Software Security

Program Analysis Techniques（2）

Taint Analysis

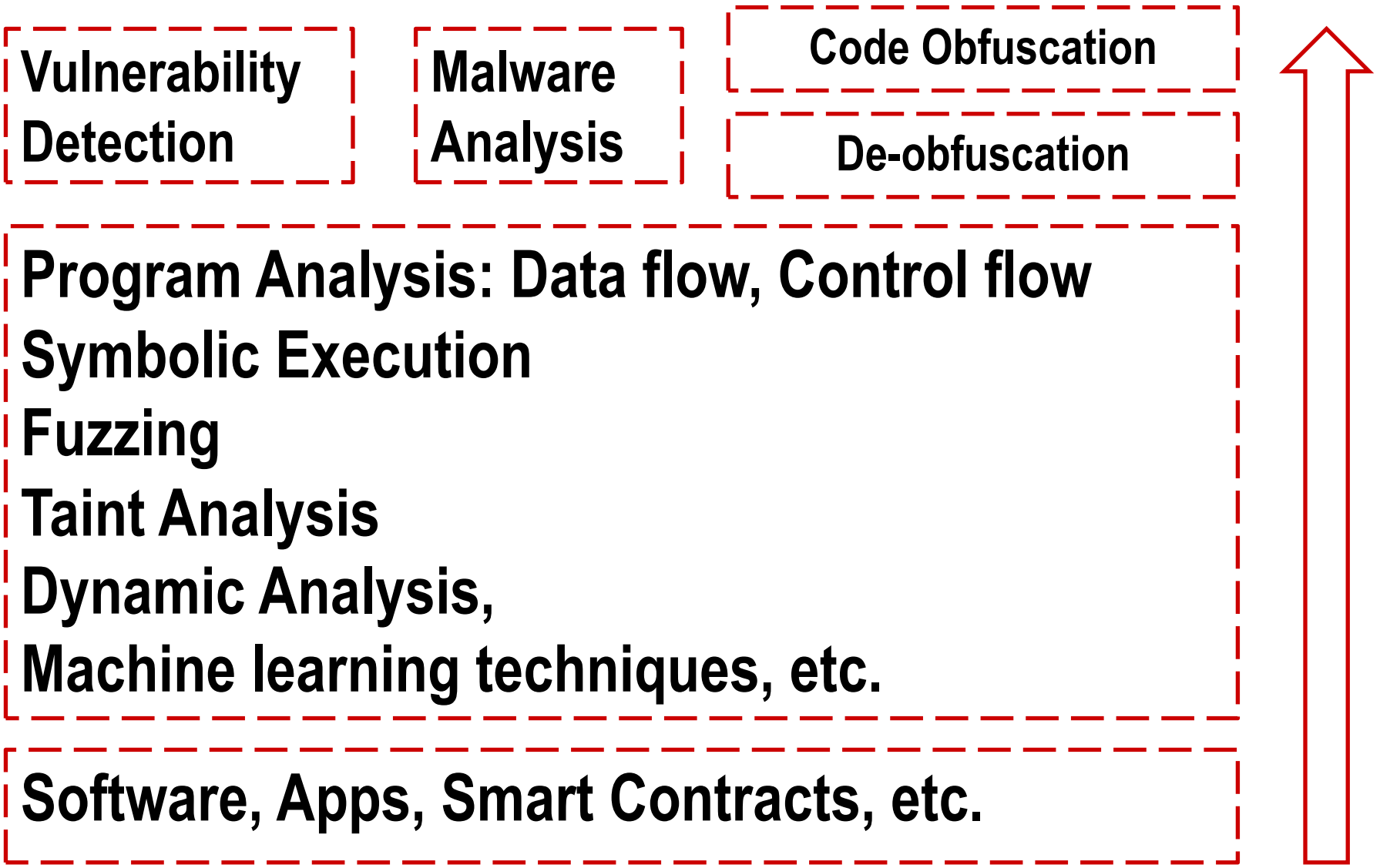
Guosheng Xu,

guoshengxu@bupt.edu.cn

Outline

- Techniques used in Security Analysis
- Basic Program Analysis
  - Control flow analysis
  - Data flow analysis
- **Taint Analysis**
- **Symbolic execution**
- **Fuzzing**

