locked
No.1: var sc->aac_io_lock(file:../../../dev/aac/aac_cam.c, line 417) locked => error
======== Backtrace stop ========

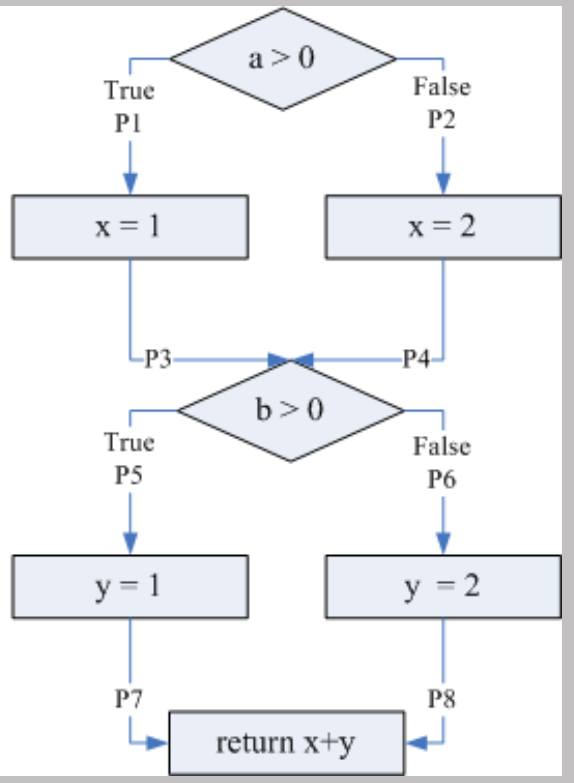
Another example (Cont.)

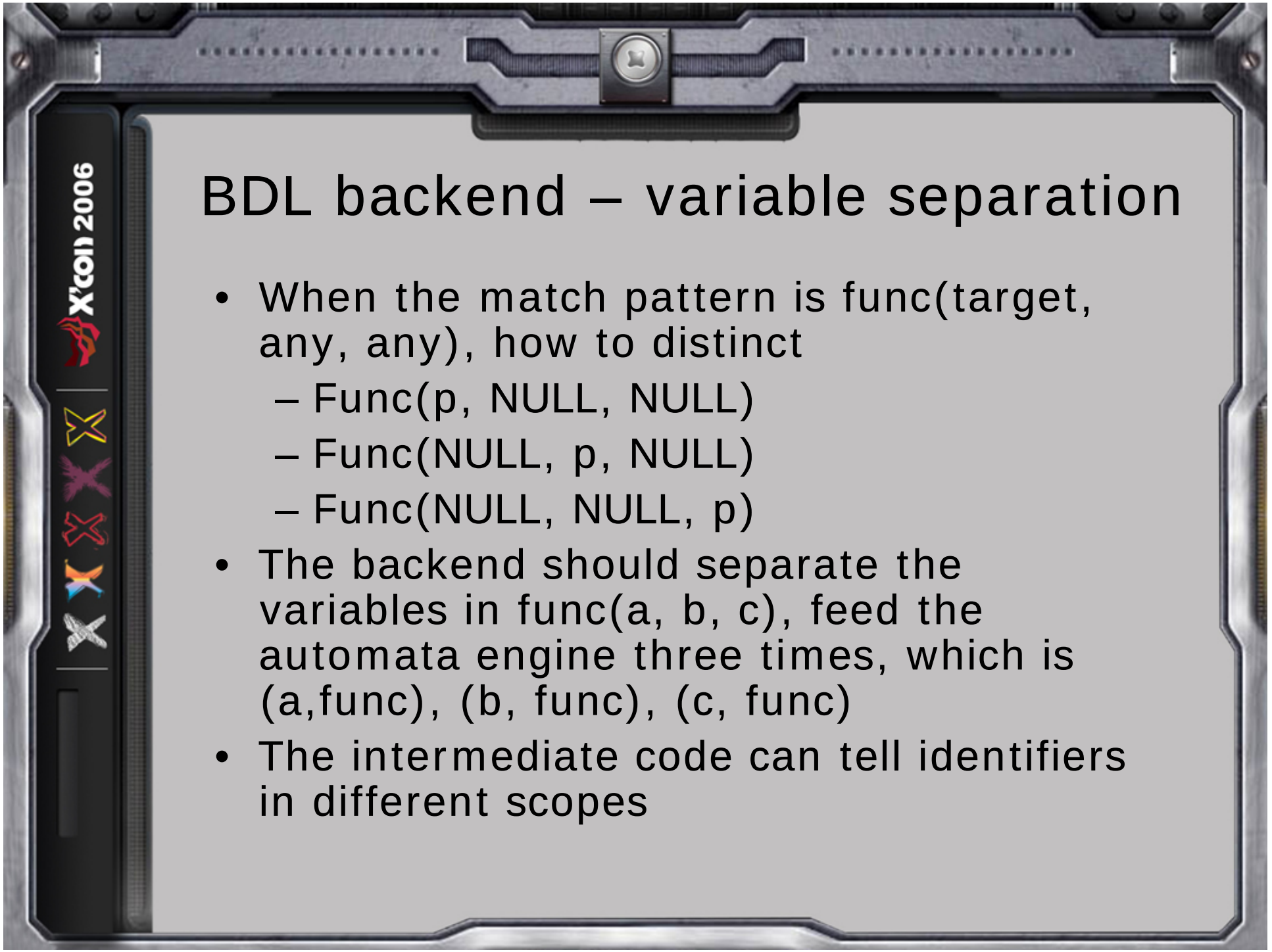
- These three bugs and the corresponding patches have been submitted and received. See: http://www.freebsd.org/cgi/query-pr.cgi?pr=100046
- The coverity company has used its product to check the kernel source code of FreeBSD 6.0, before us, but these three bugs have not be found out.

