软件安全与漏洞分析

3.6 关于软件与操作系统安全的其他技术

Previously in Software Security

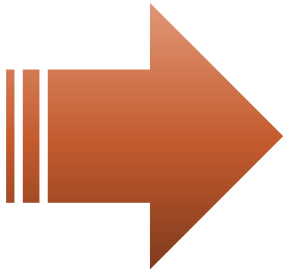
- 地址空间随机化及其破解
  - 粗/细粒度的ASLR方法
  - ASLR的无效化：Just-In-Time代码重用

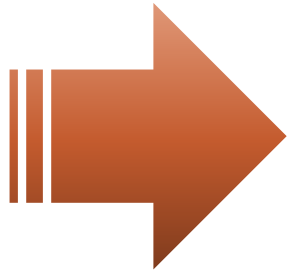
关于软件与操作系统安全的其他技术

- 本节主题1 – 其他软件与操作系统防护技术
  - 安全数据删除
  - 针对不可信操作系统的防护
- 本节主题2 – 软件验证和漏洞检测技术
  - Fuzzing
  - （静态/动态）符号执行
  - 污点传播

安全数据删除

- 对手：能够**以某种方式访问数据存储介质**的攻击者
- 安全场景：在目标数据删除后，要求攻击者无法恢复被删除的内容
- 实际场景举例：**硬件的再利用**



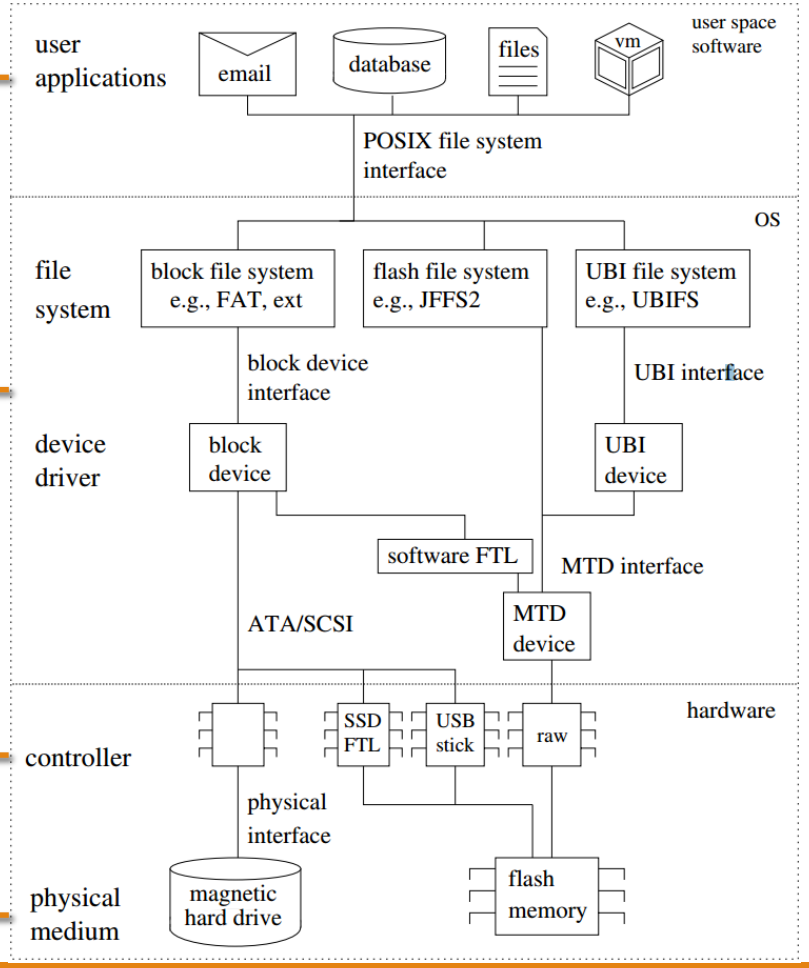

安全数据删除

- 文件重写
- 空闲存储空间填充
- 加密+密钥破坏
- 安全数据库删除

- 文件系统级数据重写
- 数据精简(compaction)
- 文件头部精简
- 内存洗涤(scrubbing)

逐单元地数据重写
遍历数据对象所占据空间

物理销毁



安全数据删除

- 根据**被攻击面（attack surface）**的不同：
  - robust-keyboard attacks（软攻击，攻击者**以驱动程序**的形式访问存储介质）
  - laboratory attacks（硬攻击，攻击者直接通过物理手段访问存储介质）
- 攻击者的访问时机和次数：
  - 访问时机**由用户控制** -- 如前述的硬件再利用场景
  - 访问时机**由攻击者控制** -- 如robust-keyboard attacks
  - 通常考虑攻击者在**删除发生后访问存储**介质
  - 也需要考虑攻击者在**数据写入前**和**删除发生后**分别访问存储介质

安全数据删除

- 针对“加密后销毁密钥”的数据删除时：
  - **非压迫性攻击者** -- 未获得用户口令或其他用于保护数据的凭据（credentials）
  - **高压迫性攻击者** -- 获得了上述全部或部分凭据、能够解除数据保护手段
  - **弱口令攻击者** -- 能够猜测用户口令，但无法获得签名、密钥等强度更高的凭证
- 此外，需要考虑攻击者是否收到计算能力的约束