Techniques used in security analysis



Dataflow Analysis -- Continue

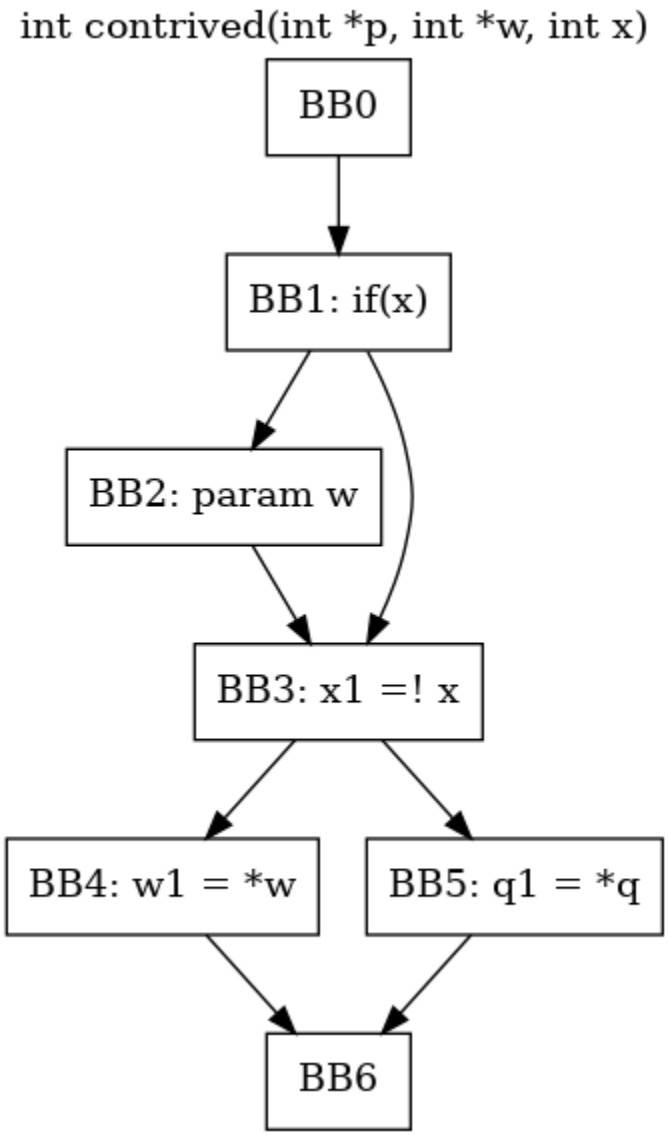
- Flow Insensitive(流不敏感分析)
  - 不考虑语句的先后顺序，按照程序语句的物理位置从上往下顺序分析每一语句，忽略程序中存在的分支
- Flow Sensitive(流敏感分析)
  - 考虑程序语句可能的执行顺序，通常需要利用程序的控制流图（CFG）
- Path Sensitive(路径敏感分析)：
  - 不仅考虑语句的先后顺序，还对程序执行路径条件加以判断，以确定分析使用的语句序列是否对应着一条可实际运行的程序执行路径

Dataflow Analysis -- Continue

- **Intra-procedure analysis(过程内分析)**
  - 只针对程序中函数内的代码
- **Inter-procedure analysis(过程间分析)**
  - 考虑函数之间的数据流，即需要跟踪分析目标数据在函数之间的传递过程
  - 上下文不敏感分析（context-insensitive）：将每个调用或返回看做一个“goto”操作，忽略调用位置和函数参数取值等函数调用的相关信息
  - 上下文敏感分析（context-sensitive）：对不同调用位置调用的同一函数加以区分

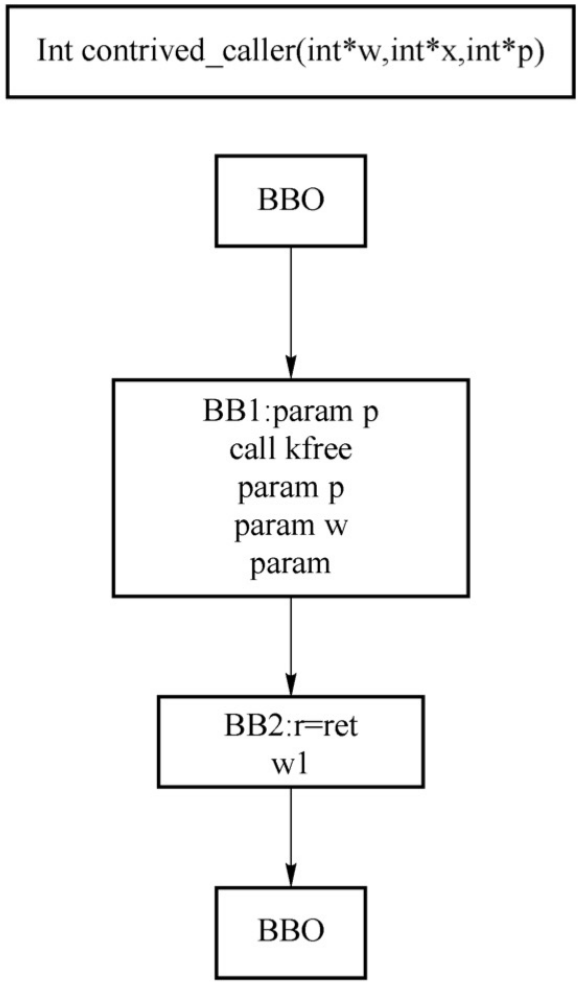
Dataflow Analysis - Example

```
int contrived(int *p, int *w, int x) {
        int *q;
        if (x) {
                kfree(w); // w free
                q = p;
        } [...]
        if (!x)
                return *w;
        return *q; // p use after free
}
int contrived_caller(int *w, int x, int *p) {
        kfree(p); // p free
        [...]
        int r = contrived(p, w, x);
        [...]
        return *w; // w use after free
}
```



Dataflow Analysis - Example

```
int contrived(int *p, int *w, int x) {
        int *q;
        if (x) {
                kfree(w); // w free
                q = p;
        } [...]
        if (!x)
                return *w;
        return *q; // p use after free
}
int contrived_caller(int *w, int x, int *p) {
        kfree(p); // p free
        [...]
        int r = contrived(p, w, x);
        [...]
        return *w; // w use after free
}
```