BDL backend – Path enumeration

- All possible execution path between function entry to exit
- Using DFS algorithm
- The possible paths are:
  P1→P3→P5→P7
  P2→P4→P5→P7
  P1→P3→P6→P8
  P2→P4→P6→P8
- The loops count can be controlled

```
int foo ( int a, int b )
{
    int x, y;

    if ( a > 0 )    x = 1;
    else            x = 2;
    if ( b > 0 )    y = 1;
    else            y = 2;

    return x+y;
}
```



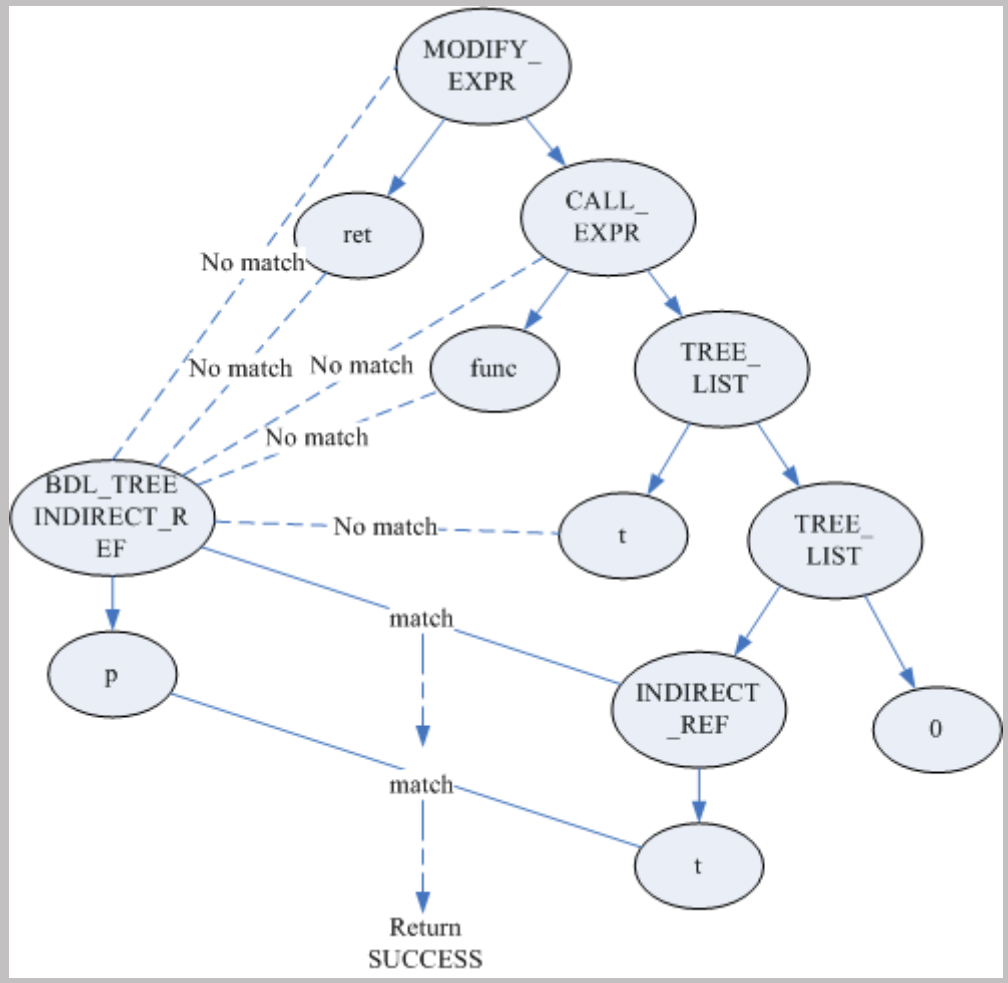


BDL backend – variable separation

- When the match pattern is func(target, any, any), how to distinct
  - Func(p, NULL, NULL)
  - Func(NULL, p, NULL)
  - Func(NULL, NULL, p)
- The backend should separate the variables in func(a, b, c), feed the automata engine three times, which is (a,func), (b, func), (c, func)
- The intermediate code can tell identifiers in different scopes

