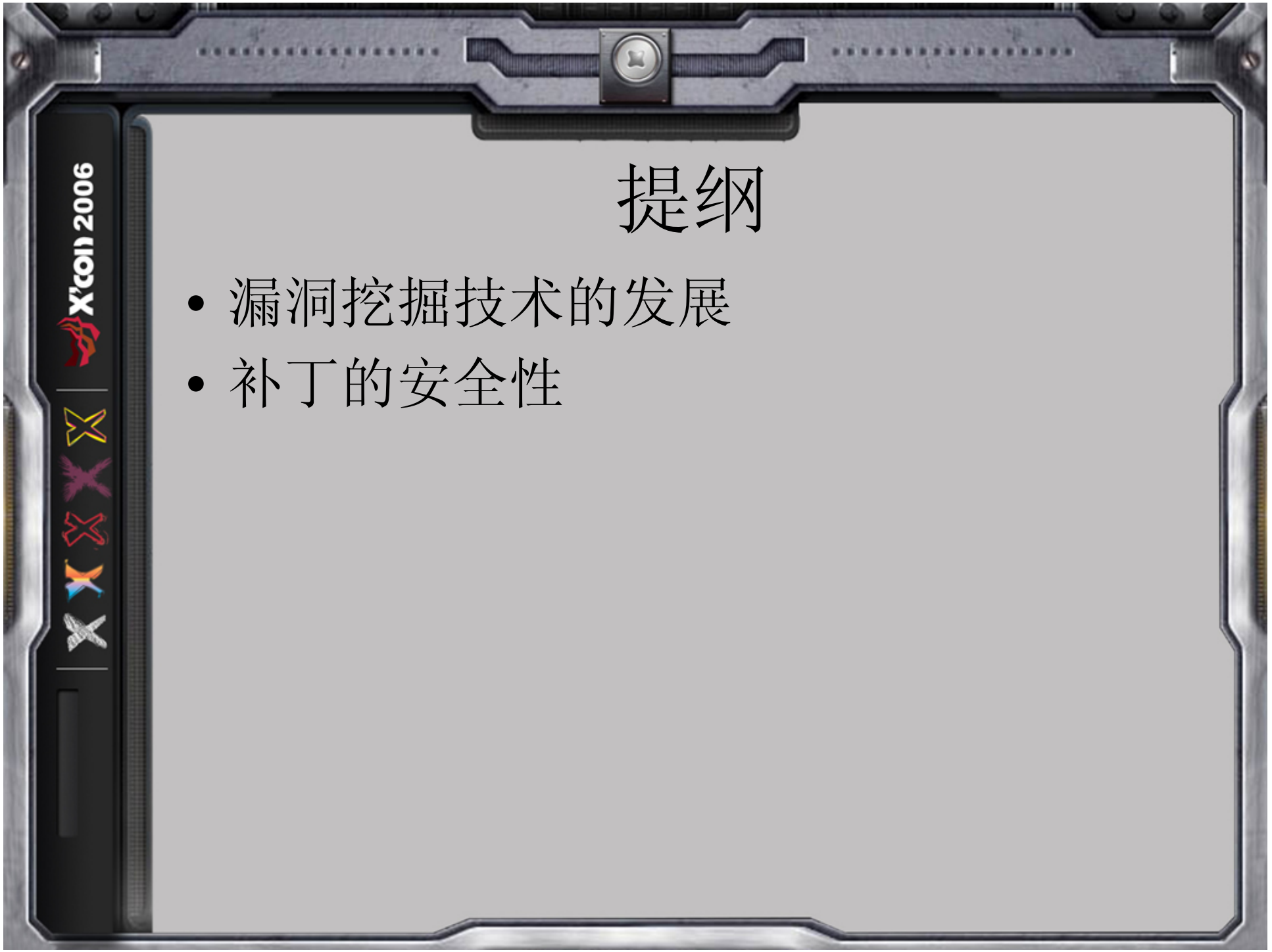

漏洞挖掘的过去、现在、未来
Funnywei
X'Con2006



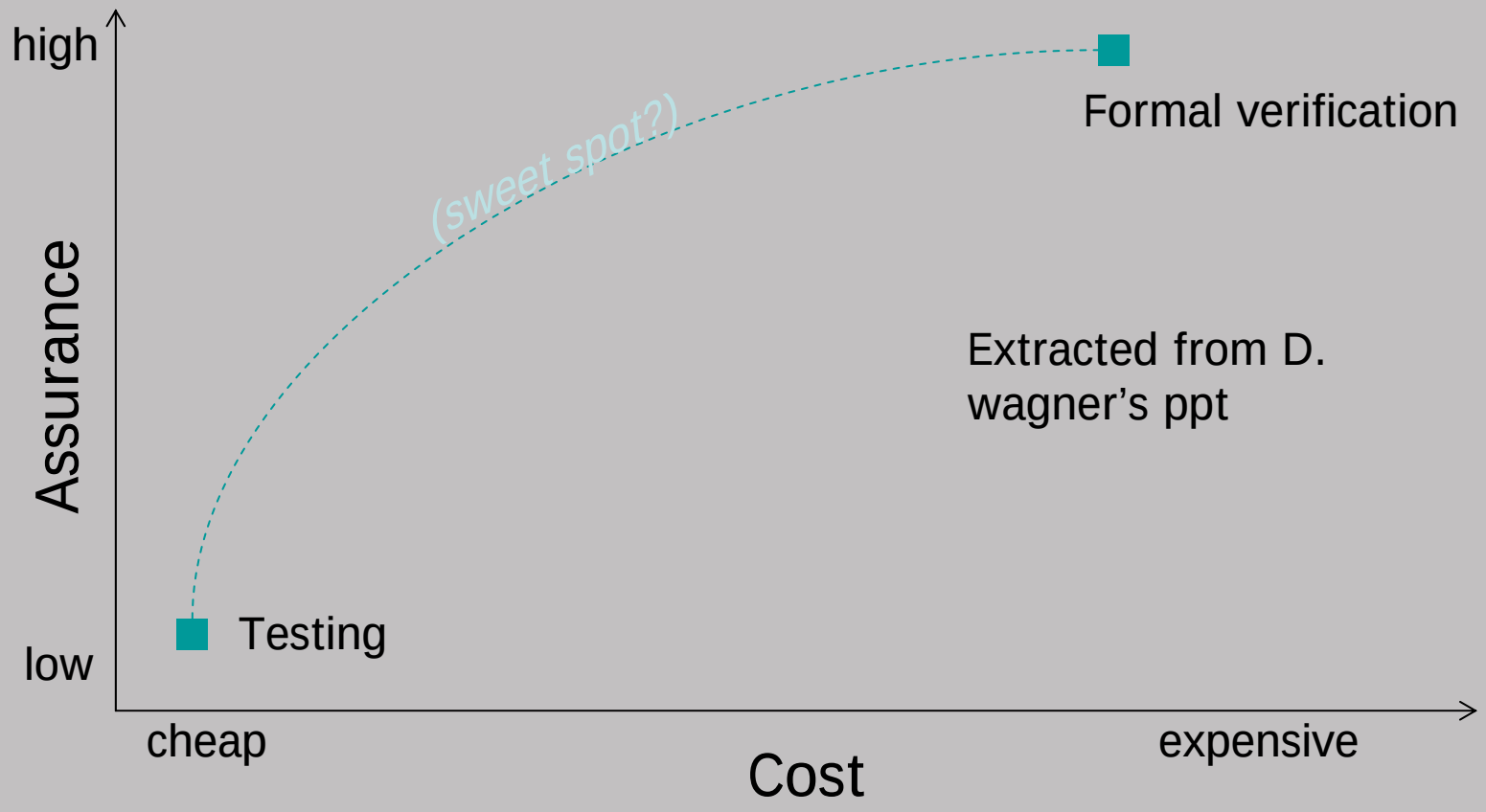
提纲

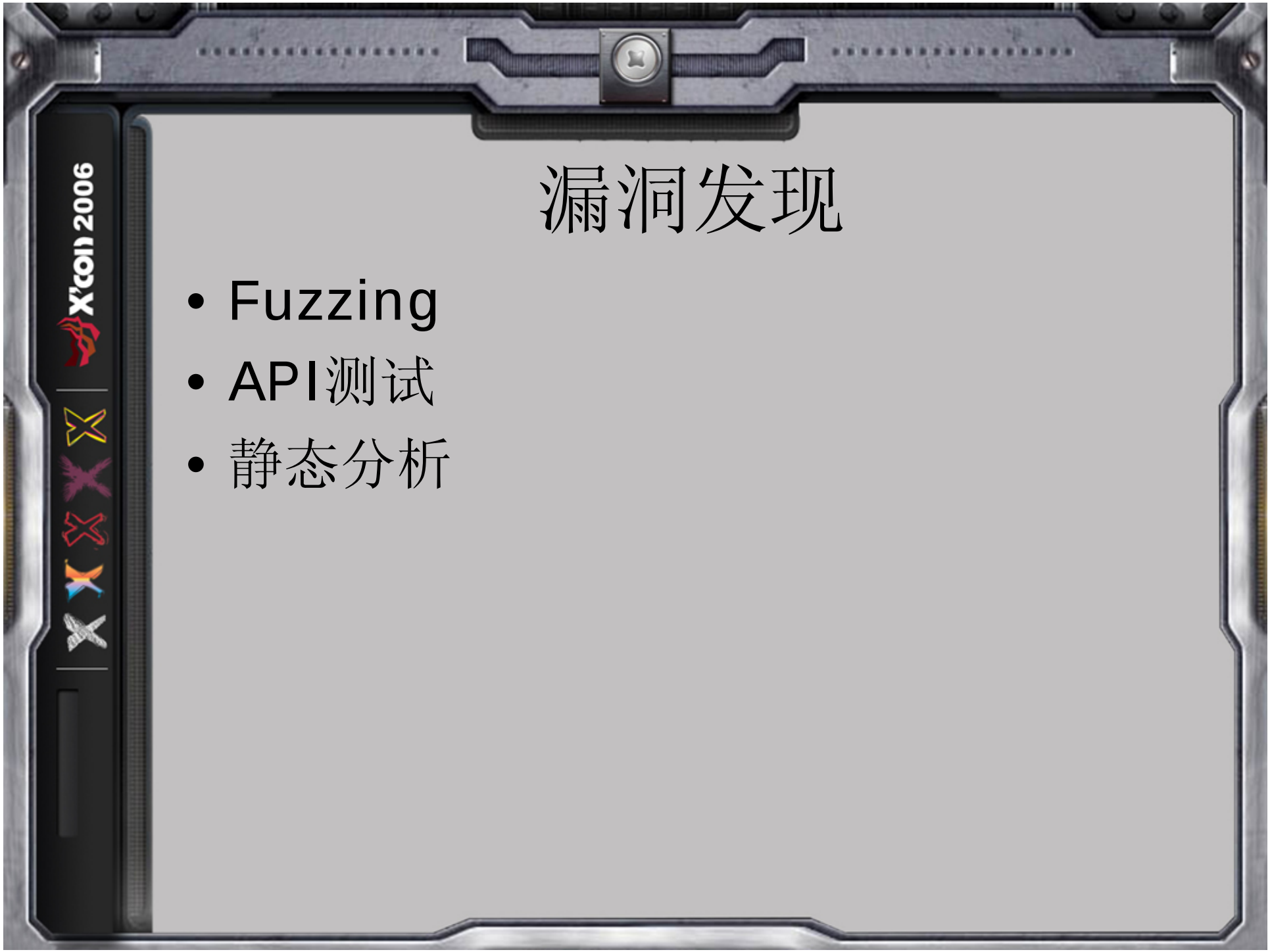
- 漏洞挖掘技术的发展
- 补丁的安全性

漏洞挖掘技术的发展

- 过去→现在
- 未来

Existing Paradigms





漏洞发现

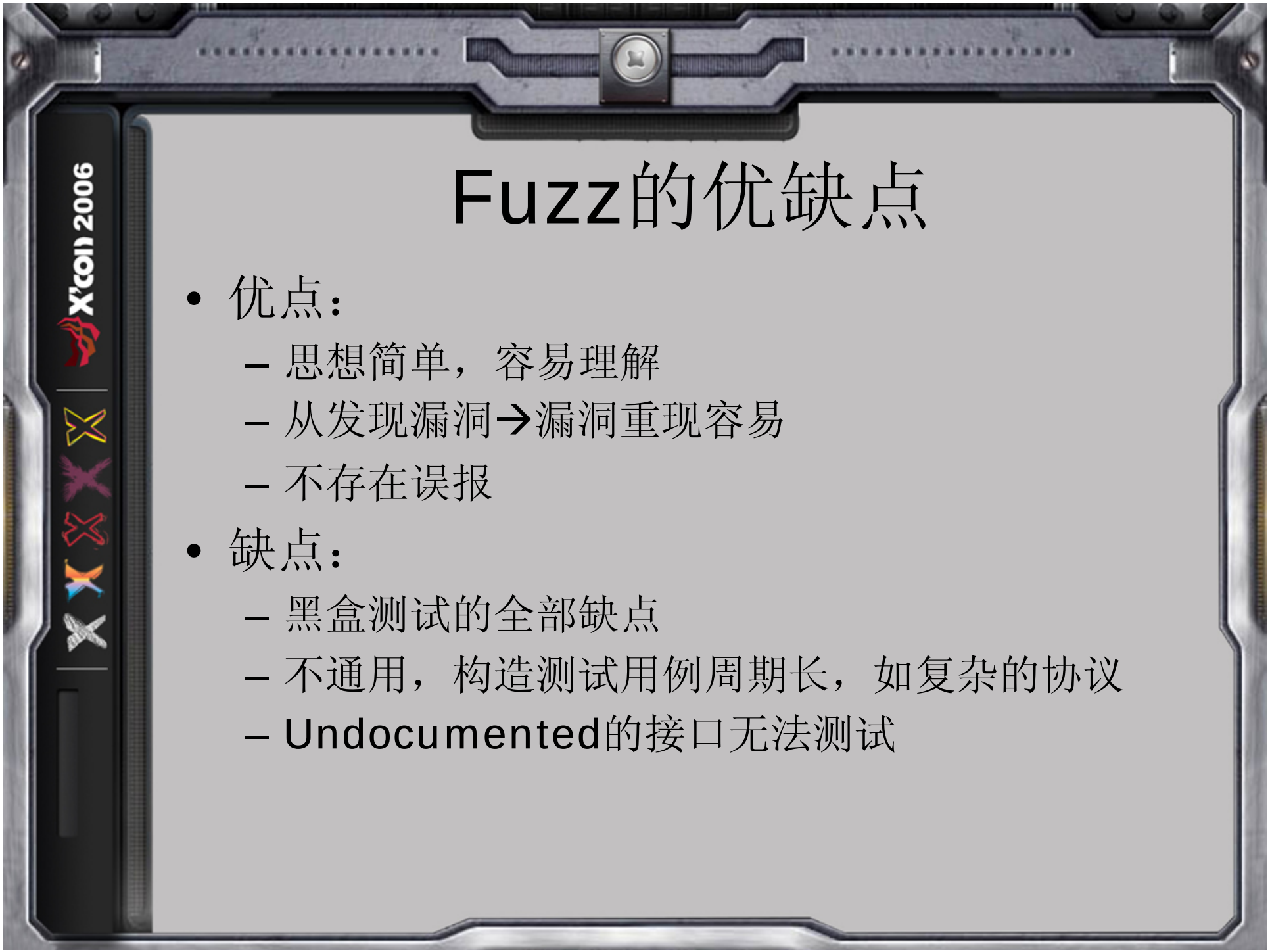
- Fuzzing
- API测试
- 静态分析

Fuzzing的目标

- File Formats
- Network Protocols
- Web applications
- Environment variables
- COM Objects
- IPCs