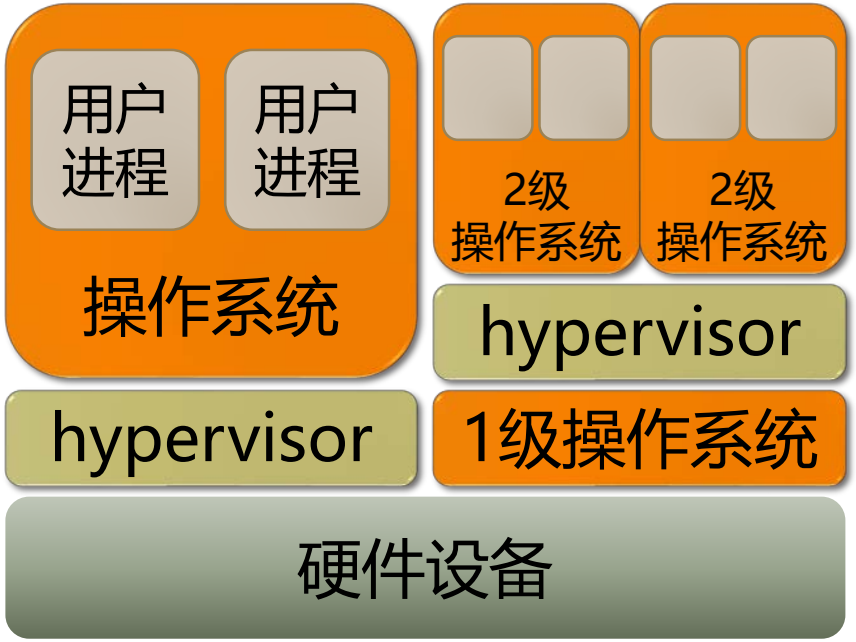
安全数据删除

Adversary's Name	Disclosure	Credentials	Bound	Accesses	Surface	Example
internal repurposing	user	non-coercive	bounded	sing/mult	controller	loan your account
external repurposing	user	non-coercive	bounded	single	physical	sell old hardware
advanced forensic	user	non-coercive	unbounded	single	physical	unfathomable forensic power
border crossing	user	coercive	bounded	sing/mult	physical	perjury to not reveal password
unbounded border crossing	user	coercive	unbounded	sing/mult	physical	corporate policy on encrypted data
malware	adversary	non-coercive	bounded	sing/mult	user-level	malicious application
compromised OS	adversary	non-coercive	bounded	sing/mult	block device	malware in the operating system
bounded coercive	adversary	coercive	bounded	single	physical	legal subpoena
unbounded coercive	adversary	coercive	unbounded	single	physical	legal subpoena and broken crypto
bounded peek-a-boo	adversary	coercive	bounded	multiple	physical	legal subpoena with earlier spying
unbounded peek-a-boo	adversary	coercive	unbounded	multiple	physical	legal subpoena, spying, broken crypto

- 参考文献：Reardon J, Basin D, Capkun S. Sok: Secure data deletion[C]//Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013: 301-315.