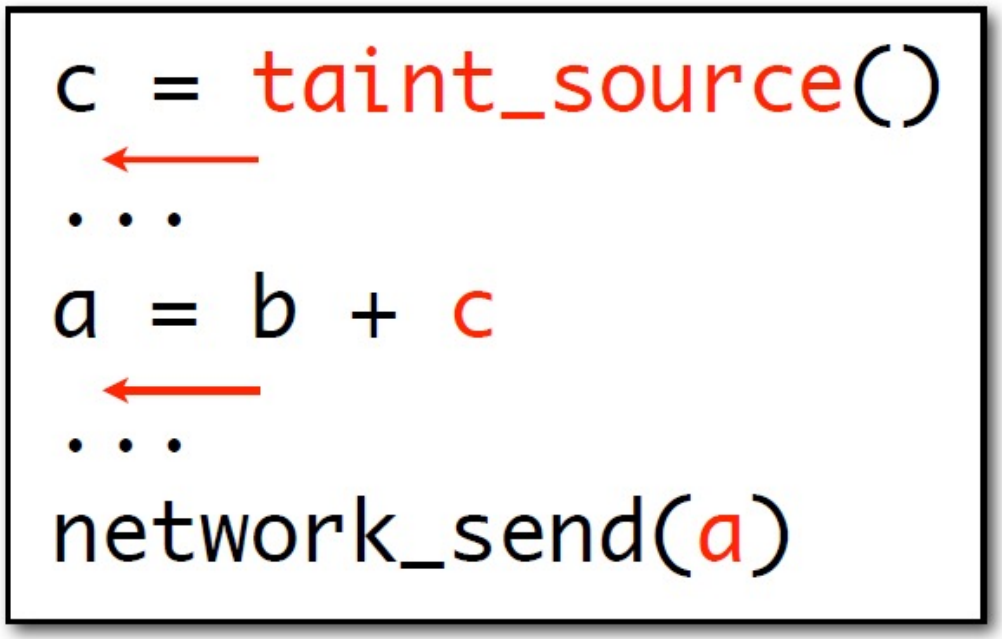
TAINT ANALYSIS 污点分析

Taint Analysis

- Taint analysis is a technique that tracks information dependencies from an origin

Conceptual idea:

- Taint source
- Taint propagation
- Taint sink



Information Flow

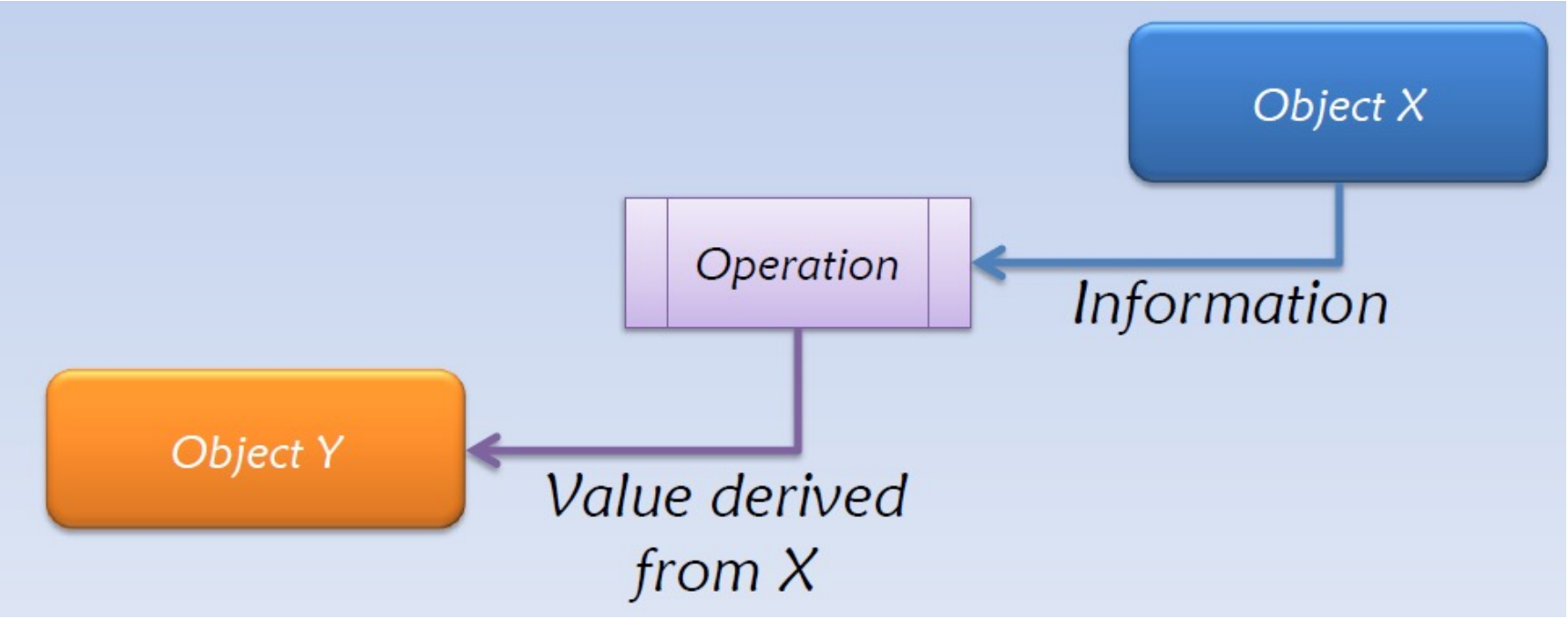
- Follow any application inside a debugger and you'll see that data information is being copied and modified all the time. In another words, *information is always moving.*
- Taint analysis can be seen as a form of Information Flow Analysis.
- Great definition provided by Dorothy Denning at the paper "*Certification of programs for secure information flow*":
  - "*Information* flows *from object x to object y, denoted x→y , whenever information stored in x is transferred to, object y.*"