Fuzz技术

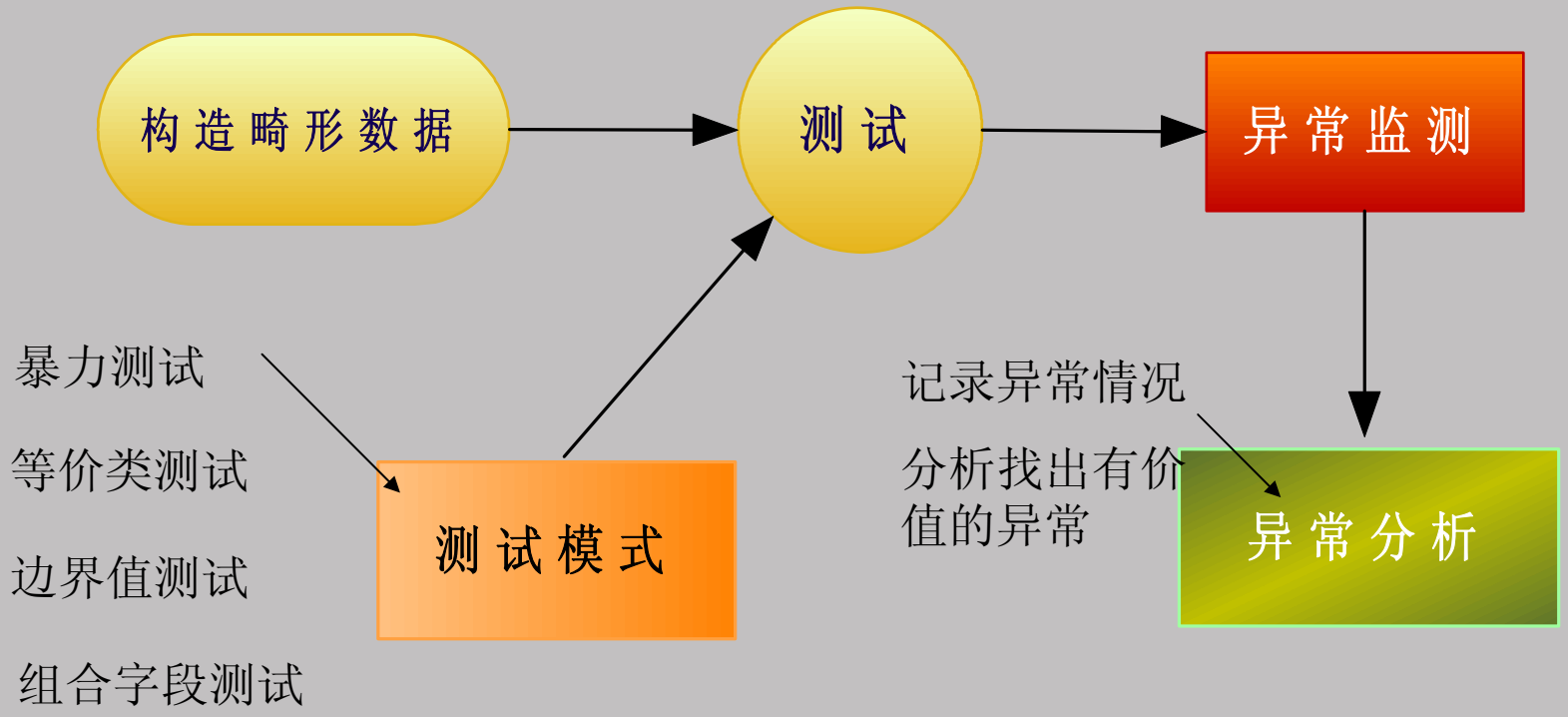
- 文件格式的Fuzz
  - 图像格式
  - 文档格式
  - 等等
- 协议的Fuzz
  - RPC协议
  - Http协议
  - 等等

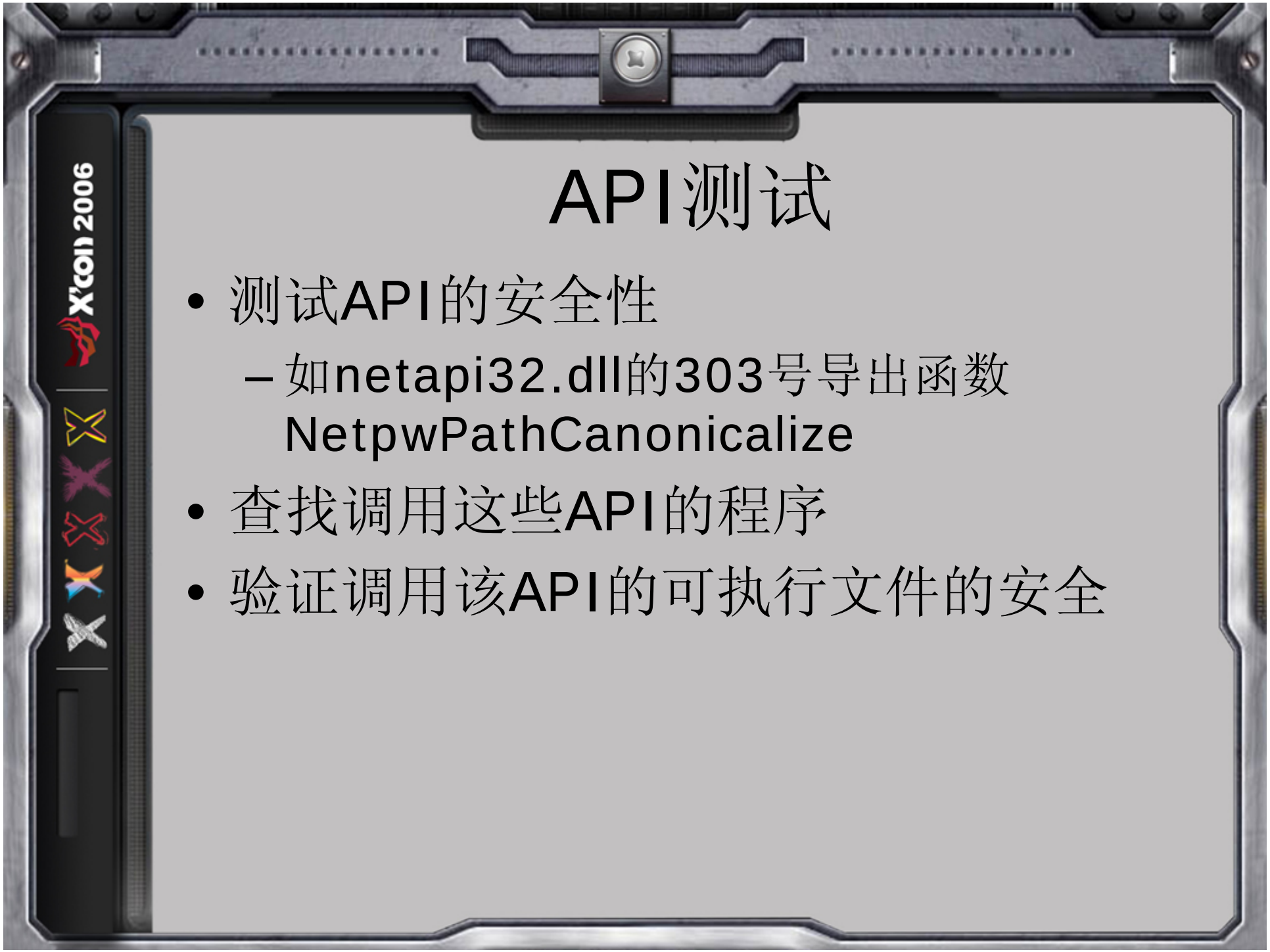


Fuzz的优缺点

- 优点：
  - 思想简单，容易理解
  - 从发现漏洞→漏洞重现容易
  - 不存在误报
- 缺点：
  - 黑盒测试的全部缺点
  - 不通用，构造测试用例周期长，如复杂的协议
  - Undocumented的接口无法测试

文件格式的Fuzzer框架





API测试

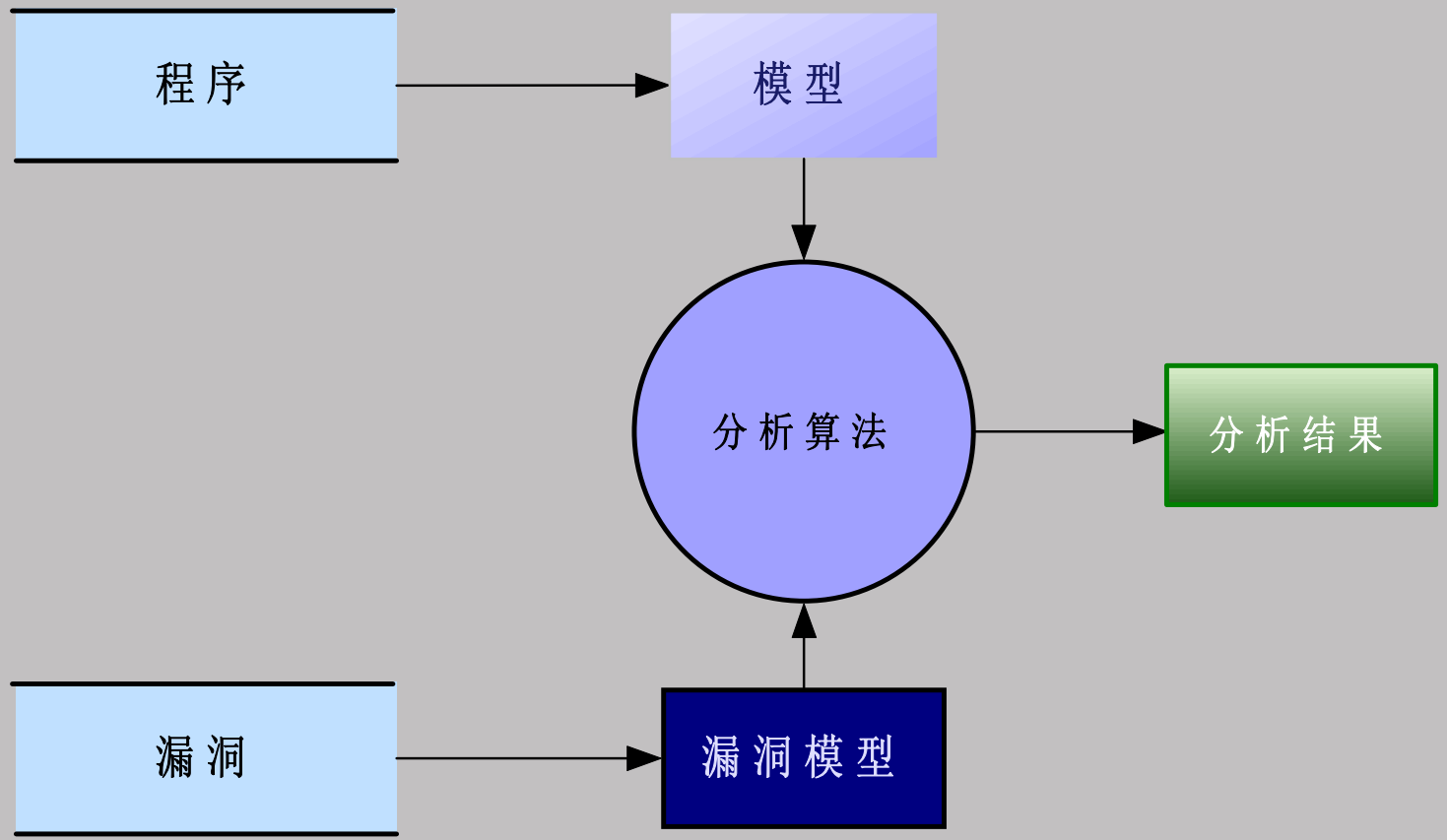
- 测试API的安全性
  - 如netapi32.dll的303号导出函数 NetpwPathCanonicalize
- 查找调用这些API的程序
- 验证调用该API的可执行文件的安全