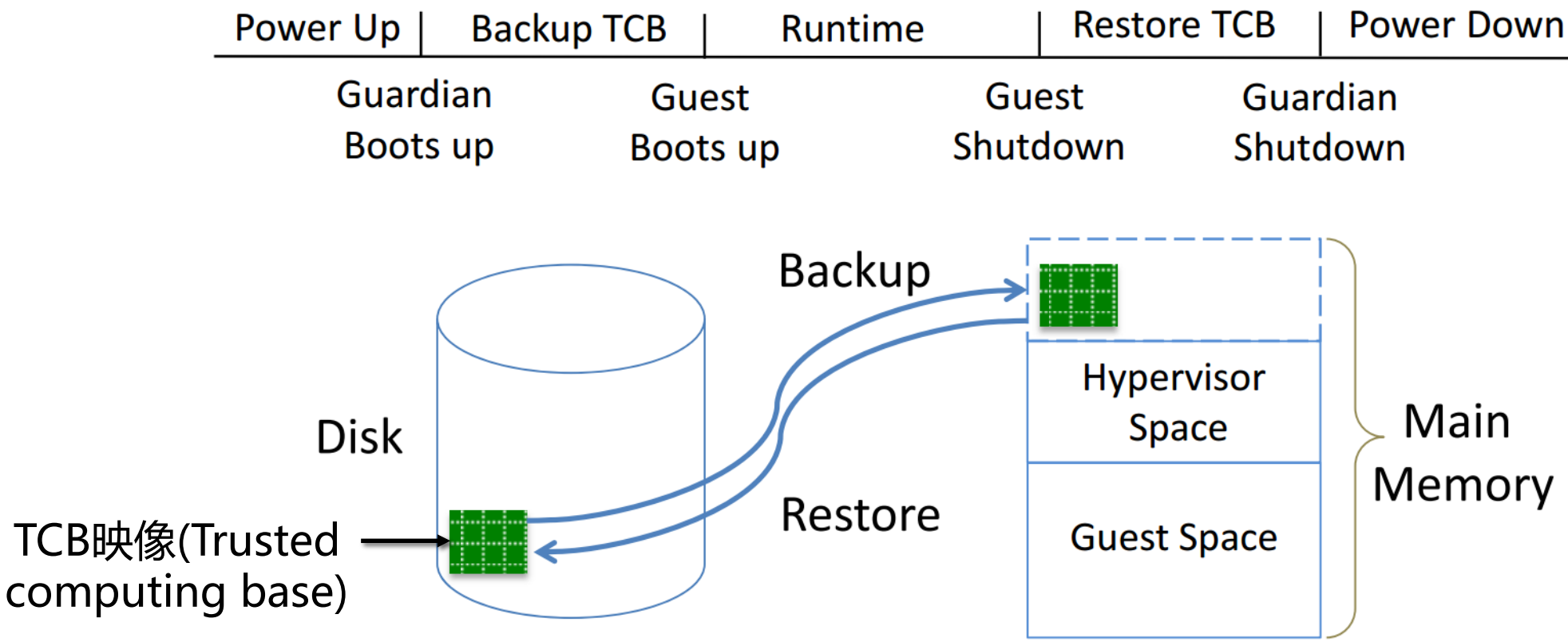
针对不可信操作系统的防护

- 软件安全的常规安全模型：
  - 用户软件存在漏洞
  - 攻击者通过进程的非内核空间实施攻击
- 如果攻击者将恶意软件植入操作系统内核呢？
  - **软件保护还需要针对不可信的操作系统**
  - 手段：**超级监督者（hypervisor）**