每当存储在x中的信息传输到对象y时，信息便从对象x流向对象y，表示为x→y。

Flow

使用某个对象（例如x）的值来推导另一个对象（例如y）的值的操作或一系列操作会导致从x到y的流动。

- *"An operation, or series of operations, that uses the value of some object, say x, to derive a value for another, say y, causes a* **flow** *from x to y."*



Explicit/Implicit Information Flow

- 污点信息不仅可以通过数据依赖传播，还可以通过控制依赖传播
- 通过数据依赖传播的信息流称为显式信息流，将通过控制依赖传播的信息流称为隐式信息流。

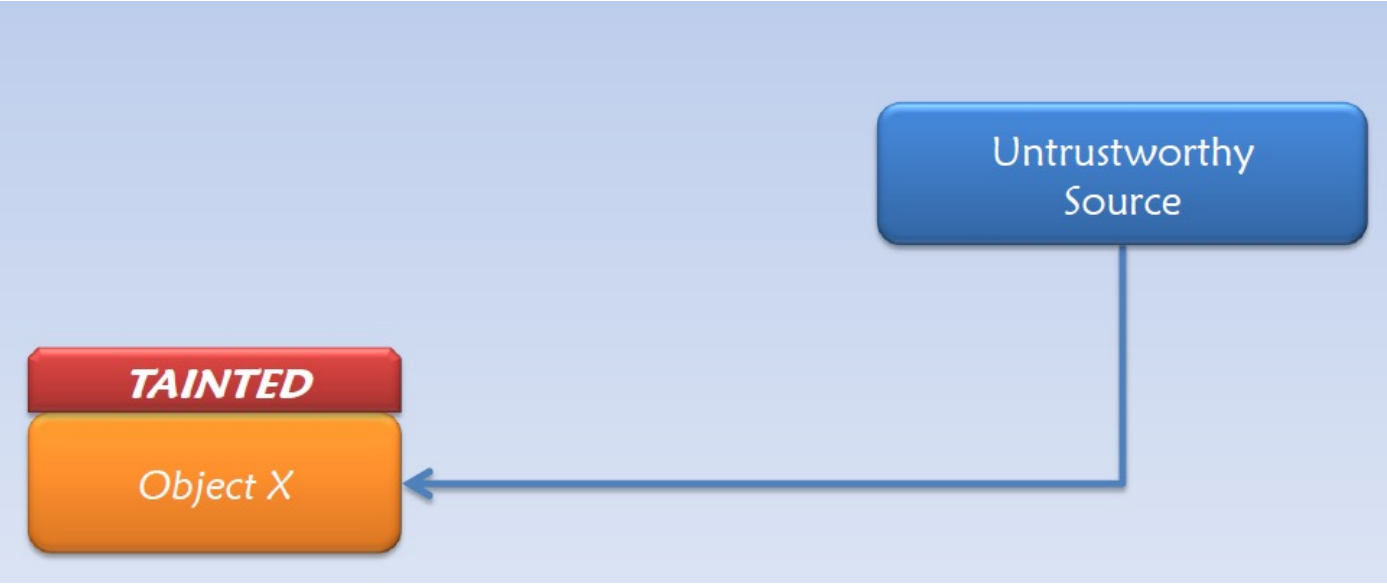
```
if (x > 0)
        y = 1;
else
        y = 0;
```

变量 y 的取值依赖于变量 x 的取值——如果变量 x 是污染的，那么变量 y 也应该是污染的

Tainted objects

如果对象X的值的来源不可信，则可以说X是受污染的。

- If the **source** of the value of the object *X* is **untrustworthy**, we say that *X* is **tainted**.
- Usually, it depends on the performed tasks
  - E.g., for privacy leakage detection, the source is sensitive data/information



Taint

要“污染”用户数据，就是为用户数据的每个对象插入某种标签或标签。

- To “taint” user data is to insert some kind of tag or **label** for each object of the user data.
- The tag allow us to track the influence of the tainted object along the execution of the program.