静态分析技术

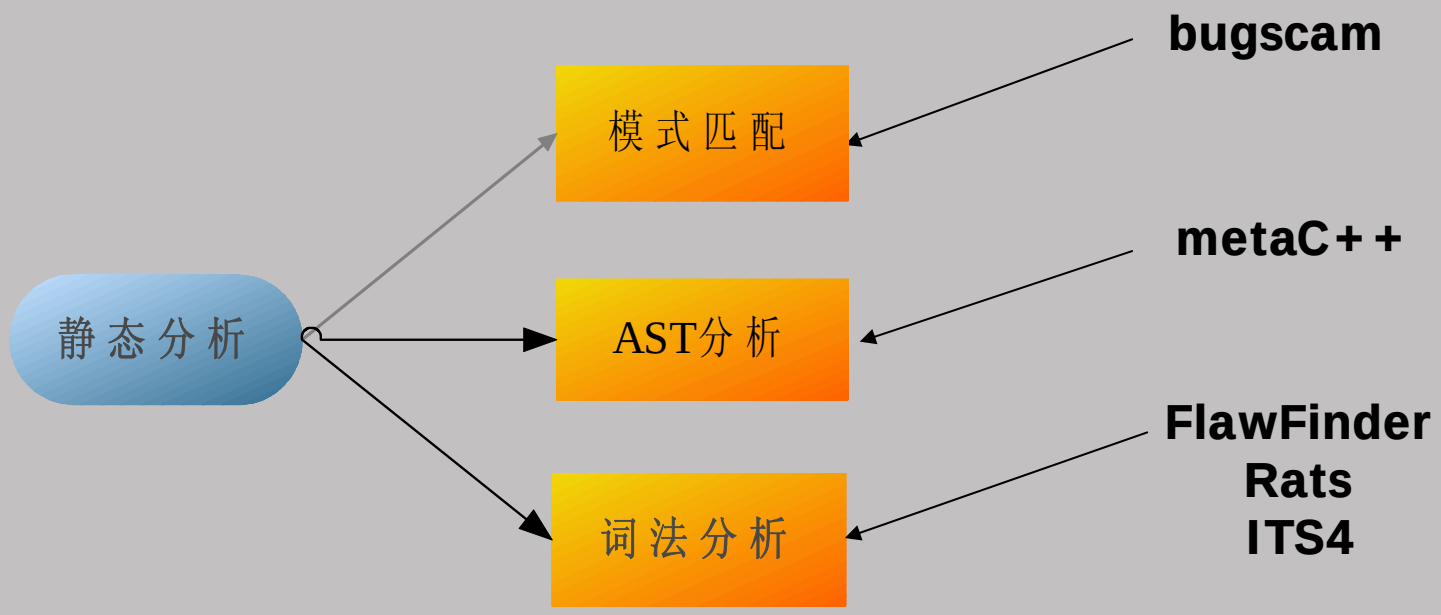
- 提出问题：

  给定程序和安全属性，验证程序是否满足属性，如果不满足，找出原因。

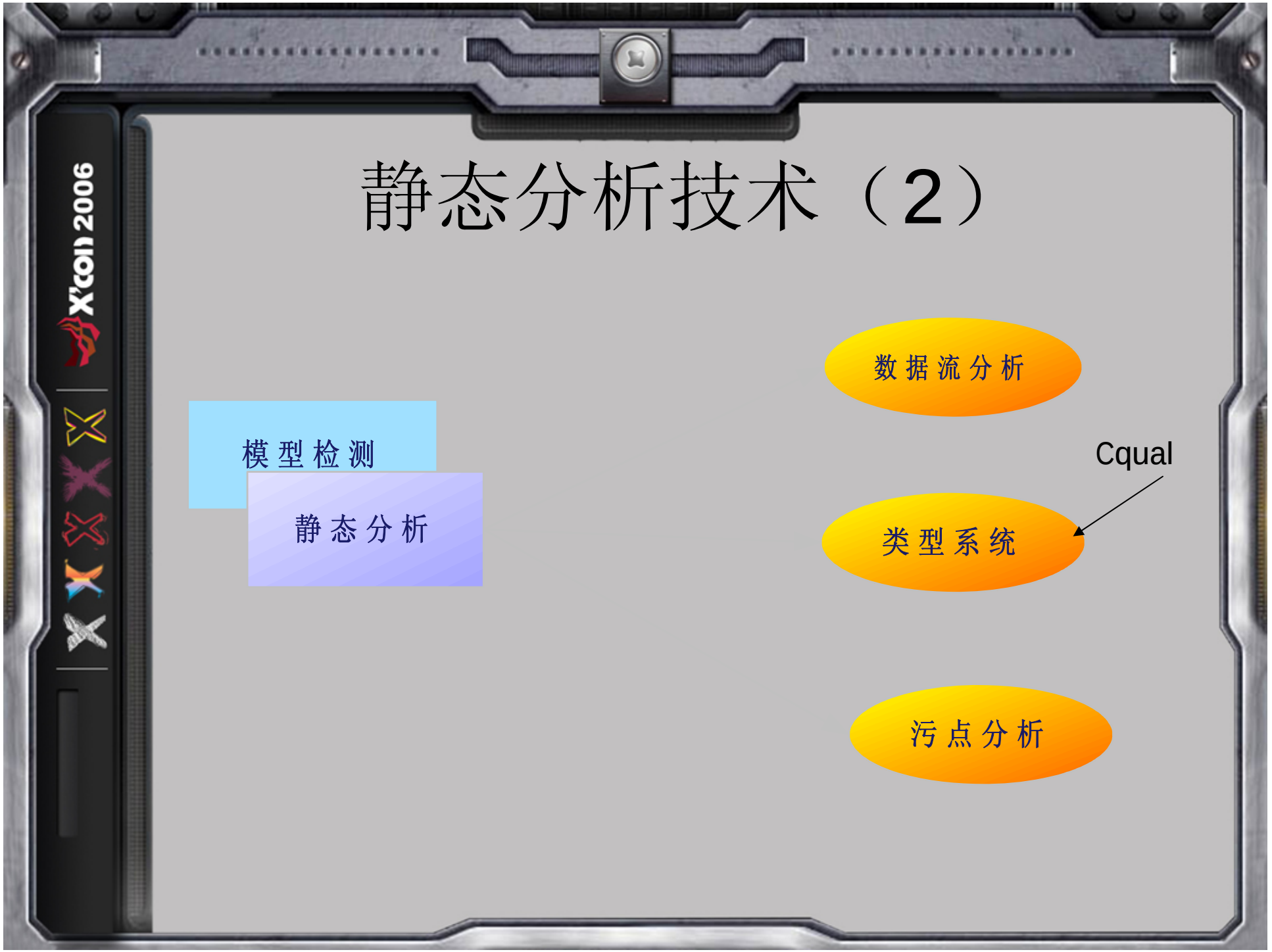
静态分析框架



静态分析技术（1）



静态分析技术（2）



漏洞的建模

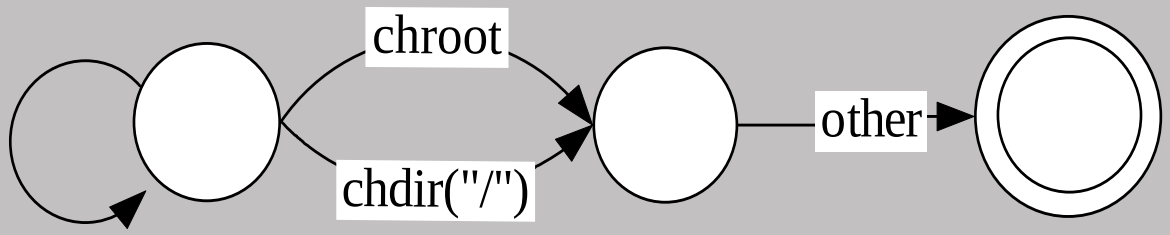
- 时序逻辑模型
- 约束模型
- 类型限定模型
- 依赖图模型

时序逻辑模型

- 将安全相关的操作序列描述为时序安全逻辑
- 采用FSA描述
- 可以参考”Model Checking One Million Lines of C Code”

时序安全属性的简单例子

- 在调用chroot之后，进程应立即调用chdir(“/”)来切换到根目录下。



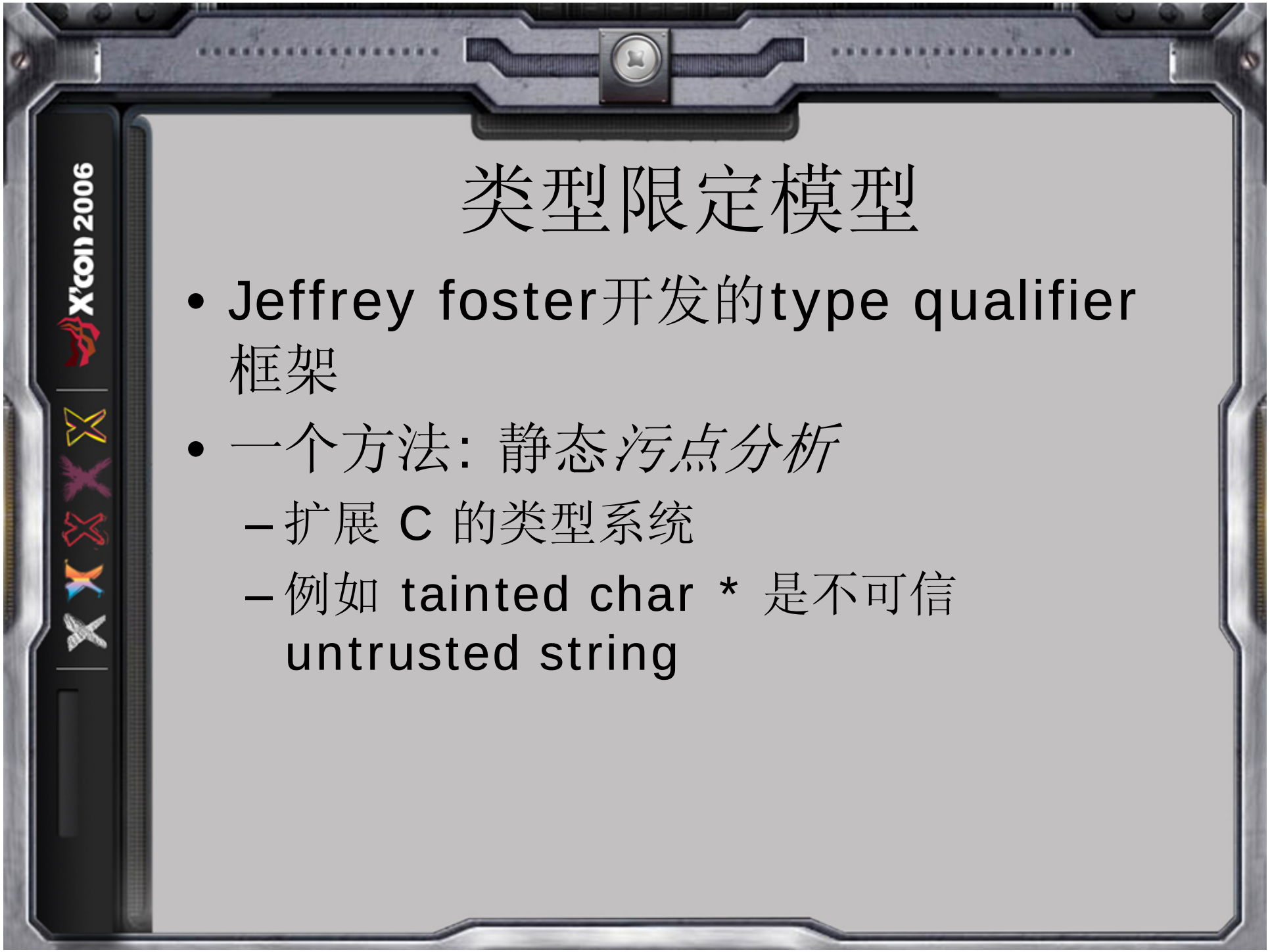
约束模型

David Wagner等人将缓冲区溢出的检测问题规范化为整数约束的问题，并定义了约束语言（Constraint Language）。

线形约束条件的产生

符号len(s)表示当前使用的长度（包含结束字符'\0'），范围属性为[a, b]。Alloc(s)表示buffer实际分配的大小,范围属性为[c, d]。
在对所有的变量进行了范围推断之后，再进行安全属性检查：

如果b<c，溢出不可能发生。
如果a>d，那么肯定发生。
如果d>b>c>a, 那么溢出有可能发生。



类型限定模型

- Jeffrey foster开发的type qualifier 框架
- 一个方法: 静态*污点分析*
  - 扩展 C 的类型系统
  - 例如 tainted char * 是不可信 untrusted string

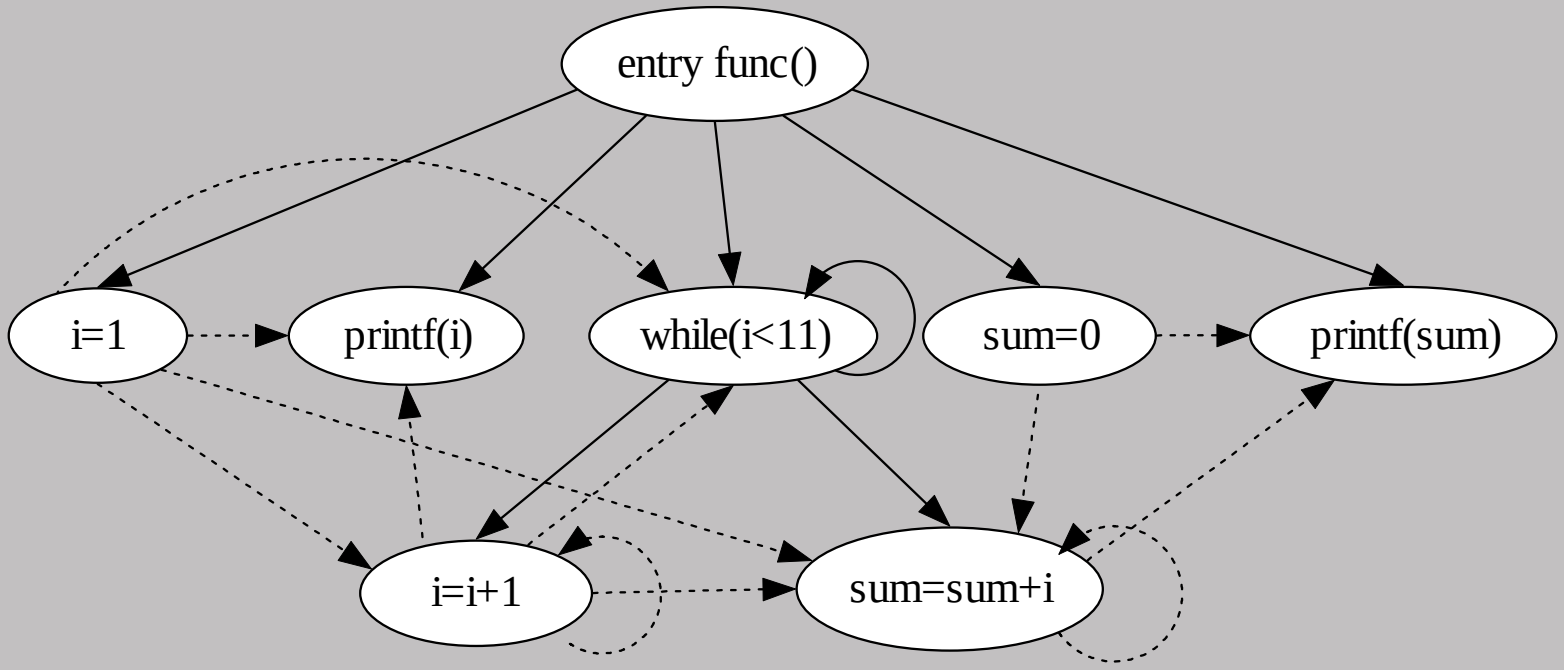
基于依赖图的分析

- **PDG是1984年Ottenstein等人提出，用于intra-procedural的中间表示形式**
- 什么是依赖图？
  - 节点--表示语句
  - 边表示--控制流和数据流
    - 虚线边表示数据流依赖
    - 实线边表示控制流依赖

PDG的简单例子

```
void func()
{
    int sum = 0, i = 1;
    while (i < 11)
    {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}
```

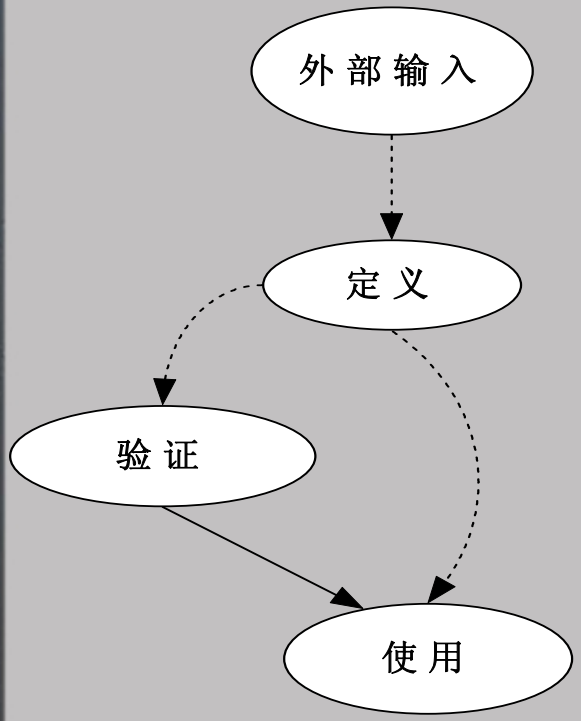
PDG的简单例子



SDG

- 1990年，Horwitz等人提出。
- 从函数A到B的调用，在A中增加一个Call的顶点，B中增加一个入口点，并且增加一条从A到B的inter-procedural的控制依赖边。
- 在A中增加进出的实参顶点，B中增加进出的形参顶点。并且增加interprocedural的数据依赖边。
- 如果有全局变量，则看作输入参数，并被编码为实参和形参顶点。