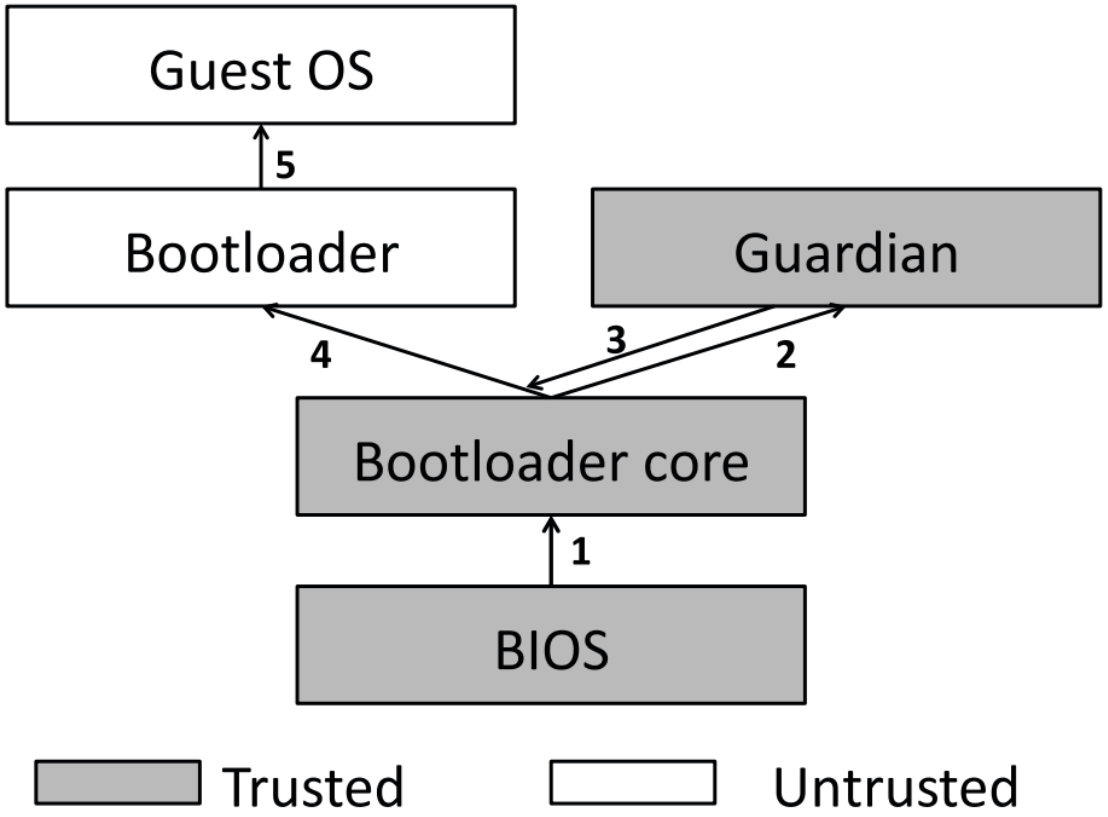
针对不可信操作系统的防护

- 应用1：作为计算机系统的安全根



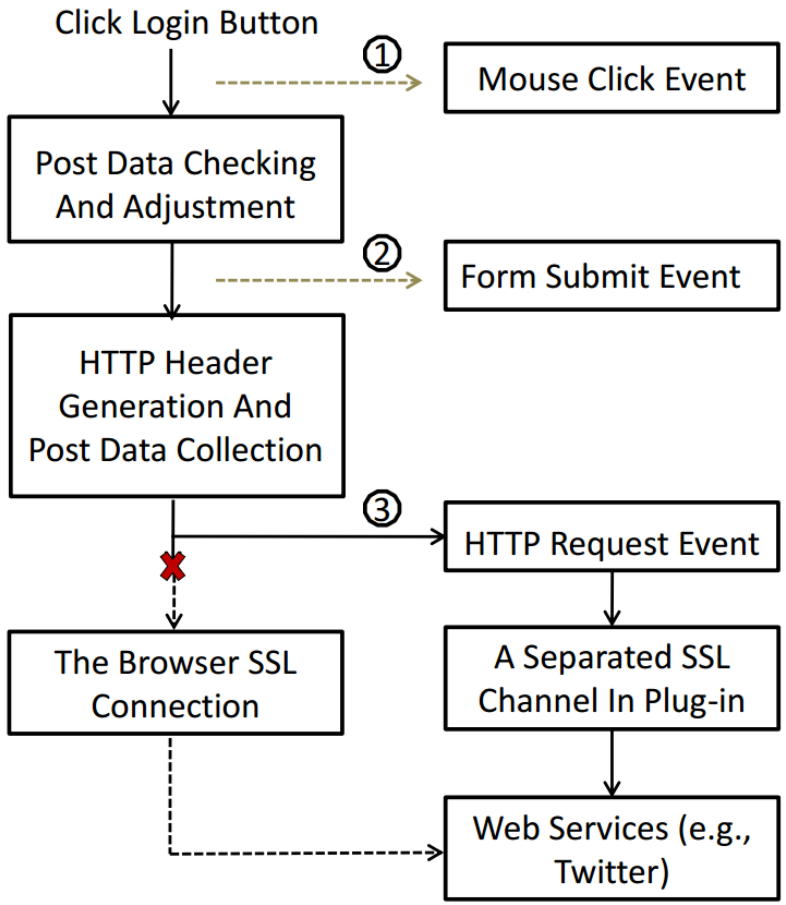
针对不可信操作系统的防护

- 应用1：作为计算机系统的安全根



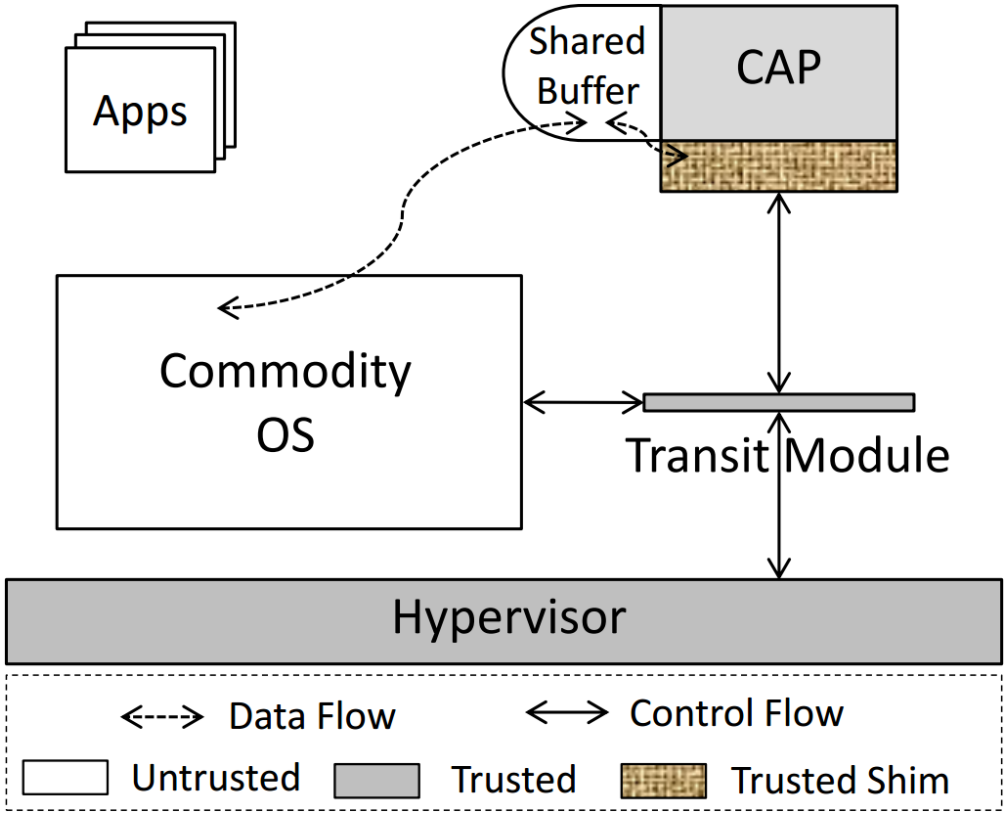
针对不可信操作系统的防护

- 应用2：用户口令保护



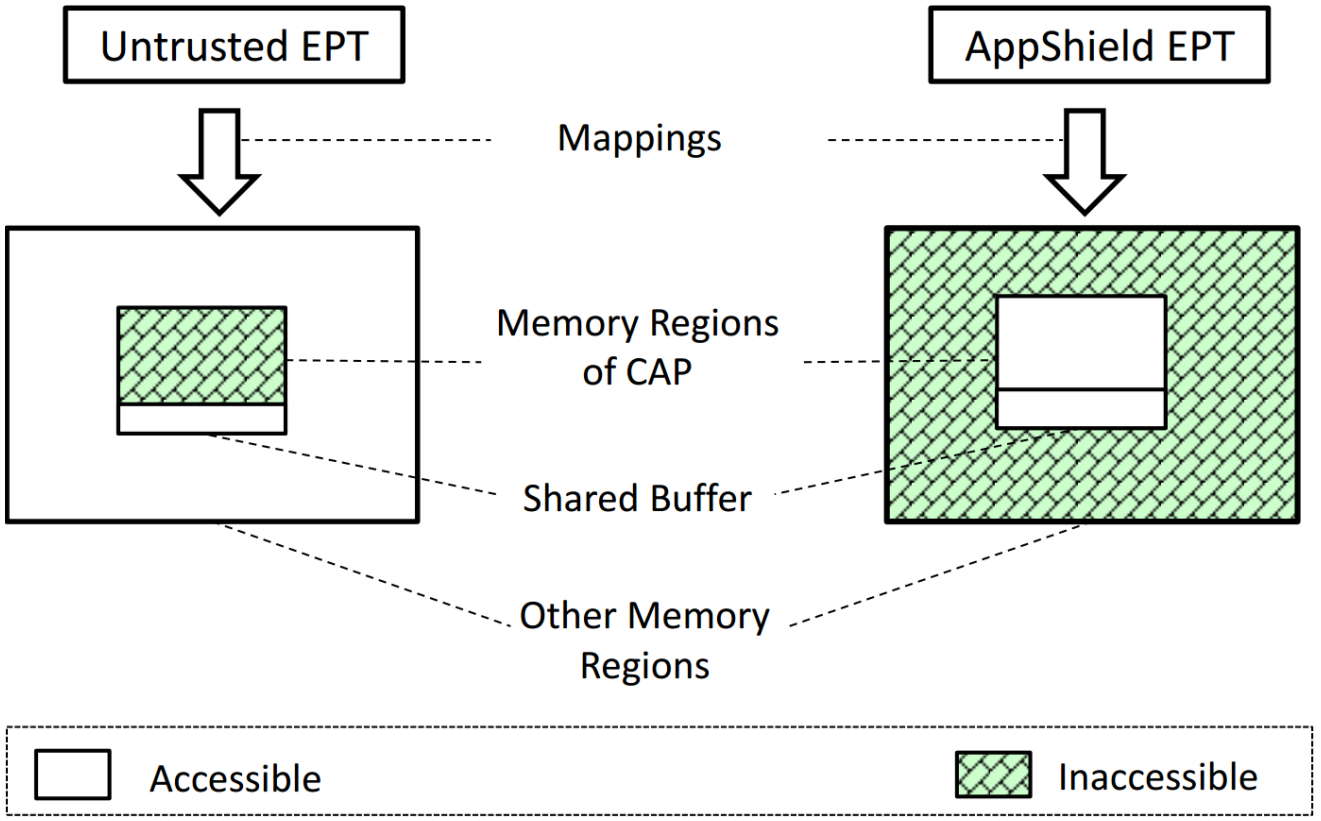
针对不可信操作系统的防护

- 应用3：应用空间的动态隔离与应用执行流的保护



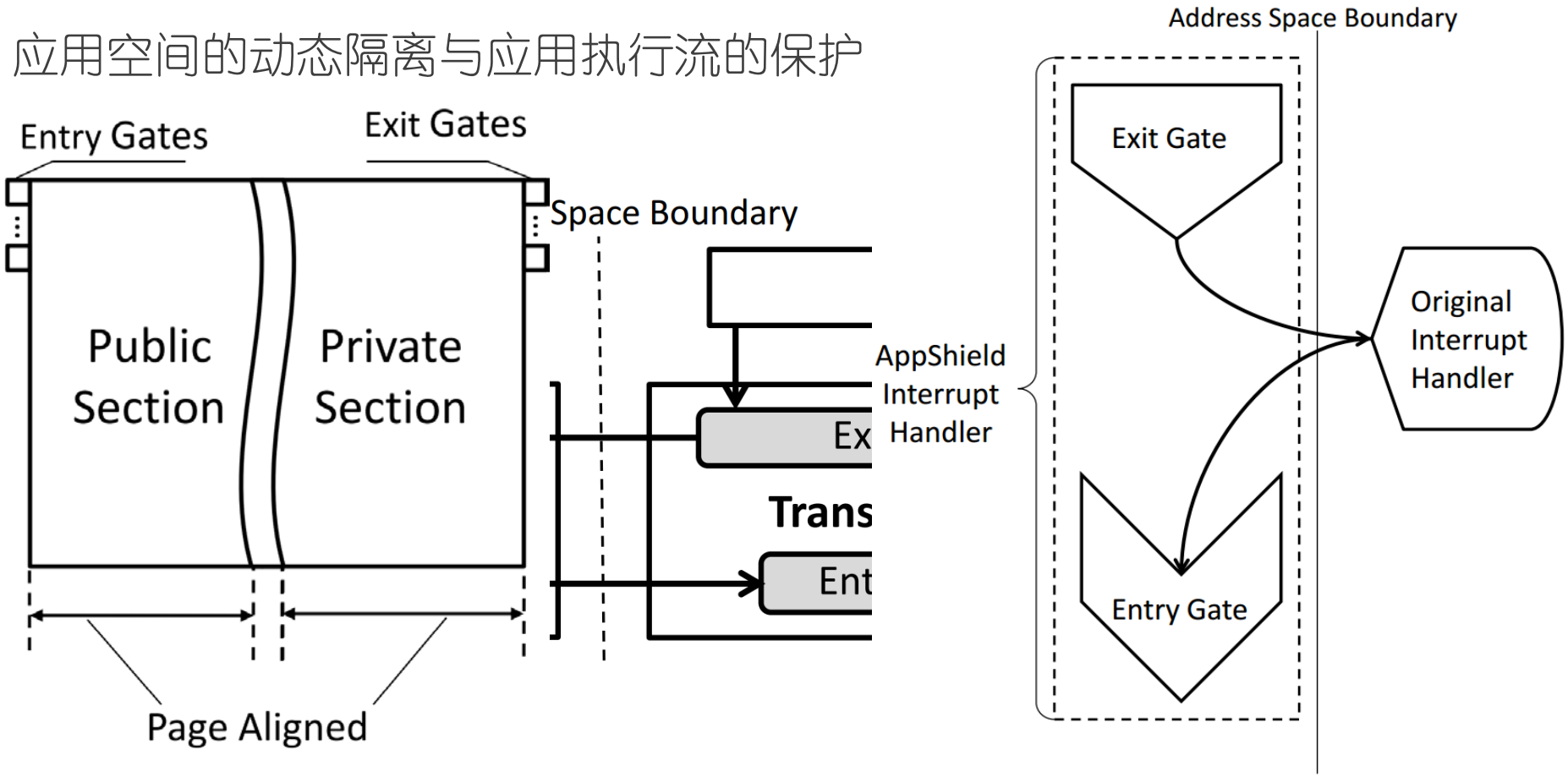
针对不可信操作系统的防护

- 应用3：应用空间的动态隔离与应用执行流的保护