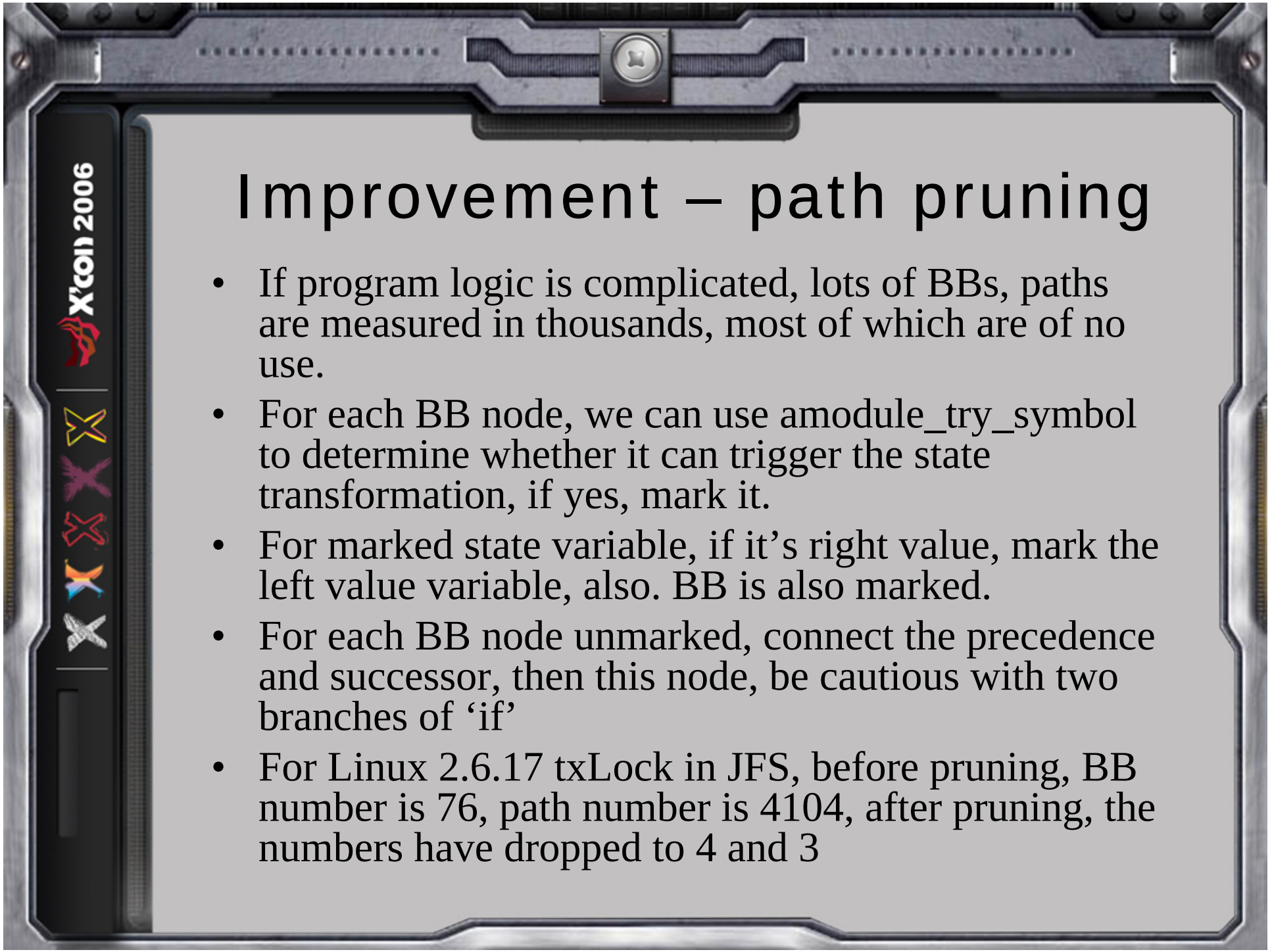
BDL built-in functions

- Pattern match function
  - Recursively match
  - Function match separately
  - An example to match ret = func(t, *t, 0)
- Action function
- Halt function
- Backtrace function



BDL automata engine

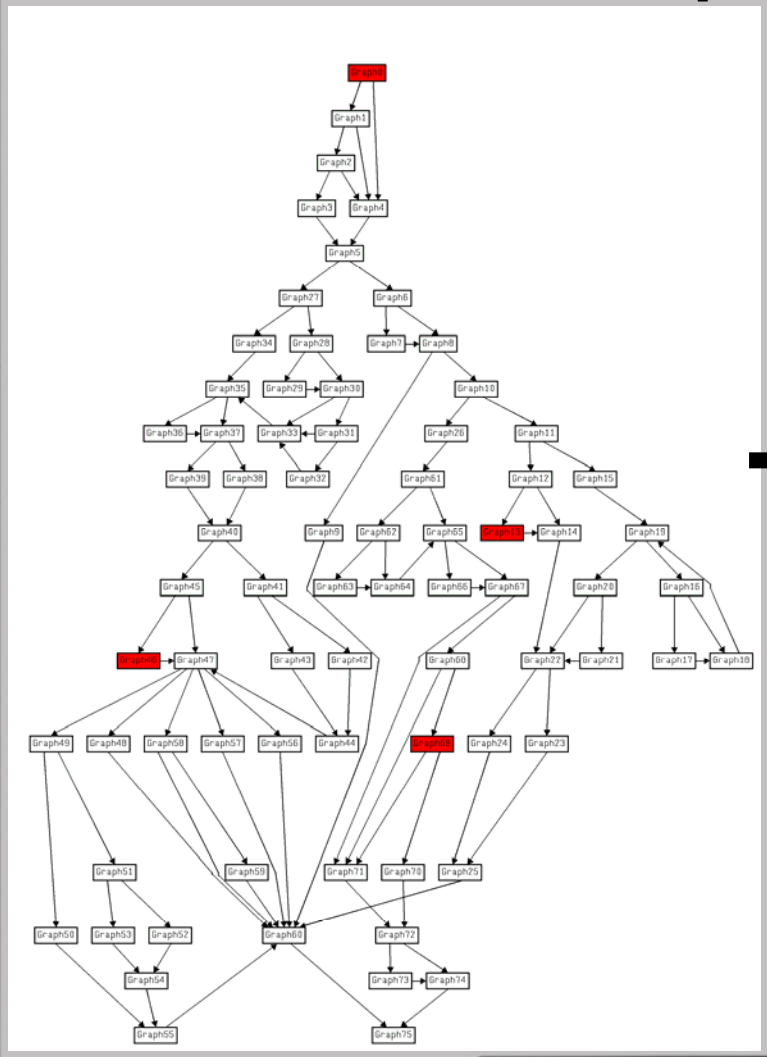
- Generate instance for each variable
- Get transformation rules from BDL front-end
- Get streams of target variable and expression from BDL backend
- Call built-in pattern match function to decide the transformation
- When automata halts, built-in halt function is called to give bug infomation