整型错误的检测（1）



正确编码模式

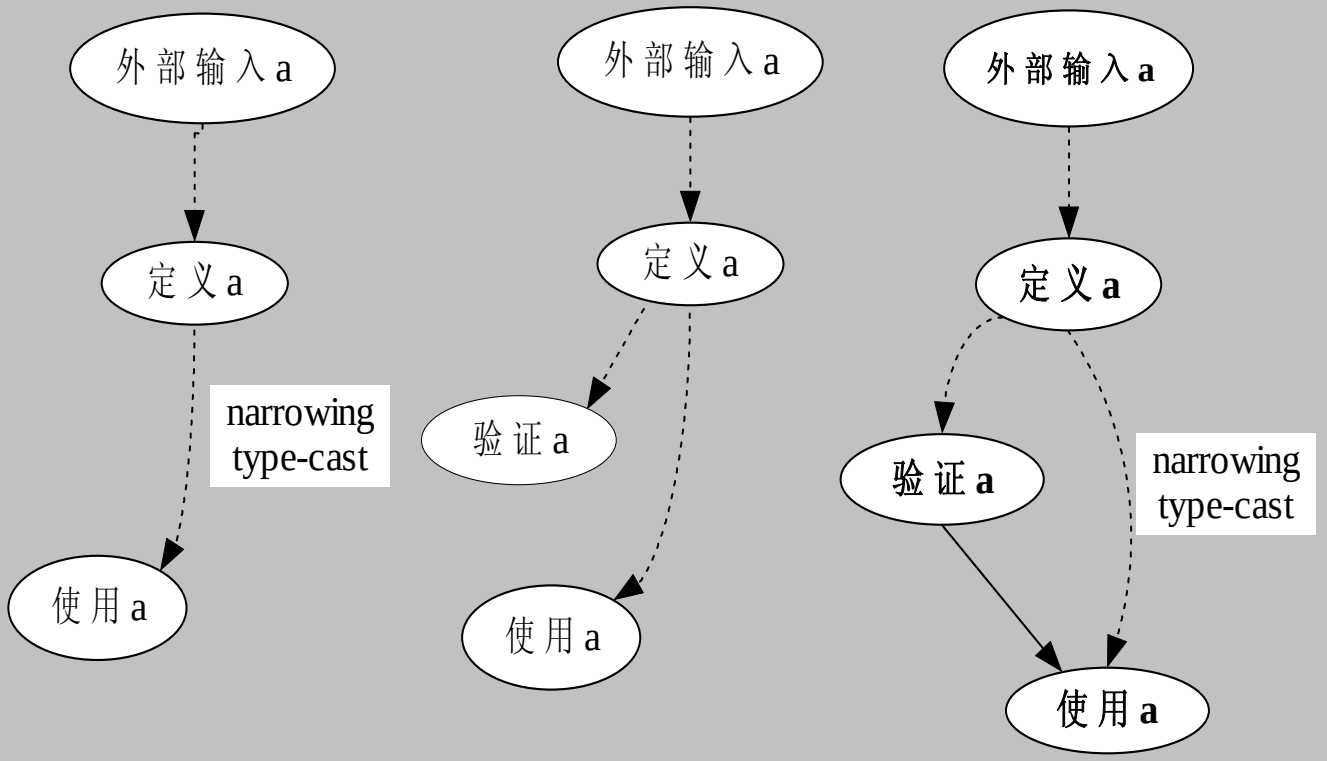
所有的使用都必须经过验证

→使用控制依赖于验证

所有的使用都必须有定义

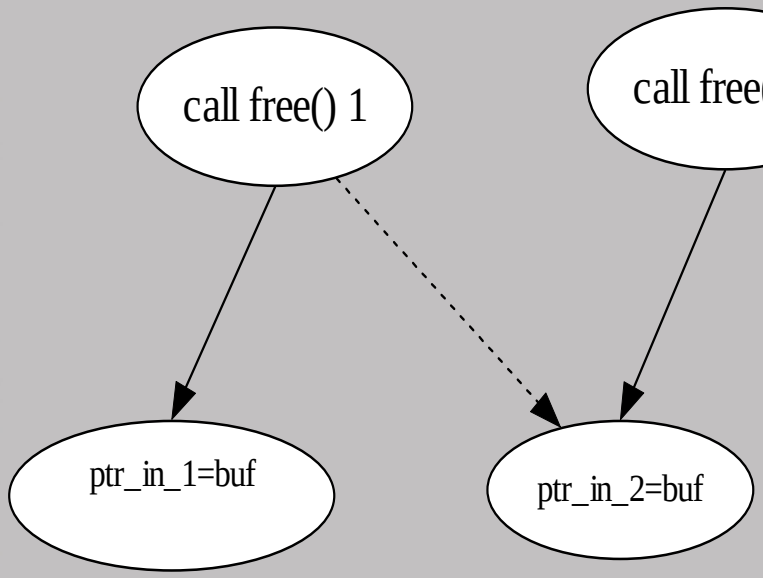
→使用数据流依赖于定义

整型错误的检测（2）



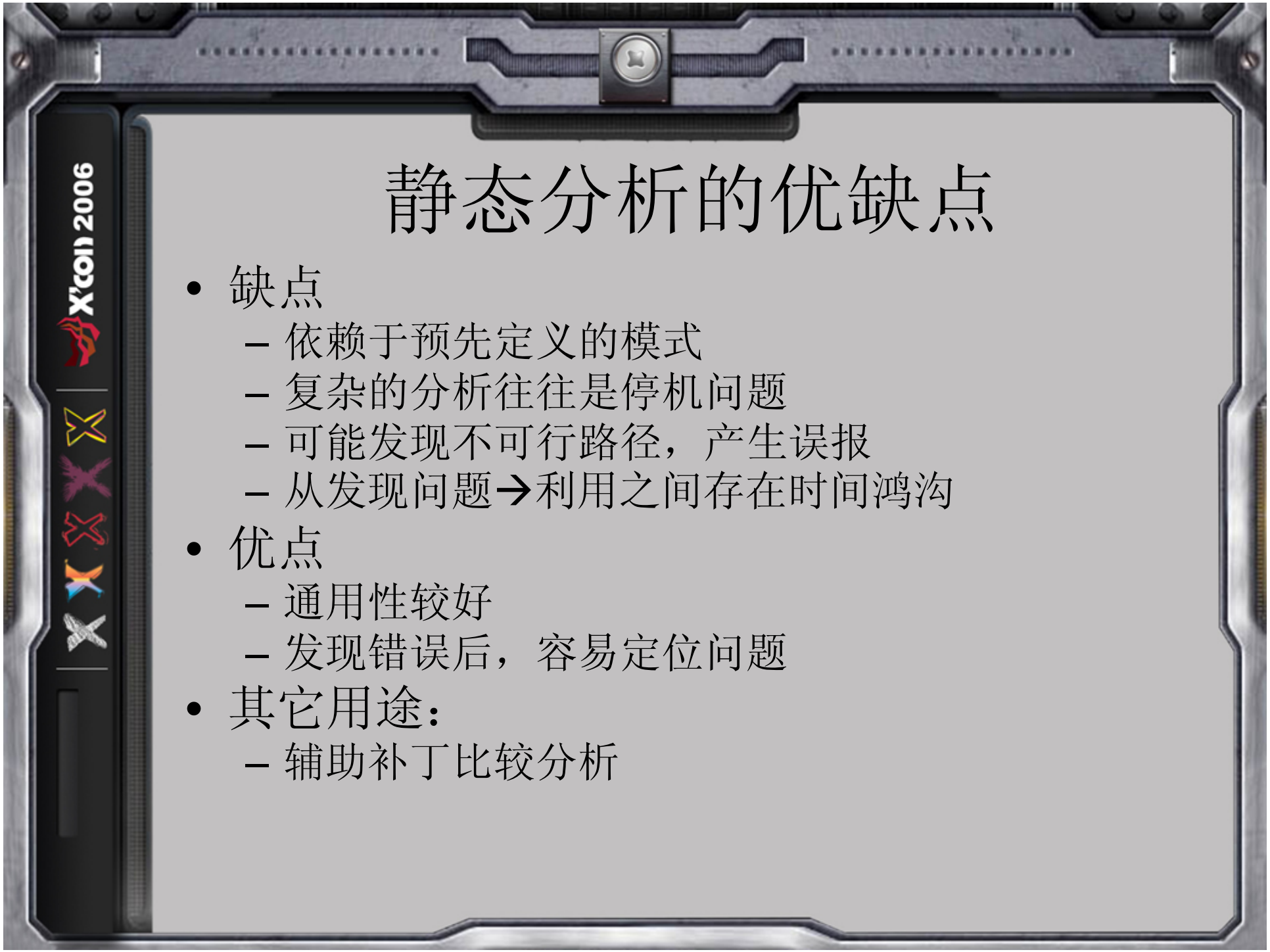
错误编码模式（只举了部分）

两次释放漏洞



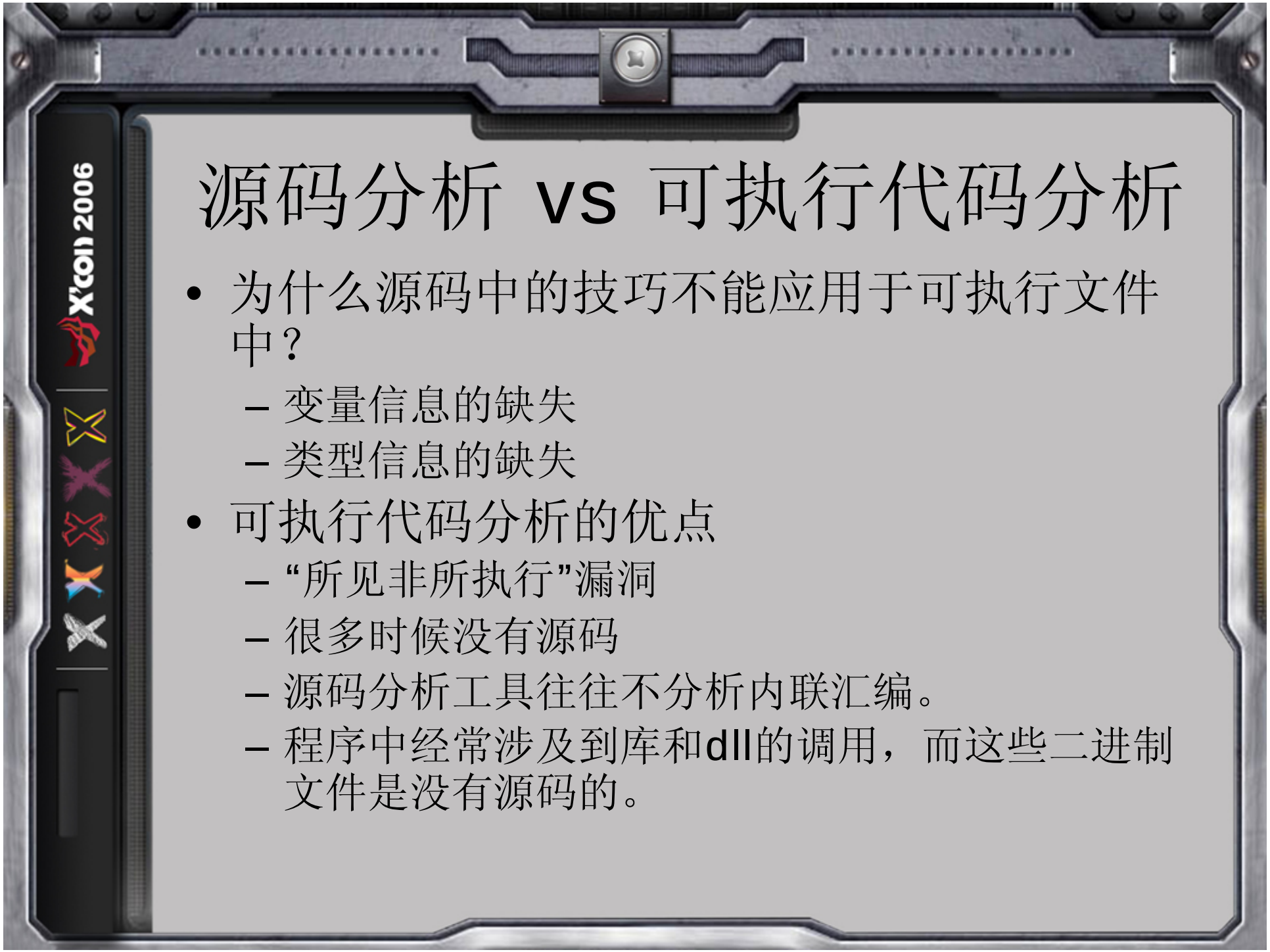

```
char * buf = (char*)malloc(1024);
...
free(buf);
...
free(buf);
```

错误编码模式



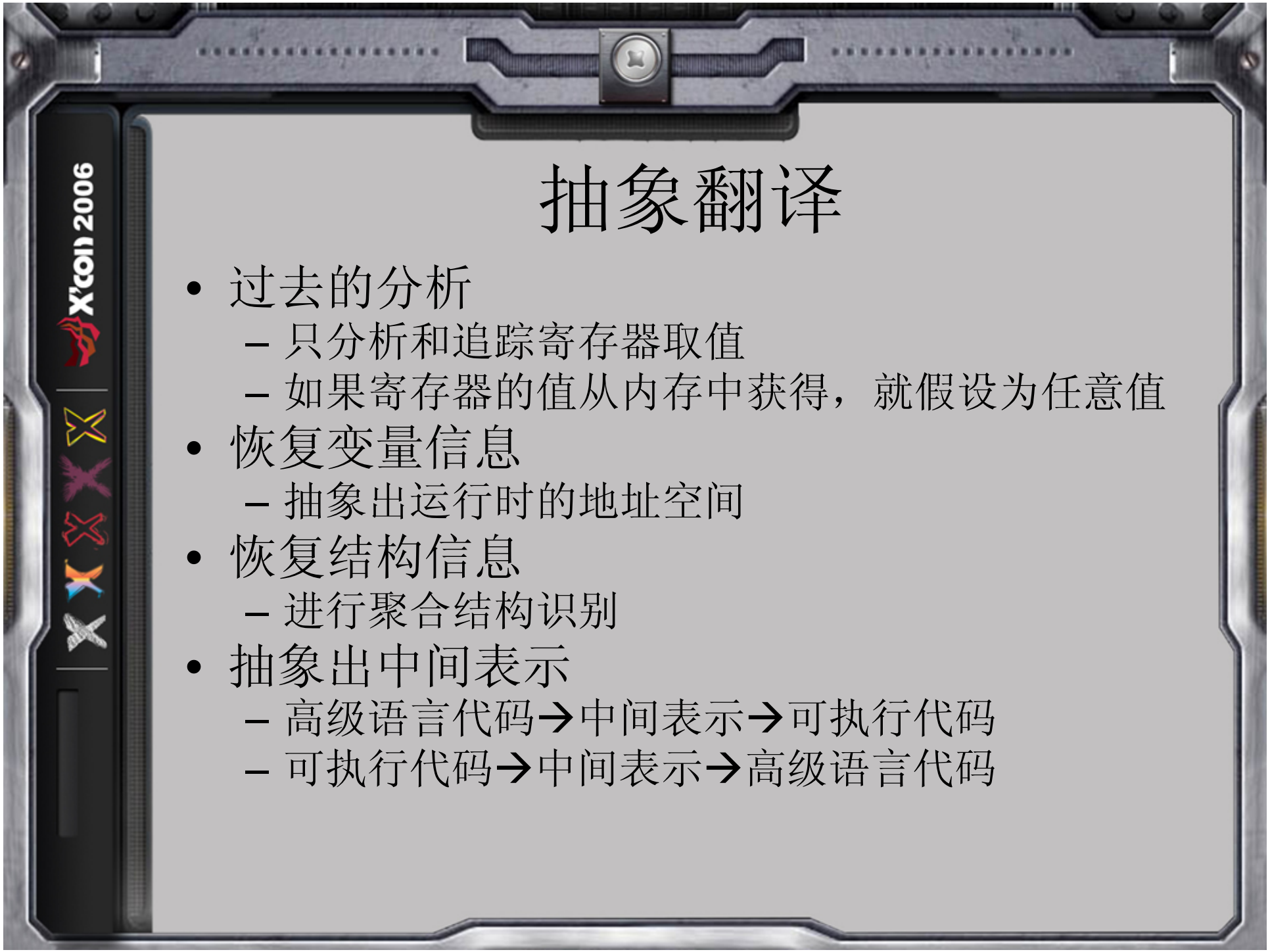
静态分析的优缺点

- 缺点
  - 依赖于预先定义的模式
  - 复杂的分析往往是停机问题
  - 可能发现不可行路径，产生误报
  - 从发现问题→利用之间存在时间鸿沟
- 优点
  - 通用性较好
  - 发现错误后，容易定位问题
- 其它用途：
  - 辅助补丁比较分析



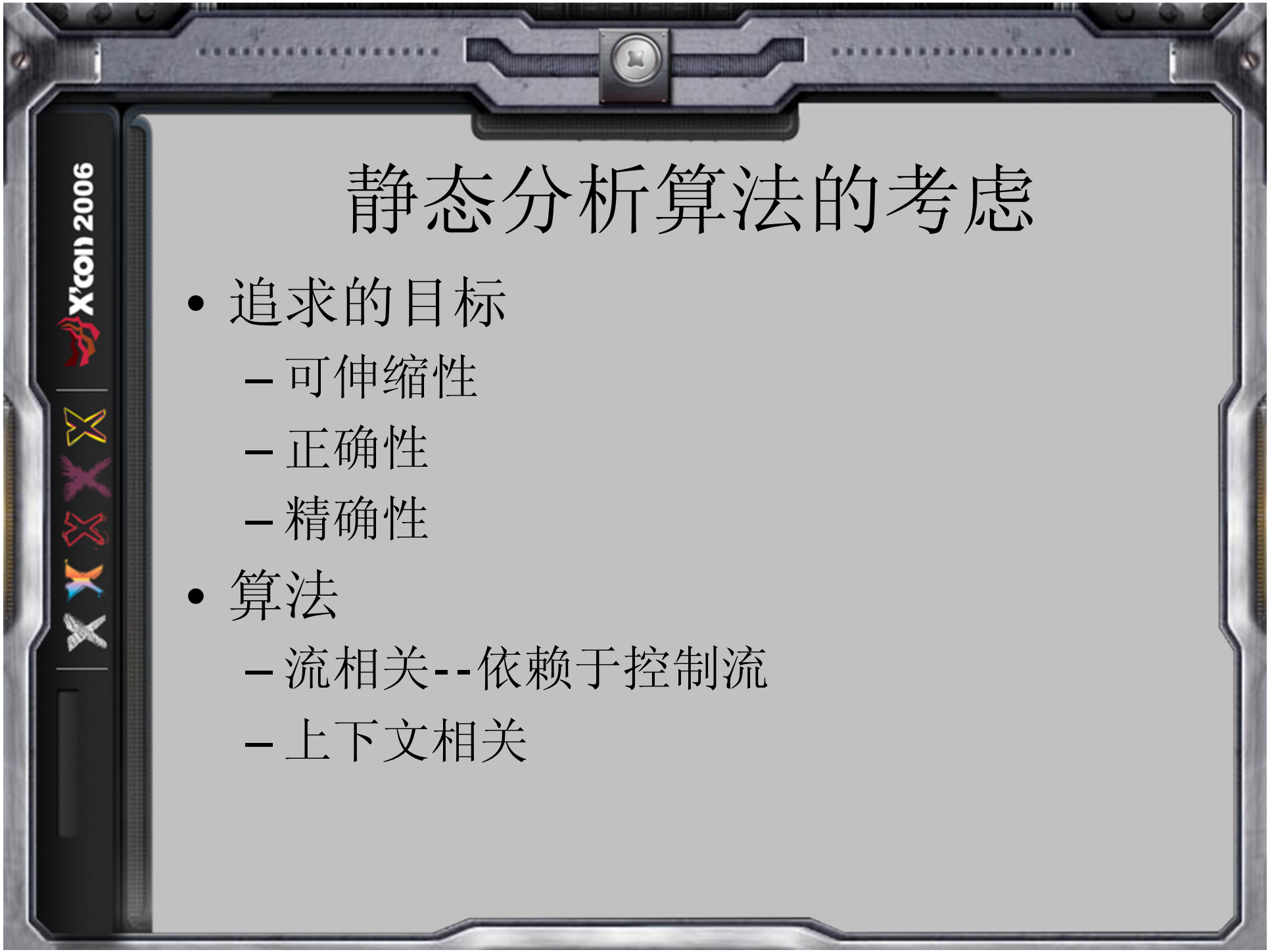
源码分析 vs 可执行代码分析

- 为什么源码中的技巧不能应用于可执行文件中？
  - 变量信息的缺失
  - 类型信息的缺失
- 可执行代码分析的优点
  - “所见非所执行”漏洞
  - 很多时候没有源码
  - 源码分析工具往往不分析内联汇编。
  - 程序中经常涉及到库和dll的调用，而这些二进制文件是没有源码的。