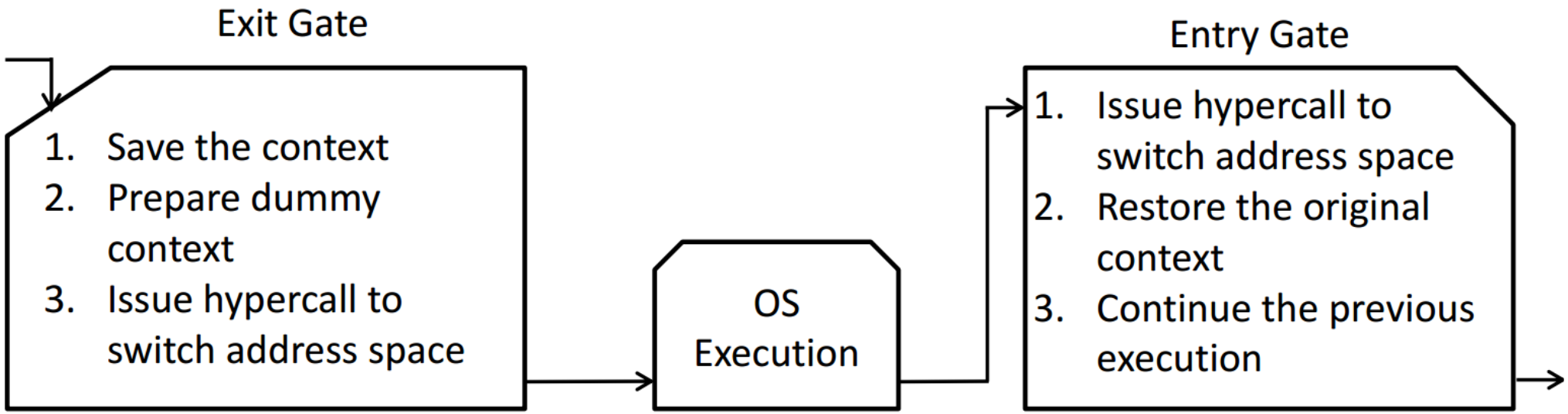
针对不可信操作系统的防护

- 应用3：应用空间的动态隔离与应用执行流的保护



针对不可信操作系统的防护

- 应用3：应用空间的动态隔离与应用执行流的保护

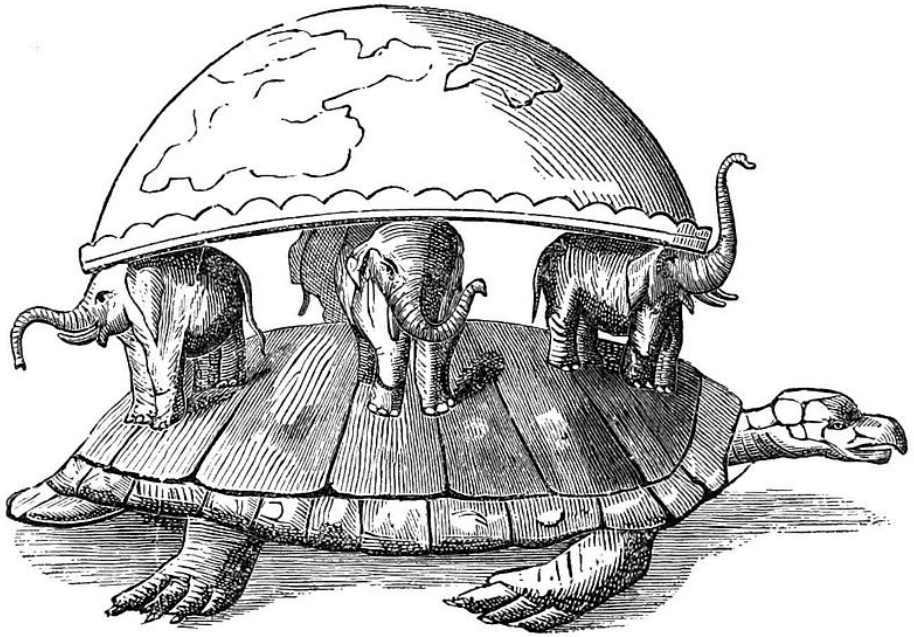


针对不可信操作系统的防护

- 参考文献：
  - Cheng Y, Ding X. Virtualization based password protection against malware in untrusted operating systems[C]//International Conference on Trust and Trustworthy Computing. Springer Berlin Heidelberg, 2012: 201-218.
  - Cheng Y, Ding X. Guardian: Hypervisor as security foothold for personal computers[C]//International Conference on Trust and Trustworthy Computing. Springer Berlin Heidelberg, 2013: 19-36.
  - Cheng Y, Ding X, Deng R. Appshield: Protecting applications against untrusted operating system[J]. Singaport Management University Technical Report, SMU-SIS-13, 2013, 101.