Taint Sources

- Sensitive APIs (location, SMS, …)
- Files (*.mp3, *.pdf, *.svg, *.html, *.js, …)
- Network protocols (HTTP, UDP, DNS, ...)
- Keyboard, mouse and touchscreen input messages
- Webcam
- USB
- Virtual machines (Vmware images)

Taint propagation/污染传播

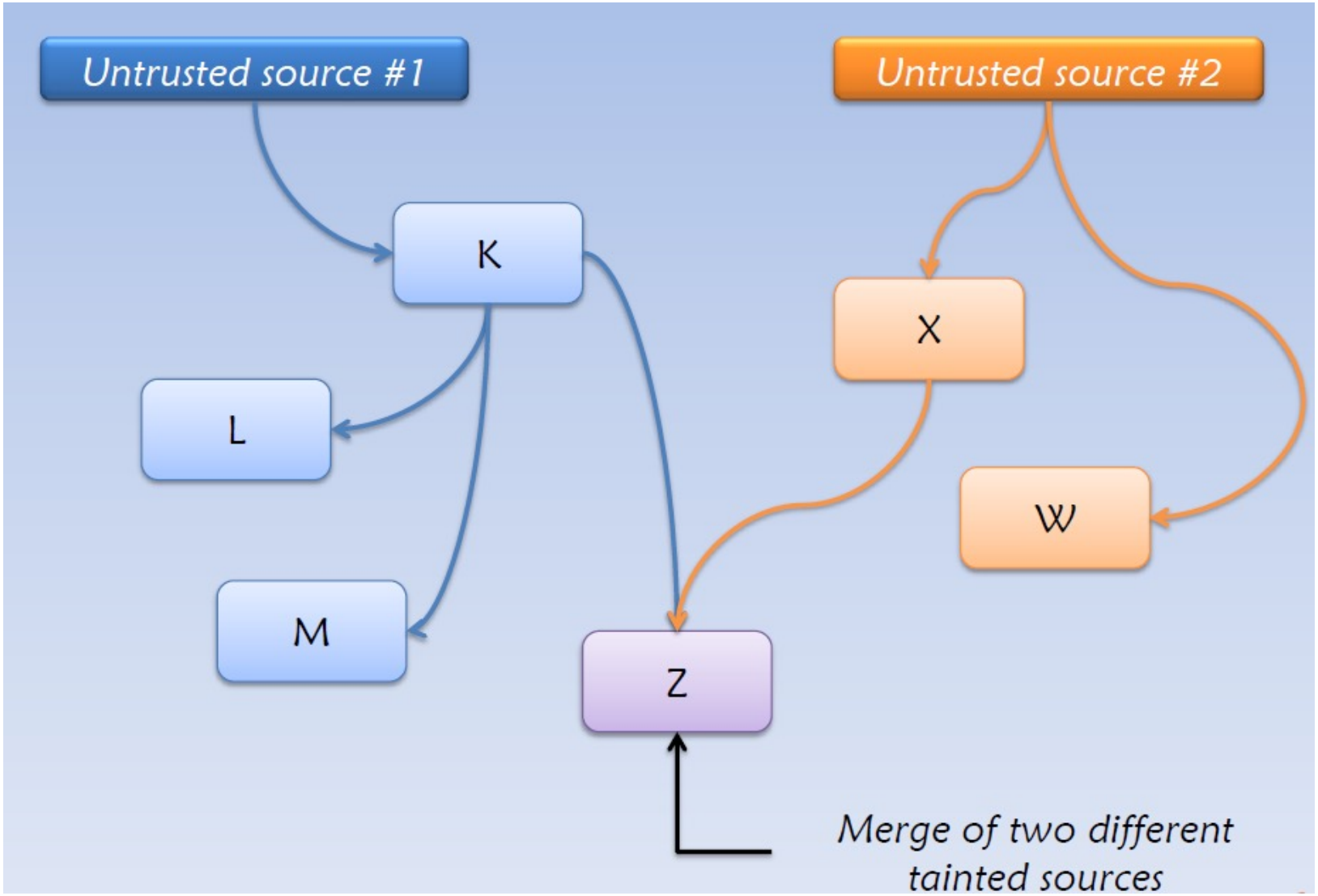
- If an operation uses the value of some **tainted** object, say X, to derive a value for another, say Y, then object Y becomes **tainted.**

  -> Object X tainted the object Y

- Taint operator **t**
  - X→ t(Y)
- Taint operator is transitive/传递性

  –X → t(Y) and Y→ t(Z), then X→ t(Z)

Taint propagation



Applications

- Exploit detection
  - If we can track user data, we can detect if non-trusted data reaches a privileged location
  - SQL injection, buffer overflows, XSS, …
  - Perl tainted mode
  - Detects even unknown attacks!
  - Taint analysis for web applications
- Before execution of any statement, the taint analysis module checks if the statement is tainted or not! If tainted issue an attack alert!