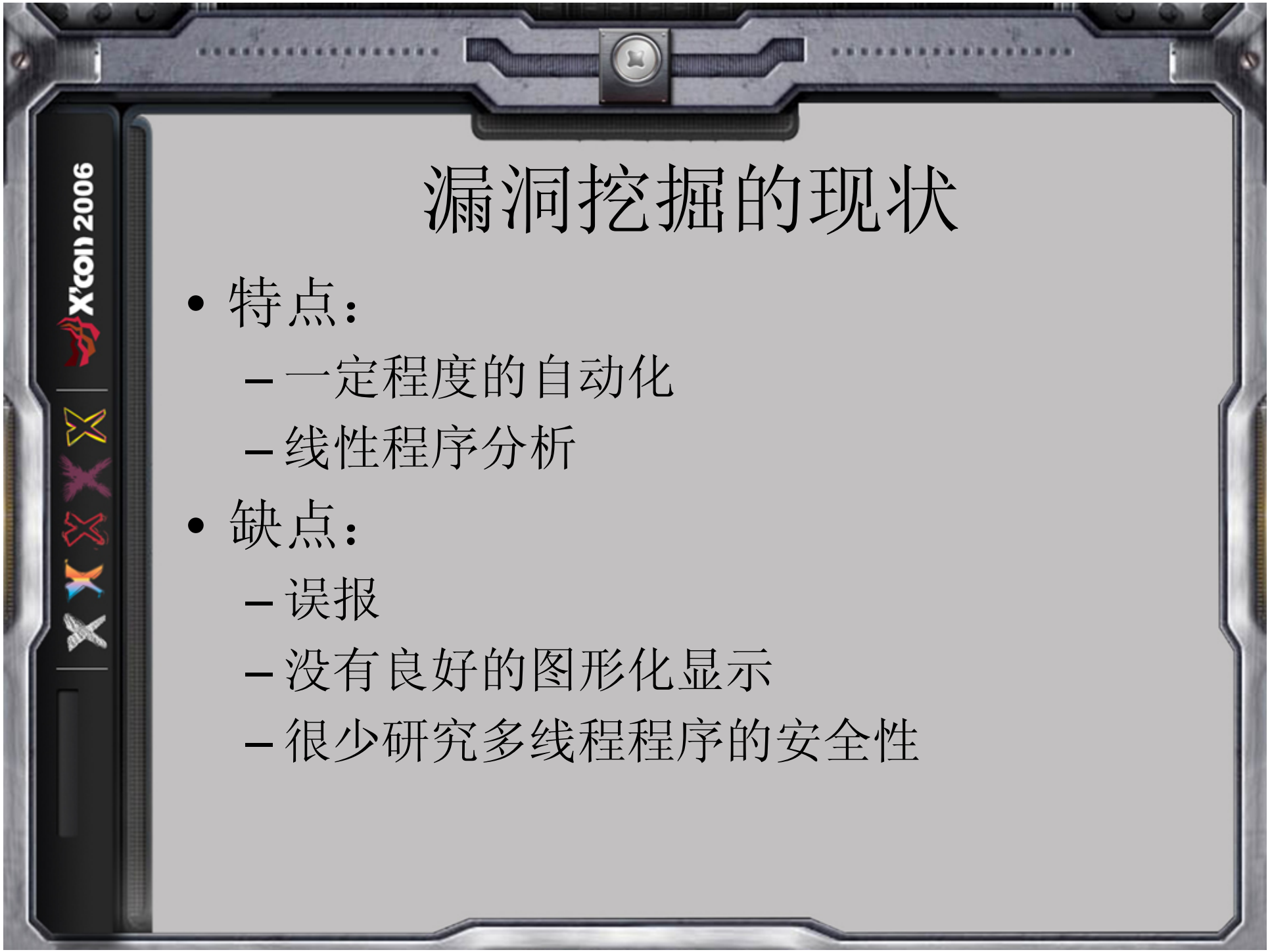
抽象翻译

- 过去的分析
  - 只分析和追踪寄存器取值
  - 如果寄存器的值从内存中获得，就假设为任意值
- 恢复变量信息
  - 抽象出运行时的地址空间
- 恢复结构信息
  - 进行聚合结构识别
- 抽象出中间表示
  - 高级语言代码→中间表示→可执行代码
  - 可执行代码→中间表示→高级语言代码



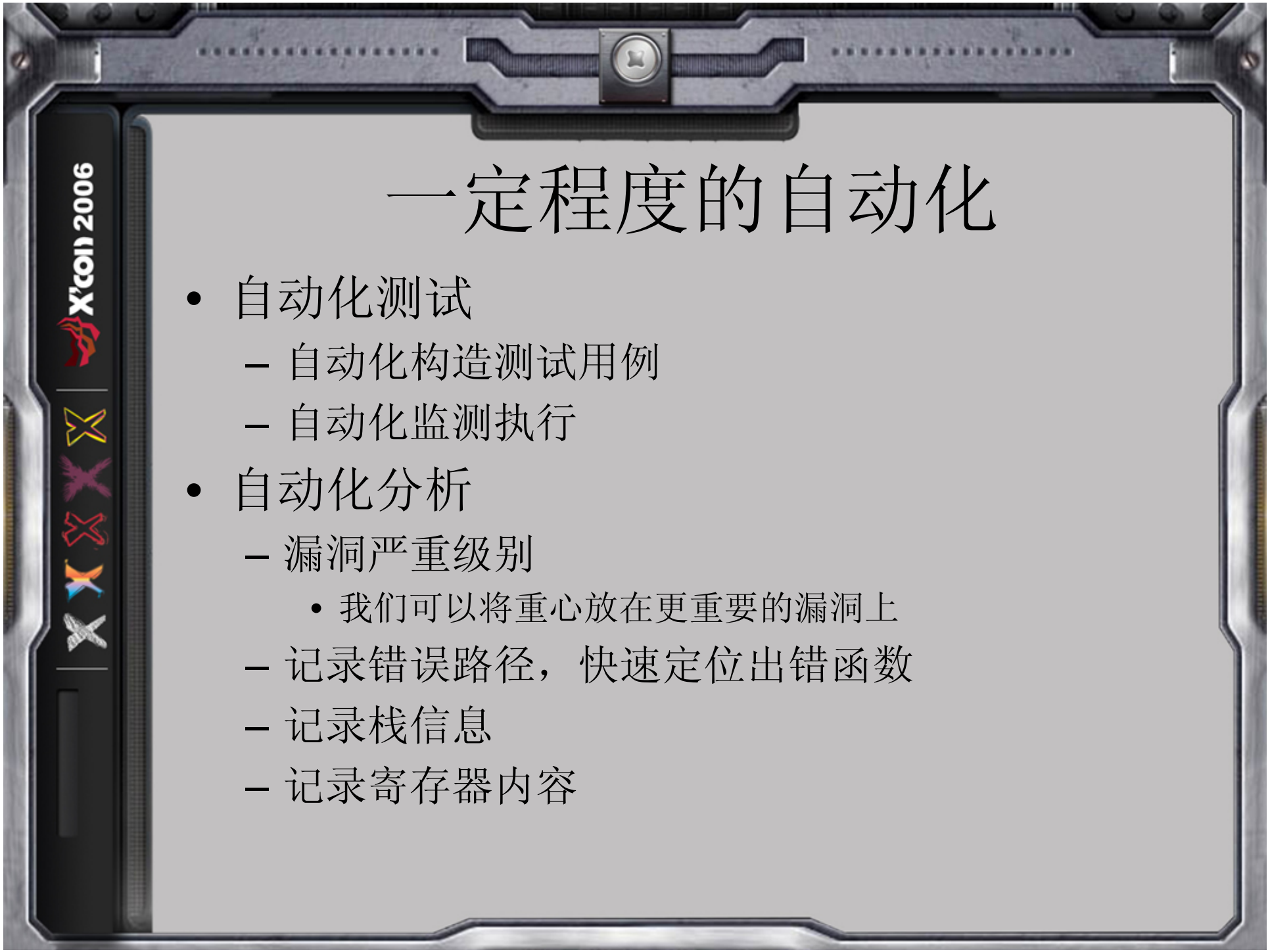
静态分析算法的考虑

- 追求的目标
  - 可伸缩性
  - 正确性
  - 精确性
- 算法
  - 流相关--依赖于控制流
  - 上下文相关



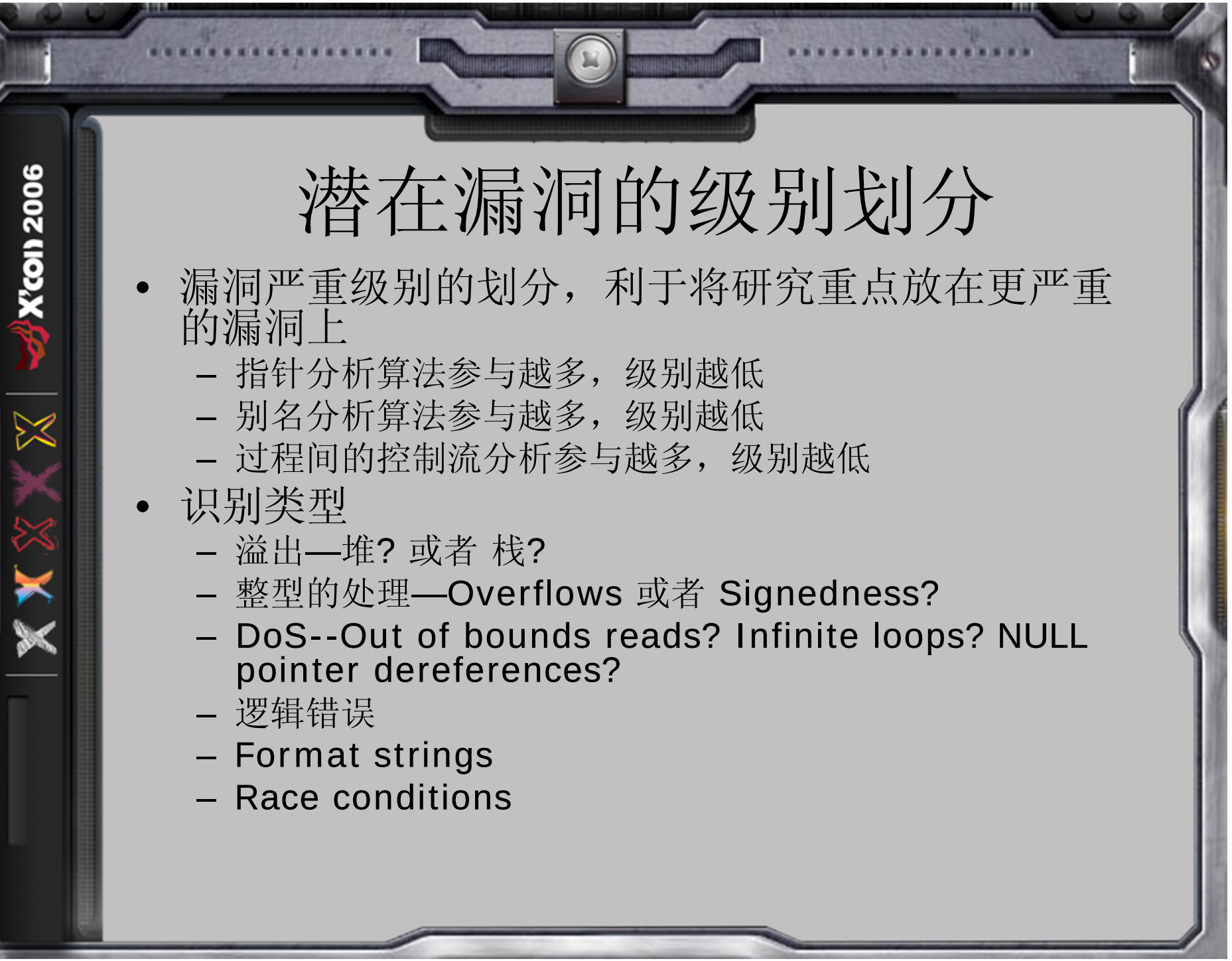
漏洞挖掘的现状

- 特点：
  - 一定程度的自动化
  - 线性程序分析
- 缺点：
  - 误报
  - 没有良好的图形化显示
  - 很少研究多线程程序的安全性



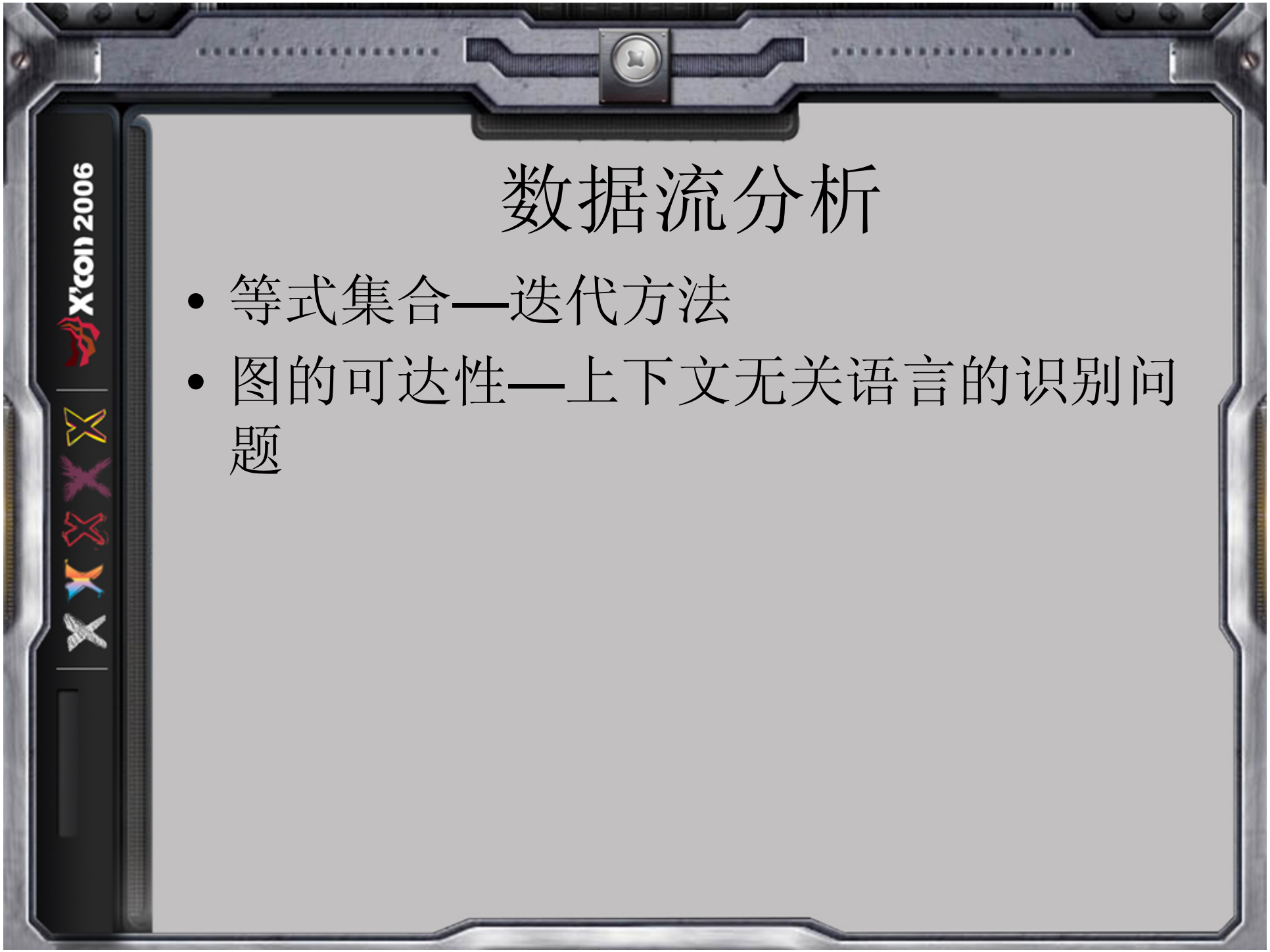
一定程度的自动化

- 自动化测试
  - 自动化构造测试用例
  - 自动化监测执行
- 自动化分析
  - 漏洞严重级别
    - 我们可以将重心放在更重要的漏洞上
  - 记录错误路径，快速定位出错函数
  - 记录栈信息
  - 记录寄存器内容