针对不可信操作系统的防护

- 思考问题：海龟下面还是海龟（**turtles all the way down**）



那么，这海龟又是站在什么上面……

软件验证和漏洞检测技术

- 软件验证：证明软件执行逻辑是否吻合预期，指出逻辑错误的位置和影响

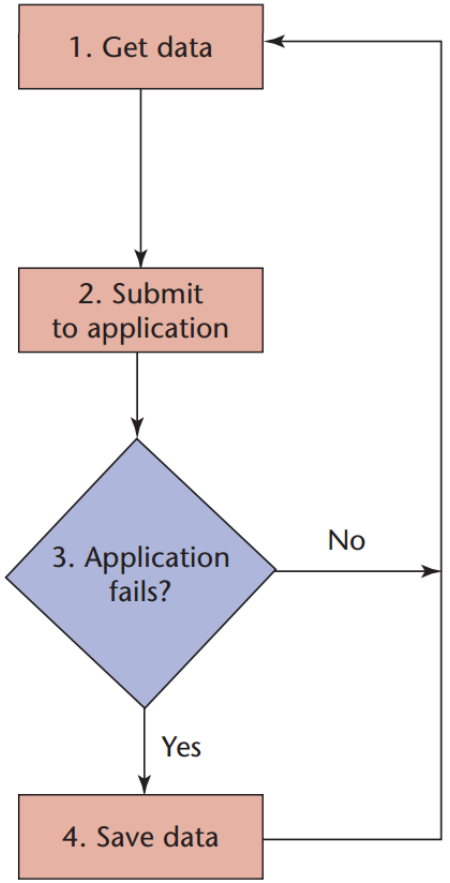
然而事情往往是：
走着走着就瘸了……

软件验证和漏洞检测技术

- 软件验证的主要手段：
  - Fuzzing
  - （静态/动态）符号执行
  - 污点传播

软件验证和漏洞检测技术

- Fuzzing：
  - 通过注入错误，寻找软件中可能的崩溃点
  - 设定程序输入，跟踪并收集执行状态、动态数据等信息
  - 若程序正常，根据所收集的信息推断并生成一个新的输入值
  - 重复测试直到程序崩溃
- Fuzzing存在的不足：
  - 代码覆盖率差
  - 无法有效地产生出具有代表性的测试输入值



软件验证和漏洞检测技术

- 静态符号执行：
  - 程序即函数
  - 函数可以表示为代数逻辑
  - 因此 -- 程序也可以表示为代数符号之间的逻辑

```
1  SUM: PROCEDURE (A,B,C);
2          X ← A + B;
3          Y ← B + C;
4          Z ← X + Y − B;
5          RETURN (Z);
6  END;
```

After statement	X	Y	Z	A	B	C	*pc*
			Values would be				
1	?	?	?	α_1	α_2	α_3	*true*
2	$\alpha_1+\alpha_2$	—	—	—	—	—	—
3	—	$\alpha_2+\alpha_3$	—	—	—	—	—
4	—	—	$\alpha_1+\alpha_2+\alpha_3$	—	—	—	—
5			(Returns $\alpha_1+\alpha_2+\alpha_3$)				

软件验证和漏洞检测技术

- 静态符号执行：

```
1  POWER: PROCEDURE(X, Y);
2             Z ← 1;
3             J ← 1;
4  LAB:       IF Y ≥ J THEN
5             DO; Z ← Z * X;
6                 J ← J + 1;
7                 GO TO LAB; END;
8             RETURN (Z);
9  END;
```

After statement	J	X	Y	Z	*pc*
	Values would be				
1	?	α_1	α_2	?	*true*
2	—	—	—	1	—
3	1	—	—	—	—
4 execution in detail:					
(a) evaluate Y≥J getting $\alpha_2 \geq 1$.					
(b) use path condition and check:					
(i) *true* ⊃ $\alpha_2 \geq 1$					
(ii) *true* ⊃ ¬($\alpha_2 \geq 1$					
(c) neither are theorems, so fork.					
Case ¬($\alpha_2 \geq 1$):					
4	1	α_1	α_2	1	¬($\alpha_2 \geq 1$)
8 this case completed. (returns 1 when $\alpha_2 < 1$.)					
Case $\alpha_2 \geq 1$:					
4	1	α_1	α_2	1	$\alpha_2 \geq 1$
5	—	—	—	α_1	—
6	2	—	—	—	—
7	—	—	—	—	—
4 execution in detail:					
(a) evaluate Y≥J, getting $\alpha_2 \geq 2$					