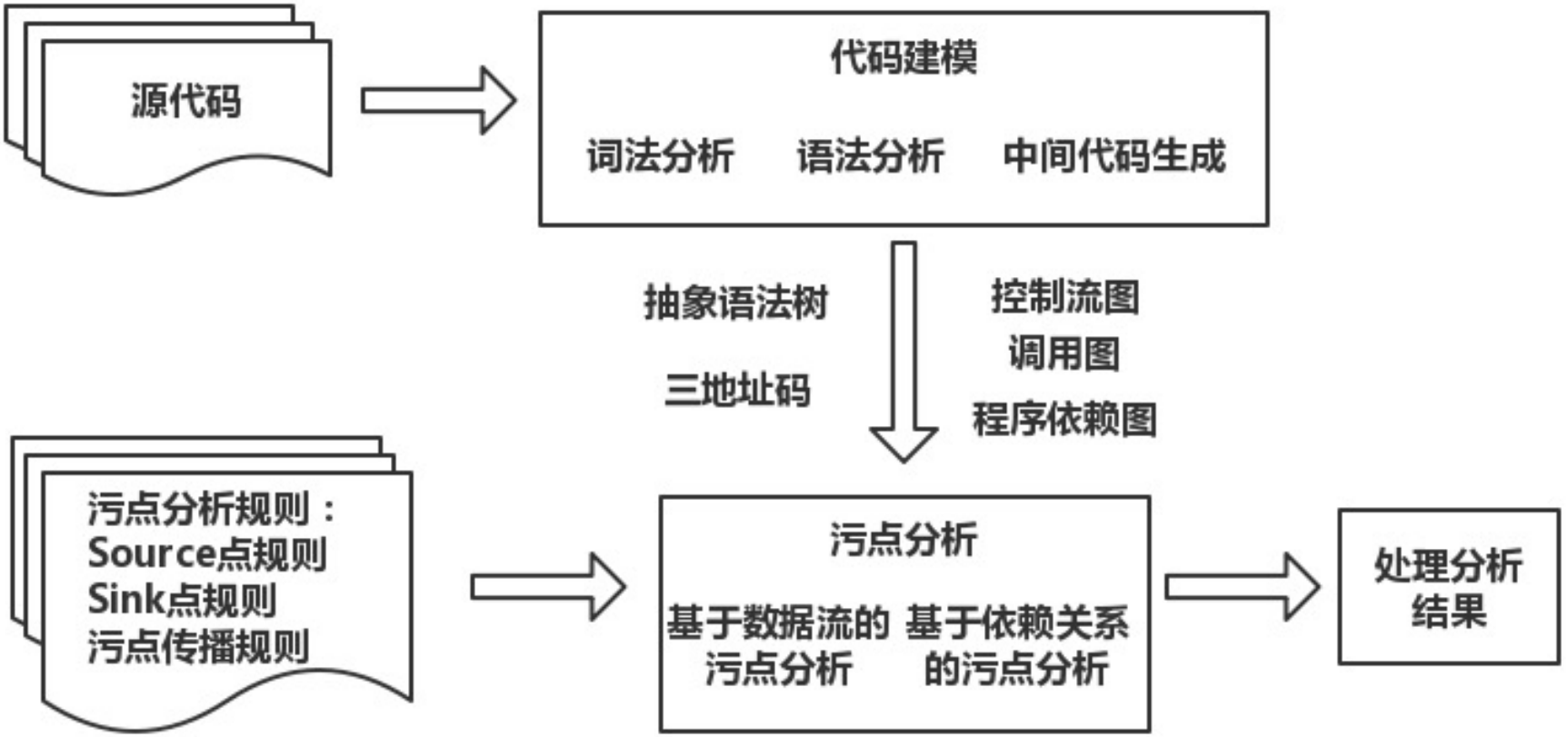
Example

```
String user = getUser();
String pass = getPass();
String sqlQuery = "select * from login where user='" + user +
"' and pass='" + pass + "'";
Statement stam = con.createStatement(); ResultSetrs =
stam.executeQuery(sqlQuery); if (rs.next())
        success = true;
```

Applications

- Data Lifetime analysis
  - Jin Chow –“Understanding data lifetime via whole system emulation” –presented at Usenix‟04.
  - Created a modified Bochs (TaintBochs) emulator to taint sensitive data.
  - Keep track of the lifetime of sensitive data (passwords, pin numbers, credit card numbers) stored in the virtual machine memory
  - Tracks data even in the kernel mode.
  - Concluded that most applications doesn’t have any measure to minimize the lifetime of the sensitive data in the memory.

Static Taint Analysis



Static Taint Analysis

- **基于数据流的污点分析**
  - 在不考虑隐式信息流的情况下，可以将污点分析看做针对污点数据的数据流分析
  - 根据污点传播规则跟踪污点信息或者标记路径上的变量污染情况，进而检查污点信息是否影响敏感操作
- **基于依赖关系的污点分析**
  - 考虑隐式信息流，在分析过程中，根据程序中的语句或者指令之间的依赖关系，检查 Sink 点处敏感操作是否依赖于 Source 点处接收污点信息的操作。

基于数据流的污点分析

- 在基于数据流的污点分析中，常常需要一些辅助分析技术，例如别名分析、取值分析等，来提高分析精度
- 辅助分析和污点分析交替进行，通常沿着程序路径的方向分析污点信息的流向，检查 Source 点处程序接收的污点信息是否会影响到 Sink 点处的敏感操作

过程内的分析

- 过程内的分析中，按照一定的顺序分析过程内的每一条语句或者指令，进而分析污点信息的流向。
- **记录污点信息**
- **程序语句分析**
- **代码遍历**

记录污点信息

- 在静态分析层面，主要关注程序变量
- 为记录污染信息，通常为变量添加一个污染标签(taint tag)
  - 最简单的就是一个布尔型变量，表示变量是否被污染。
- 复杂的标签可以记录变量的污染信息来自哪些Source ，甚至精确到 Source 点接收数据的哪一部分。
- 也可以通过对变量进行跟踪的方式达到分析污点信息流向的目的
  - E.g., 使用栈或者队列来记录被污染的变量