潜在漏洞的级别划分

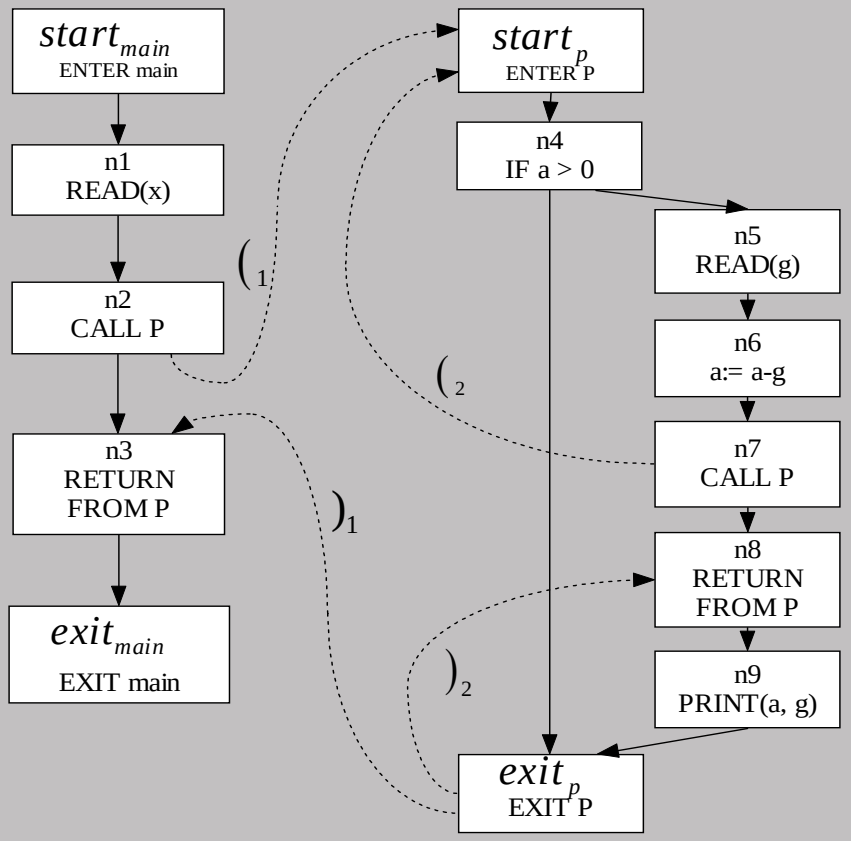
- 漏洞严重级别的划分，利于将研究重点放在更严重的漏洞上
  - 指针分析算法参与越多，级别越低
  - 别名分析算法参与越多，级别越低
  - 过程间的控制流分析参与越多，级别越低
- 识别类型
  - 溢出—堆? 或者 栈?
  - 整型的处理—Overflows 或者 Signedness?
  - DoS--Out of bounds reads? Infinite loops? NULL pointer dereferences?
  - 逻辑错误
  - Format strings
  - Race conditions



数据流分析

- 等式集合—迭代方法
- 图的可达性—上下文无关语言的识别问题

图的可达性



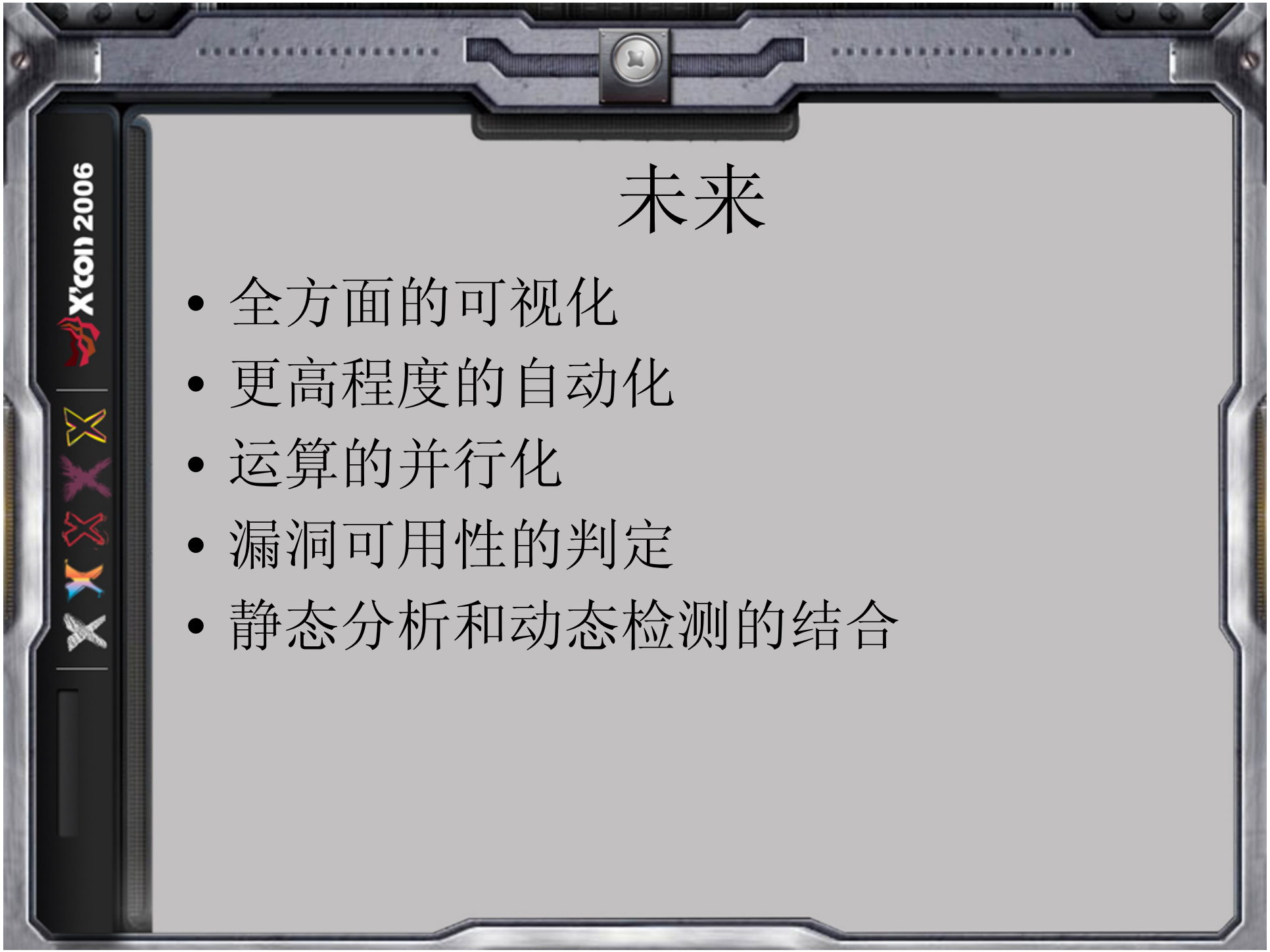

```
Declare g:int

Procedure main
Begin
    declare x:int
    read(x)
    call P(x)
End

Procedure P(value a:int)
Begin
    if (a > 0) then
        read (g)
        a:= a-g
        call P(a)
        print(a, g)
    fi
end
```

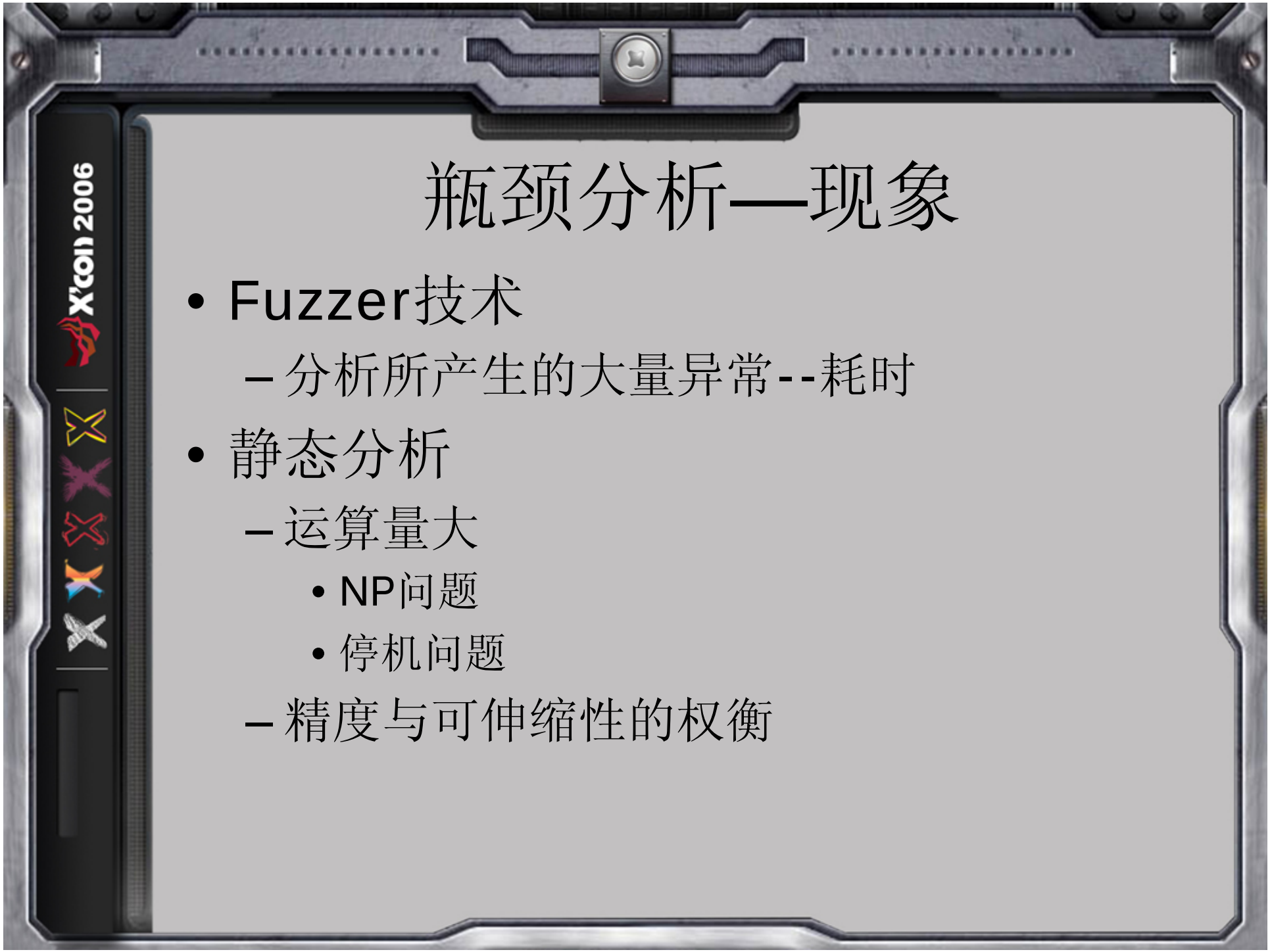
CFL的可达性问题

- 路径 “$start_{main}$->n1->n2->$start_p$->n4->$exit_p$->n3”—”$ee(_1ee)_1$”
  - 匹配和可行的
- 路径 “$start_{main}$->n1->n2->$start_p$->n4”—”$ee(_1e$”
  - 可行的但是不匹配的
- 路径 “$start_{main}$->n1->n2->$start_p$->n4->exitp->n8”—”$ee(_1ee)_2$”
  - 既不是匹配的也不是可行的



未来

- 全方面的可视化
- 更高程度的自动化
- 运算的并行化
- 漏洞可用性的判定
- 静态分析和动态检测的结合



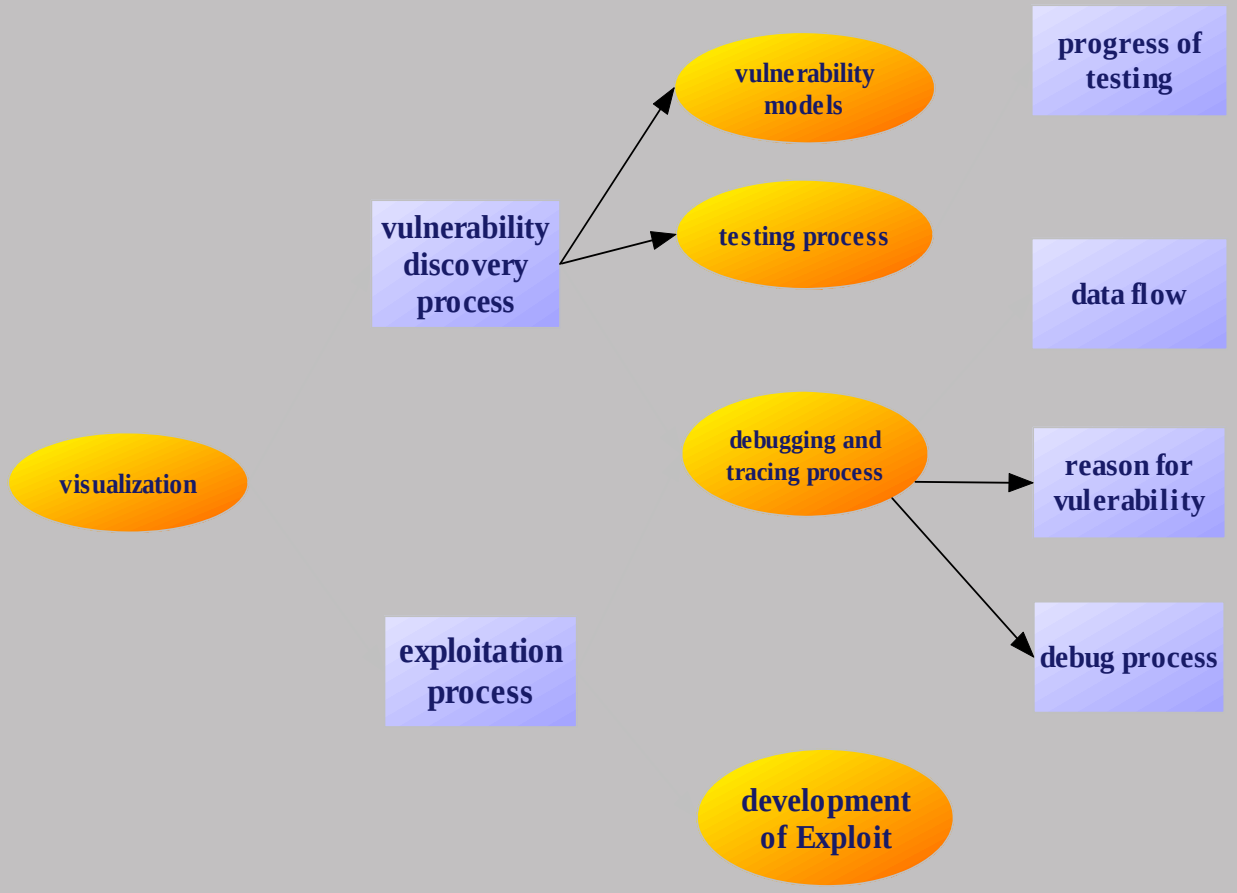
瓶颈分析—现象

- Fuzzer技术
  - 分析所产生的大量异常--耗时
- 静态分析
  - 运算量大
    - NP问题
    - 停机问题
  - 精度与可伸缩性的权衡