软件验证和漏洞检测技术

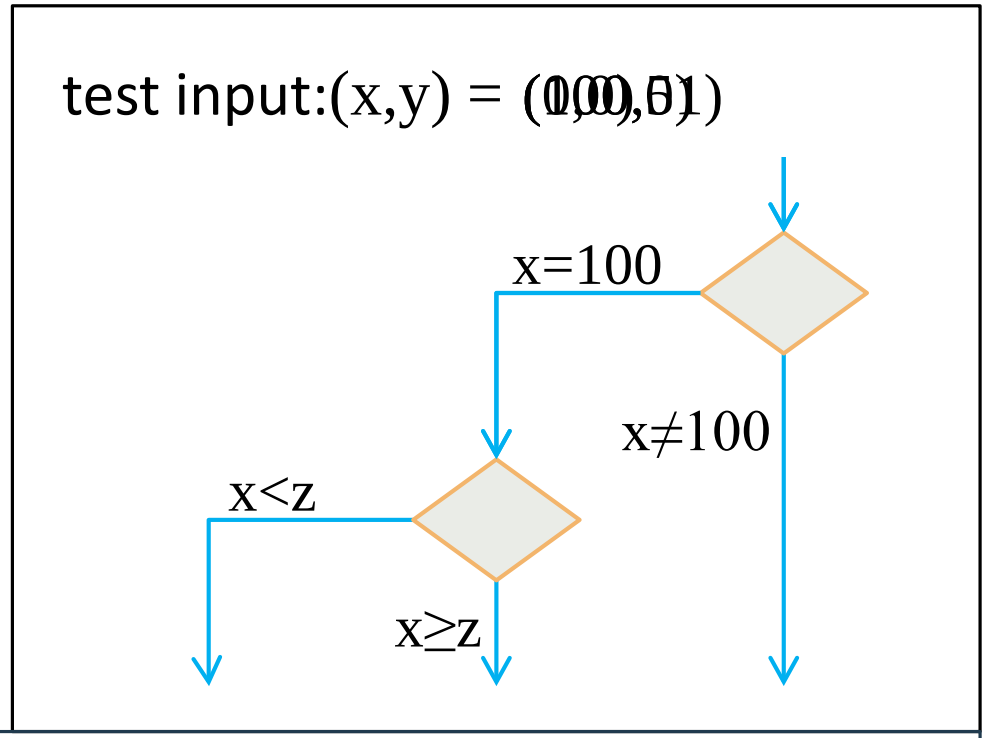
- 静态符号执行所面临的现实问题：
  - 程序中的几乎每条指令都是符号变量的逻辑表达式
  - 随着程序执行，符号逻辑的复杂程度迅速爆炸
  - 无法处理循环
- 对静态符号执行的改进 – **与动态的具体执行相结合**：
  - 程序输入不再由代数符号表示，而是赋予具体数值
  - 执行过程中，遇到无法完全具体化的指令时，才对其进行符号执行

软件验证和漏洞检测技术

- 动态符号执行（又称concolic testing，concolic = **conc**rete + symb**olic**）：

PROGRAM:

```
void f(int x, int y) {
    int z = 2 * y;
    if (x == 100) {
        if (x < z) {
            assert(0);
        }
    }
}
```



test cases:

软件验证和漏洞检测技术

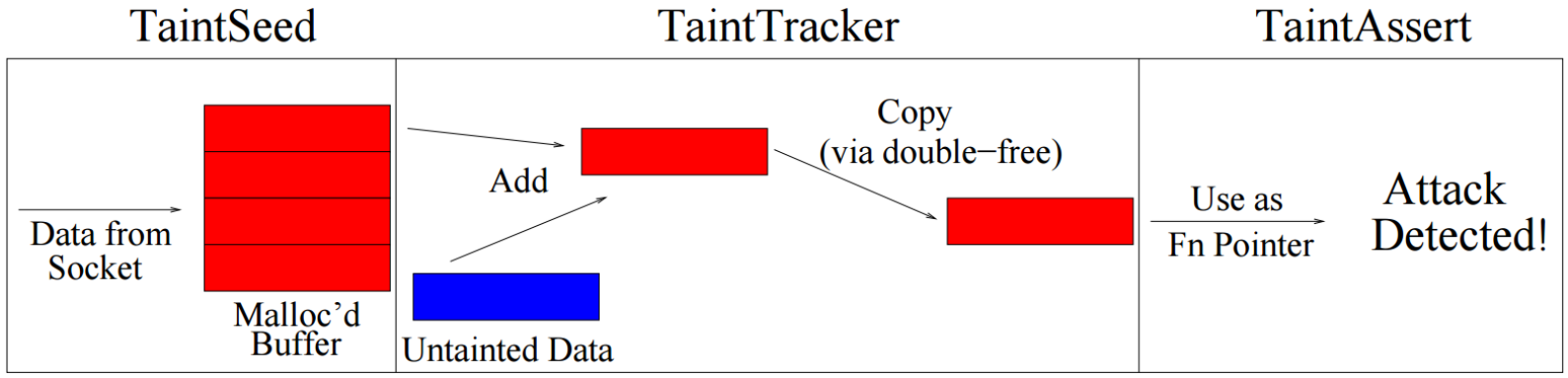
- 动态符号执行（又称concolic testing，concolic = **conc**rete + symb**olic**）：

```
1  POWER: PROCEDURE(X, Y);
2              Z ← 1;
3              J ← 1;
4  LAB:        IF Y ≥ J THEN
5              DO; Z ← Z * X;
6                  J ← J + 1;
7                  GO TO LAB; END;
8              RETURN (Z);
9  END;
```

X=2；Y=10

软件验证和漏洞检测技术

- 污点分析：
  - 所针对的核心现象 -- 恶意攻击中，攻击者需要用其输入覆盖掉合法派生的数据
  - 将不可信的输入标记为“脏”数据，并监控执行
  - 当以“脏”数据为源的操作派生出新数据时，标记新数据也为“脏”的
  - 若发现有“脏”数据被用于某些有风险的操作，则报告



软件验证和漏洞检测技术

- 面向更高效的软件验证：

静态符号执行

对所有操作符号化表示并进行推理

动态符号执行

只将与执行路径分叉直接有关的操作符号化

动态符号执行
+
污点分析

只将与路径分叉有关的“脏”数据操作符号化

软件验证和漏洞检测技术

- 参考文献：
  - Oehlert P. Violating assumptions with fuzzing[J]. IEEE Security & Privacy, 2005, 3(2): 58-62.
  - King J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7): 385-394.
  - Cadar C, Sen K. Symbolic execution for software testing: three decades later[J]. Communications of the ACM, 2013, 56(2): 82-90.
  - Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[J]. 2005.

What’s next?

- 软件自我保护技术
  - 概述
  - 安全模型
  - 主要技术方法简介