程序语句的分析

- 在确定如何记录污染信息后，将对程序语句进行静态分析
  - 赋值语句
  - 控制转移语句
  - 过程调用语句

程序语句的分析-赋值语句

- **简单赋值语句**
  - E.g., a=b
  - 记录语句左端的变量和右端的变量具有相同的污染状态
  - 程序中的常量通常认为是未污染的，如果一个变量被赋值为常量，在不考虑隐式信息流的情况下，认为变量的状态在赋值后是未污染的
- **二元操作的赋值语句**
  - E.g., a = b + c
  - 通常规定如果右端的操作数只要有一个是被污染的，则左端的变量是污染的

程序语句的分析-赋值语句

- **数组相关赋值**
  - 如果可以通过静态分析确定数组下标的取值或者取值范围，那么就可以精确地判断数组中哪个或哪些元素是污染的。
  - 但通常静态分析不能确定一个变量是污染的，那么就简单地认为整个数组都是污染的。
- **包含指针操作的赋值语句**
  - 需要用到指向分析的分析结果

程序语句的分析-控制转移语句

- **条件控制转移语句**
  - 考虑语句中的路径条件可能是包含对污点数据的限制
  - 在实际分析中常常需要识别这种限制污点数据的条件，以判断这些限制条件是否足够包含程序不会受到攻击
  - 如果得出路径条件的限制是足够的，那么可以将相应的变量标记为未污染的

程序语句的分析-控制转移语句

- **循环语句**
  - 通常规定循环变量的取值范围不能受到输入的影响
  - 否则可能会存在隐式信息流
  - 例如在语句 for (i = 1; i < k; i++){} 中，可以规定循环的上界 k 不能是污染的

程序语句的分析-过程调用语句

- 使用过程间分析或者直接应用过程摘要进行分析
- 过程摘要
  - 主要描述怎样改变与该过程相关的变量的污染状态，以及对哪些变量的污染状态进行检测
- 这些变量可以是过程使用的参数、参数的字段或者过程的返回值等。
  - E.g., 语句 flag = obj.method(str); 中, str 是污染的
  - 通过过程间分析，将变量 obj 的字段 str 标记为污染的；而记录方法的返回值的变量 flag 标记为未污染的。

程序语句的分析-过程调用语句

- 过程摘要的复用
  - 在实际的过程间分析中，可以对已经分析过的过程构建过程摘要
- 语句 flag = obj.method(str);
  - 其过程摘要描述为：方法 method 的参数污染状态决定其接收对象的实例域 str 的污染状态，并且它的返回值是未受污染的。
  - 下次分析时，可以直接应用摘要进行分析

代码的遍历

- 常使用流敏感的方式或者路径敏感的方式进行遍历，并分析过程中的代码。
- 流敏感的遍历方式
  - 可以通过对不同路径上的分析结果进行汇集，以发现程序中的数据净化规则。
- 路径敏感的遍历方式
  - 需要关注路径条件
  - 如果路径条件中涉及对污染变量取值的限制，可认为路径条件对污染数据进行了净化
  - 可以将分析路径条件对污染数据的限制进行记录，
  - 如果在一条程序路径上，这些限制足够保证数据不会被攻击者利用，就可以将相应的变量标记为未污染的

基于依赖关系的污点分析

- 构造程序的依赖关系
  - 利用程序的中间表示、控制流图和过程调用图构造程序完整的或者局部的程序的依赖关系
  - 在分析程序依赖关系后，根据污点分析规则，检测 Sink 点处敏感操作是否依赖于 Source 点
- 程序依赖图
  - 一个有向图
  - 节点是程序语句，有向边表示程序语句之间的依赖关系。程序依赖图的有向边常常包括数据依赖边和控制依赖边
  - 在构建有一定规模的程序的依赖图时，需要按需地构建程序依赖关系，并且优先考虑和污点信息相关的程序代码

Program Dependence Graph

- **Control dependency**
  - a program instruction executes if the previous instruction evaluates in a way that allows its execution.

```
S1      if x > 2 goto L1
S2      y := 3
S3   L1: z := y + 1
```

statement *S2* is *control dependent* on *S1*

Program Dependence Graph

- **Flow dependence**
  - A statement *S2* is **flow dependent** on *S1* if and only if S1 modifies a resource that S2 reads and S1 precedes S2 in execution. The following is an example of a flow dependence