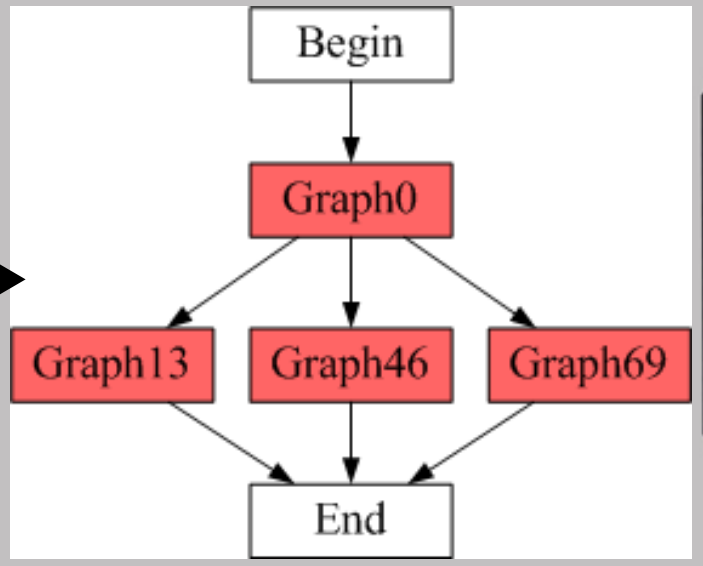


Improvement – path pruning

- If program logic is complicated, lots of BBs, paths are measured in thousands, most of which are of no use.
- For each BB node, we can use amodule_try_symbol to determine whether it can trigger the state transformation, if yes, mark it.
- For marked state variable, if it's right value, mark the left value variable, also. BB is also marked.
- For each BB node unmarked, connect the precedence and successor, then this node, be cautious with two branches of 'if'
- For Linux 2.6.17 txLock in JFS, before pruning, BB number is 76, path number is 4104, after pruning, the numbers have dropped to 4 and 3

An example of pruning





Improvement – global analysis

- Till now, only intra-procedure analysis is preformed, in real life, inter-procedure analysis is also needed.
- GCC4 takes one file as a unit to compile, which means that all the AST of one file will be freed when it begin to compile another file.
- Two different thoughts:

(1) Use cc1 directly.

(2) Dump the intermediate codes produced by GCC4 into a file, and reconstruct the AST and CFG.

Improvement – cache and annotation

- The path pruning is done with BB, but within BB, the optimization is still possible.
- For a specified BB and variable instance sti, before entering BB, state is s_x , after leaving BB, state is S_y, then <sti, s_x, S_y > is a triple, i.e. S_y is decided by sti and s_x
- So, each BB has a cache, when the transformation appears, the triple is put into cache, next time, try to fetch the result directly
- You can't store annotation information(//, /* */) in GCC4 intermediate codes
  - We can use __attribute__ keyword, add support for "bdl"

What's left?

- Combined with some dynamic analysis?
- Pointer analysis much better?
- Detect script vulnerabilities?
- Learn from commercial products?

Thanks!