瓶颈分析—原因

- 异常情况分析
  - 可用性判定需要人的鉴别
    - 需要更好的图形化
    - 需要自动分析
- 静态分析
  - 需要更高的精度

可视化


vulnerability models
progress of testing
vulnerability discovery process
testing process
data flow
visualization
debugging and tracing process
reason for vulerability
debug process
exploitation process
development of Exploit

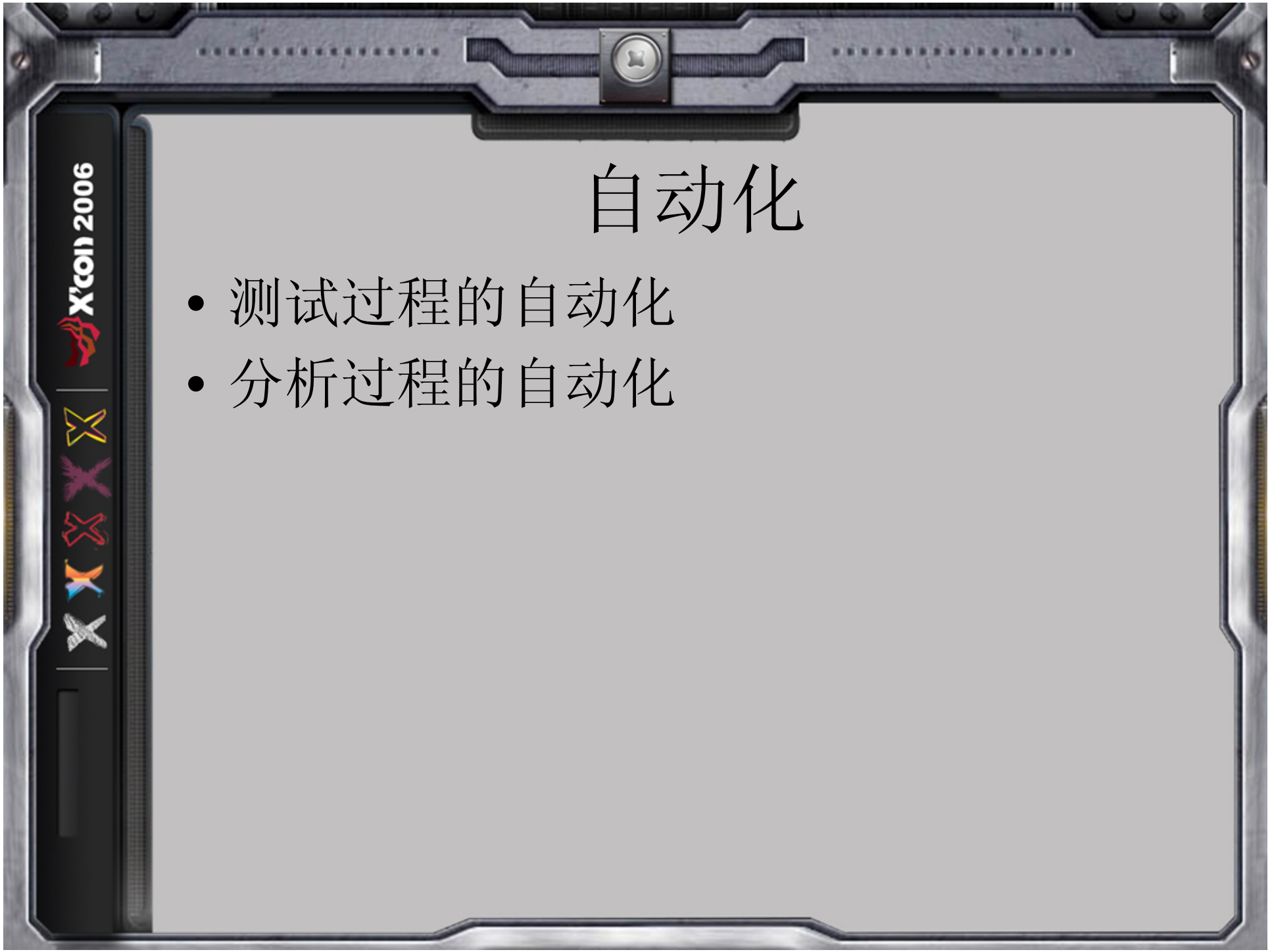
并行化

- J. Wilander 的实验

  采用基于依赖图的方法。

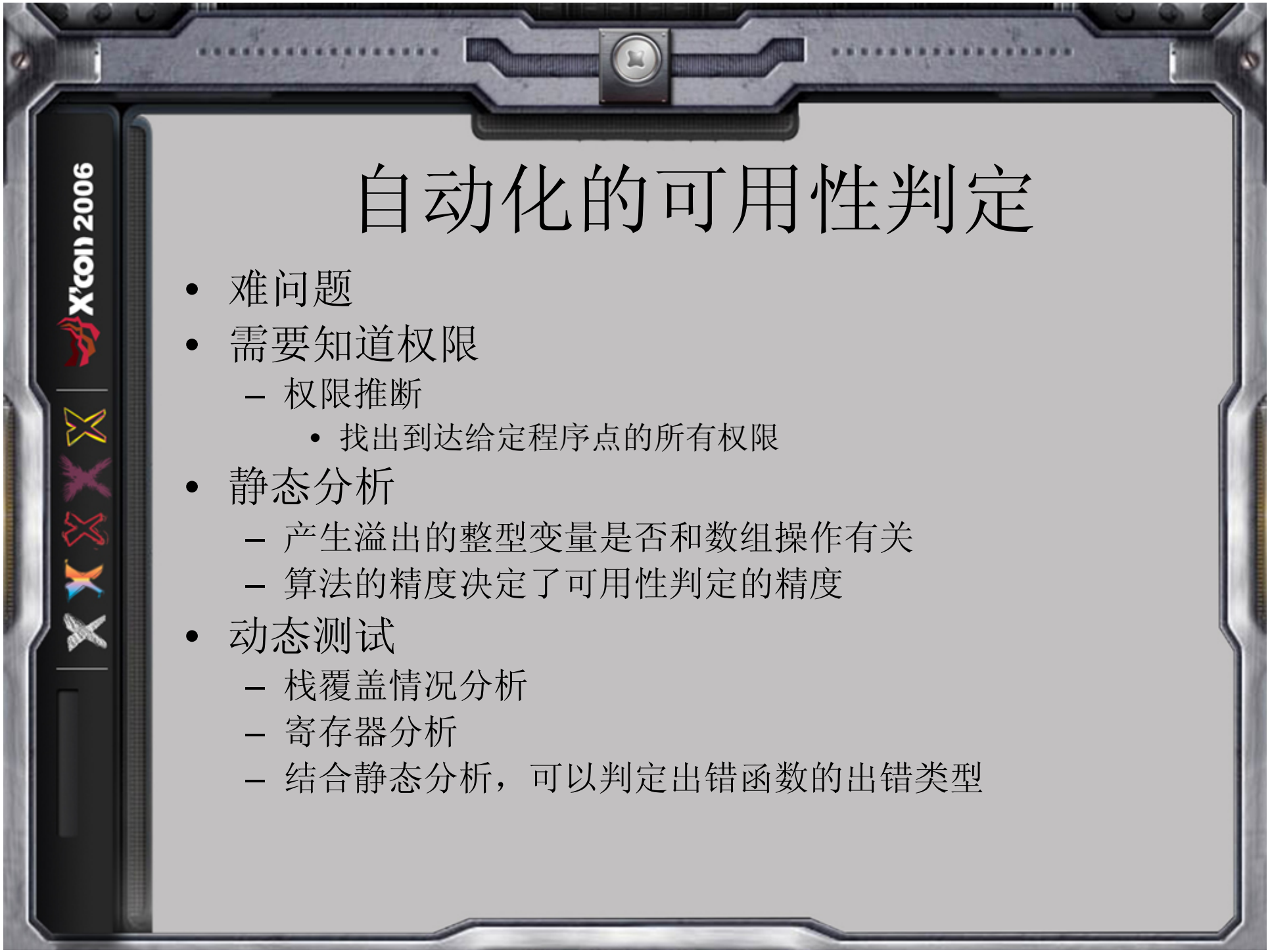
  - Wu-ftpd 2.6-4包含将近20,000行代码，产生130,000个顶点。
  - 在P4 2.66的机器上需要运行15个多小时。
- Balakrishnan的实验

Program	Procedures	Instructions	Malloc sites	Indirect jumps	Calls	Indirect calls	Memory usage (MB)	Value-set analysis (sec.)	Affine-relation analysis (sec.)
javac	36	3555	1	0	133	79	51	76	29
cat (2.0.14)	123	3892	1	3	138	4	42	9	26
cut (2.0.14)	129	4329	2	3	182	4	48	7	42
grep (2.4.2)	245	16808	18	4	654	6	102	117	75
gcc (2.96)	252	22984	8	3	1048	4	232	108	295
tar (1.13.19)	581	47739	11	21	2553	29	258	220	156
awk (3.1.0)	595	69927	84	33	3669	152	623	1017	1011
winhlp32 (5.0.2195.2014)	1018	108380	0	10	6002	1005	737	1712	1290



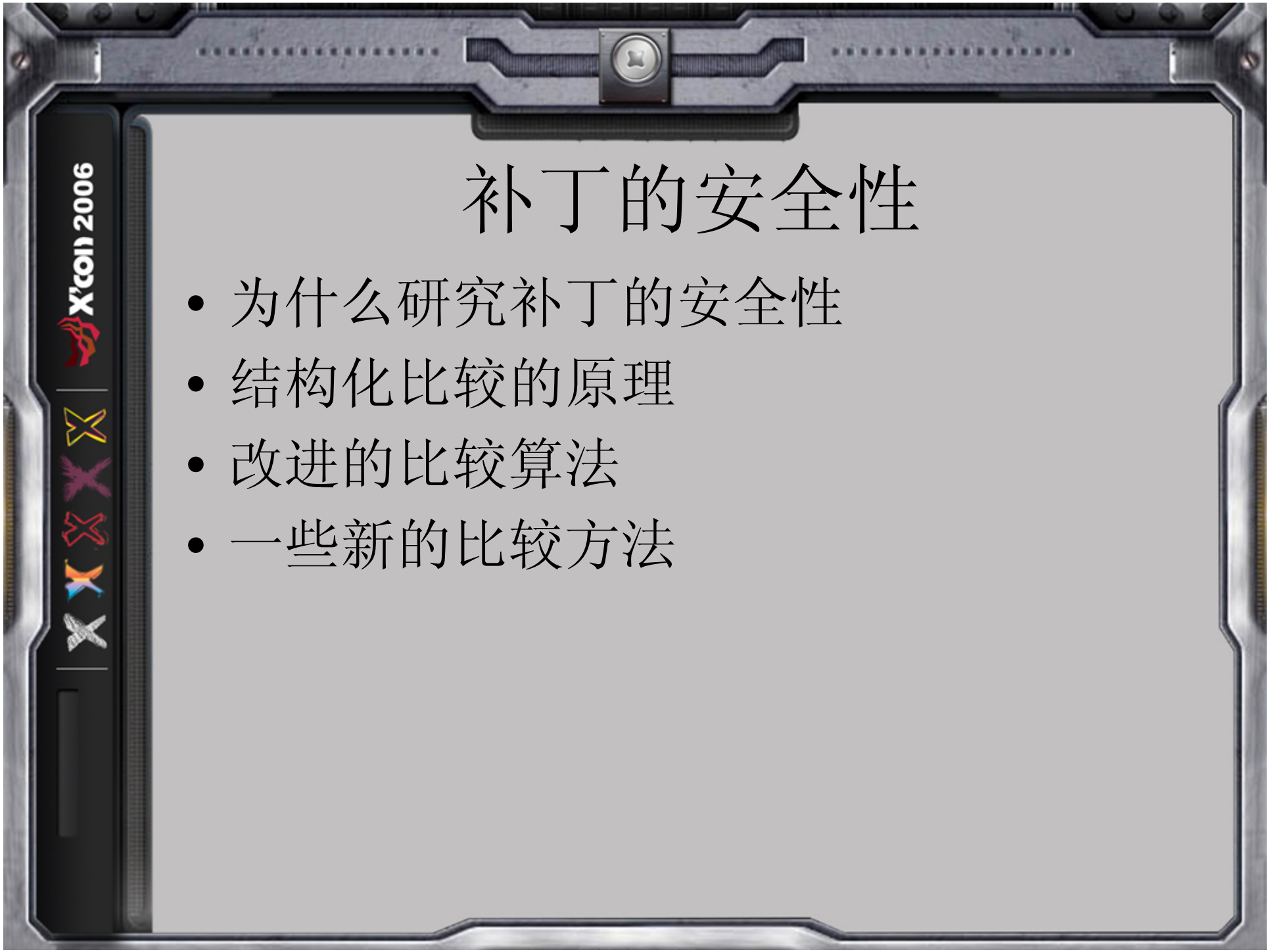
自动化

- 测试过程的自动化
- 分析过程的自动化



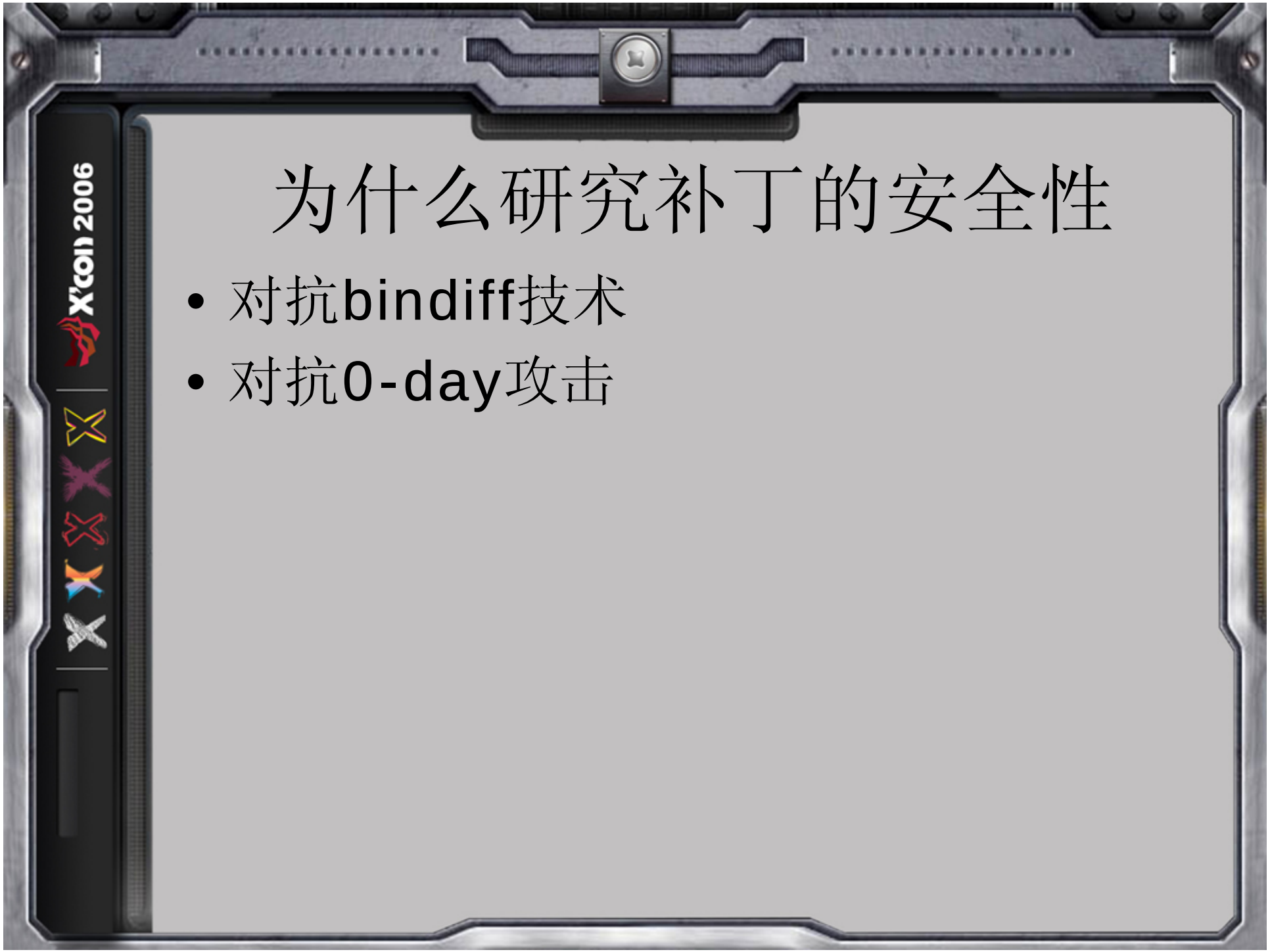
自动化的可用性判定

- 难问题
- 需要知道权限
  - 权限推断
    - 找出到达给定程序点的所有权限
- 静态分析
  - 产生溢出的整型变量是否和数组操作有关
  - 算法的精度决定了可用性判定的精度
- 动态测试
  - 栈覆盖情况分析
  - 寄存器分析
  - 结合静态分析，可以判定出错函数的出错类型



补丁的安全性

- 为什么研究补丁的安全性
- 结构化比较的原理
- 改进的比较算法
- 一些新的比较方法



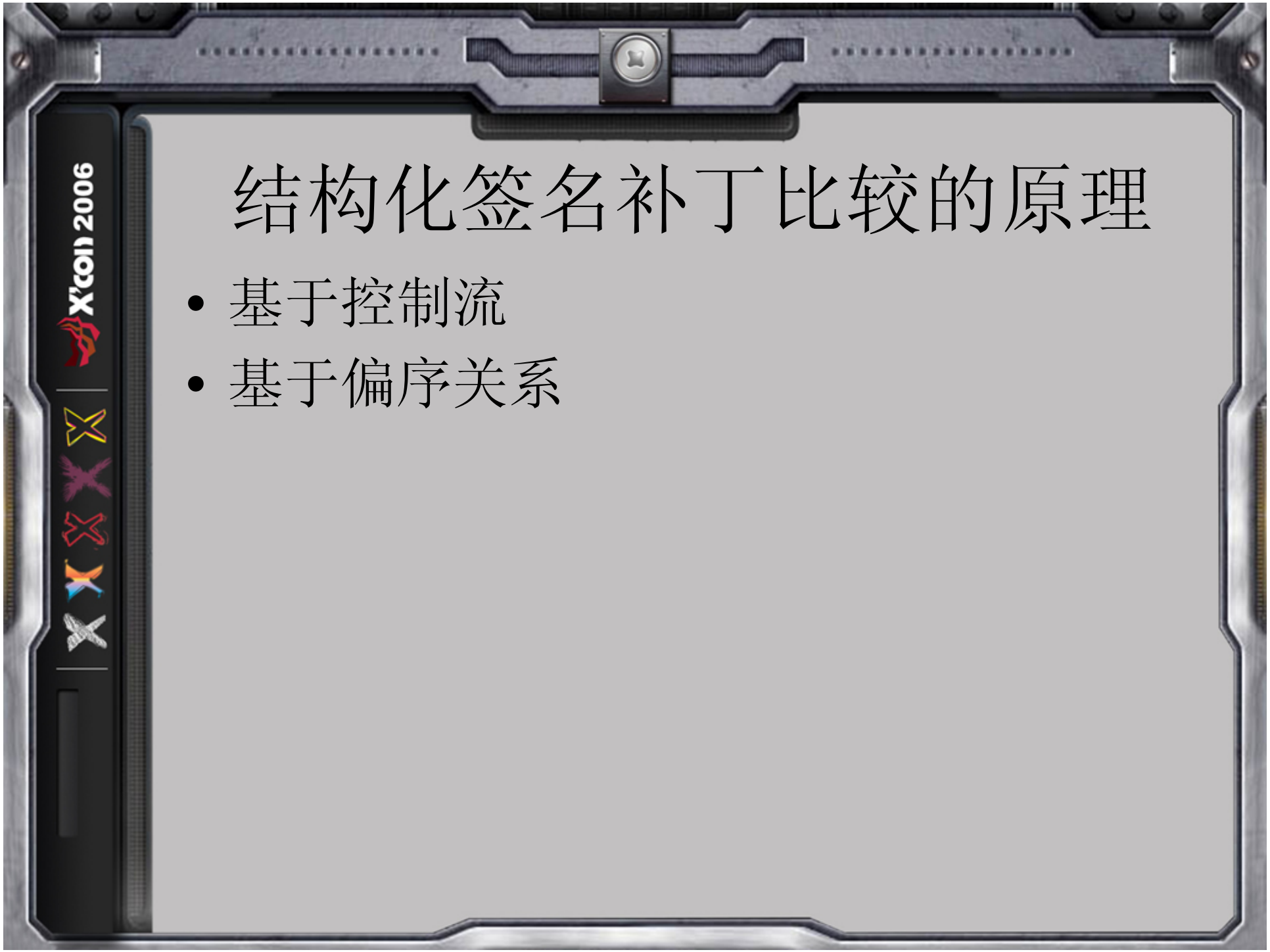
为什么研究补丁的安全性

- 对抗bindiff技术
- 对抗0-day攻击

补丁	相同名字	唯一性签名	相同字符串引用	相同素数乘积	相同入度	递归	启发式匹配
06-040	686	243	72	13	34	0	
		66	6	14	4	0	34
06-020	2529	12	4	6	0	0	
		68					2
05-047	237	3		2			
		4			1		2
06-014	873	14		4	1		
		1					0
06-002	99	38		9	6		
		18	2		6		5

补丁分析的例子(FunnyDiff)

补丁	文件名	版本	大小	函数个数	匹配个数	发生改变	运行时间(ms)
MS06-040	Netapi32.dll	5.0.2195.7105	309,520	1,172	1,168	7	7,153
	Netapi32.dll	5.0.2195.7038	326,928	1,168	1,168		
MS06-020	Msdtcprx.dll	2001.12.4414.311	426,496	2,623	2,621	4	10,828
	Msdtcprx.dll	2001.12.4414.308	425,472	2,621	2,621		
MS05-047	Umpnpmgr.dll	5.1.2600.2744	121,344	268	249	19	1,062
	Umpnpmgr.dll	5.1.2600.2710	116,224	249	249		
MS06-014	Msadco.dll	2.81.1124.0	143,360	896	894	3	2,594
	Msadco.dll	2.81.1117.0	143,360	894	894		
MS06-002	Fontsub.dll	5.0.2195.7071	79,632	204	183	35	968
	Fontsub.dll	5.0.2180.1	78,096	183	183		
M05-039	Umpnpmgr.dll	5.0.2600.2710	116,224	249	249	1	1,000
	Umpnpmgr.dll	5.0.2600.2180	116,224	249	249		

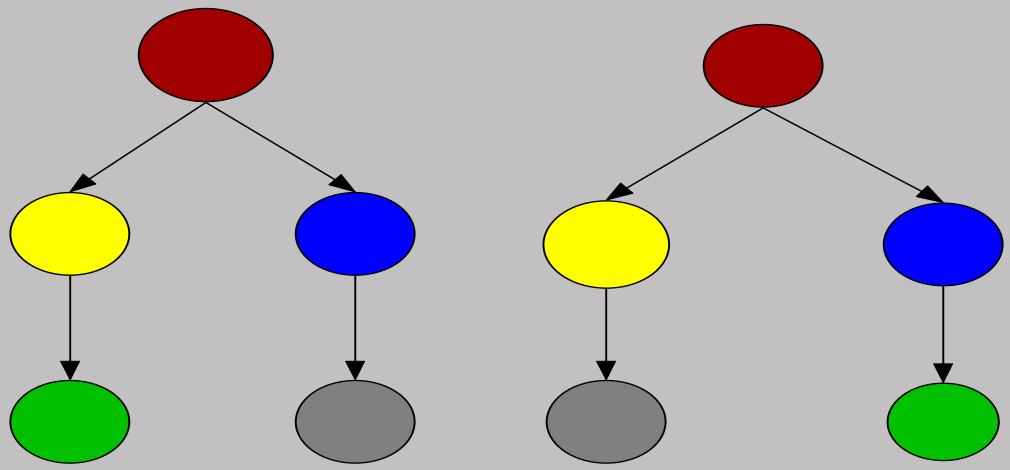


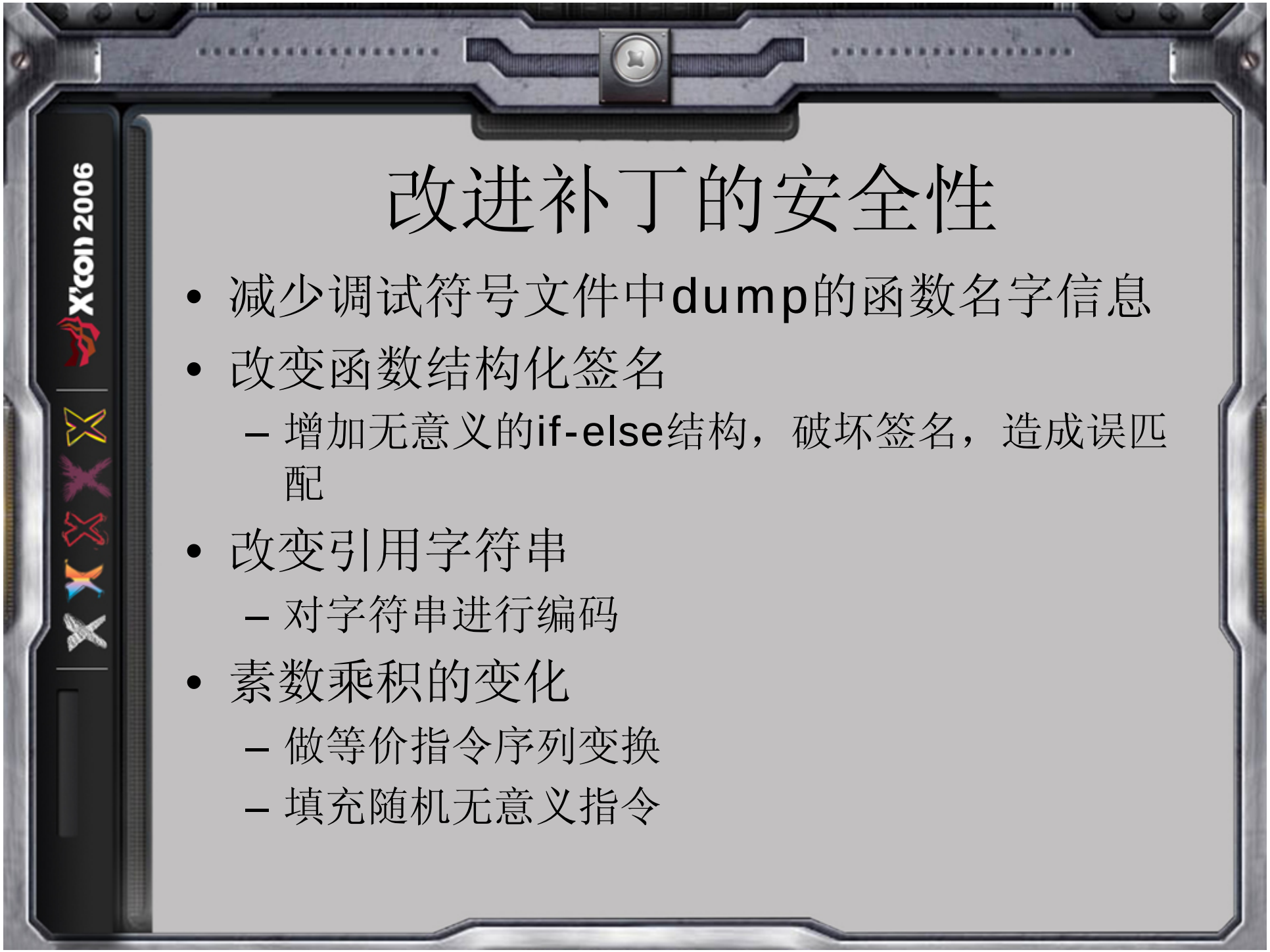
结构化签名补丁比较的原理

- 基于控制流
- 基于偏序关系

改进的比较算法

- 初始化基点的可信
  - 基于模糊聚类
- 基点传播过程的启发式校验
  - 减少误匹配



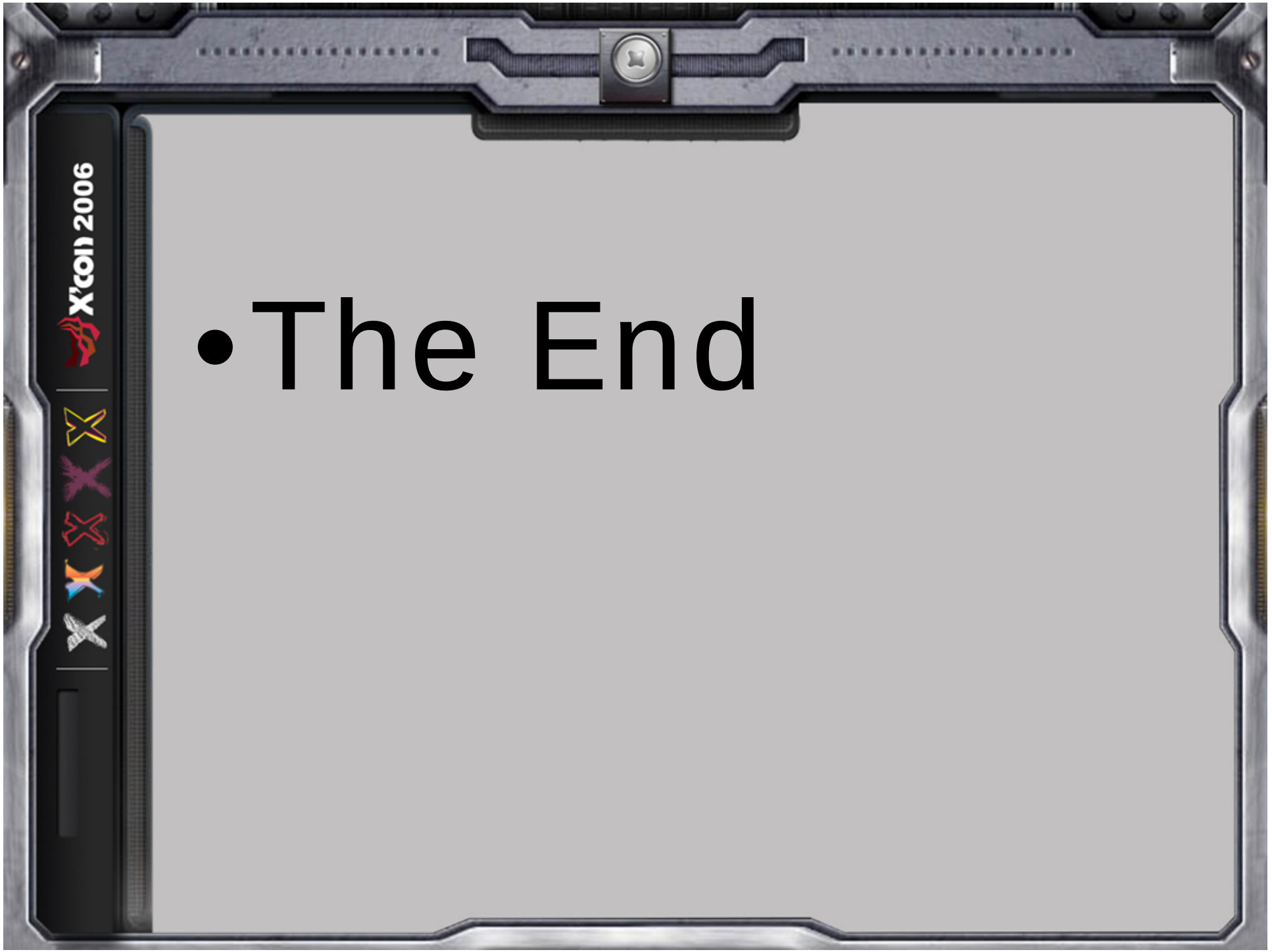


改进补丁的安全性

- 减少调试符号文件中dump的函数名字信息
- 改变函数结构化签名
  - 增加无意义的if-else结构，破坏签名，造成误匹配
- 改变引用字符串
  - 对字符串进行编码
- 素数乘积的变化
  - 做等价指令序列变换
  - 填充随机无意义指令

一些新的比较方法

- Import/export表函数作为固定点
- 使用相同变量
- RPC接口
- 相同栈布局
- 相同参数
- 具有循环属性



- The End