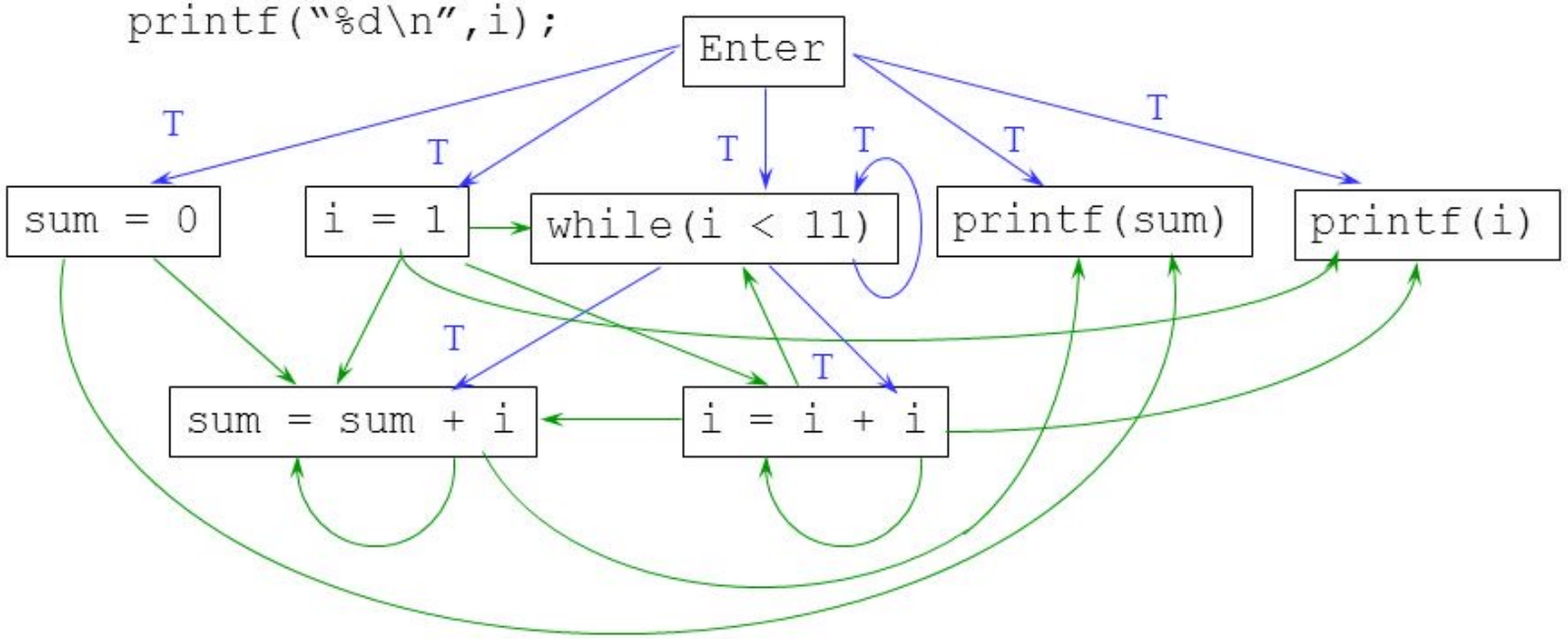
```
S1 x := 10
S2 y := x + c
```

Program Dependence Graph

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

Control dependence

Flow dependence



动态污点分析

- **动态污点分析**
  - 在程序运行的基础上，对数据流或控制流进行监控，从而实现对数据在内存中的显式传播、数据误用等进行跟踪和检测。
- 动态污点分析与静态污点分析的唯一区别在于静态污点分析技术在检测时并不真正运行程序，而是通过模拟程序的执行过程来传播污点标记，而动态污点分析技术需要运行程序，同时实时传播并检测污点标记。
  - **污点数据标记**
  - **污点动态追踪**
  - **污点误用检查**

污点数据标记

- 污点数据通常主要是指软件系统所接受的外部输入数据，在计算机中，这些数据可能以内存临时数据的形式存储，也可能以文件的形式存储
- 当程序需要使用这些数据时，一般通过函数或系统调用来进行数据访问和处理
  - 只需要对这些关键函数进行监控，即可得到程序读取或输出了什么污点信息。另外对于网络输入，也需要对网络操作函数进行监控
- 污点生命周期是指在该生命周期的时间范围内，污点被定义为有效
  - 污点生命周期开始于污点创建时刻，生成污点标记，结束于污点删除时刻，清除污点标记

污点数据标记

- 污点创建
  - 将来自于非可靠来源的数据分配给某寄存器或内存操作数时
  - 将已经标记为污点的数据通过运算分配给某寄存器或内存操作数时
- 污点删除
  - 将非污点数据指派给存放污点的寄存器或内存操作数时
  - 将污点数据指派给存放污点的寄存器或内存地址时，此时会删除原污点，并创建新污点
  - 一些会清除污点痕迹的算数运算或逻辑运算操作时

污点动态追踪

- 污点动态跟踪：在污点数据标记的基础上，对进程进行指令粒度的动态跟踪分析，分析每一条指令的效果，直至覆盖整个程序的运行过程，跟踪数据流的传播。
  - **动态代码插桩**：可以跟踪单个进程的污点数据流动，通过在被分析程序中插入分析代码，跟踪污点信息流在进程中的流动方向。
  - **全系统模拟**：利用全系统模拟技术，分析模拟系统中每条指令的污点信息扩散路径，可以跟踪污点数据在操作系统内的流动。
  - **虚拟机监视器**：通过在虚拟机监视器中增加分析污点信息流的功能，跟踪污点数据在整个客户机中各个虚拟机之间的流动。

污点动态追踪

- 当污点数据从一个位置传递到另一个位置时，则认为产生了污点传播。污点传播规则：

指令类型	传播规则	举例说明
拷贝或移动指令	T(a)<-T(b)	mov a, b
算数运算指令	T(a)<-T(b)	add a, b
堆栈操作指令	T(esp)<-T(a)	push a
拷贝或移动类函数调用指令	T(dst)<-T(src)	call memcpy
清零指令	T(a)<-false	xor a, a

污点Sink检查

- 污点敏感点，即 Sink 点，是污点数据有可能被误用的指令或系统调用点，主要分为：
  - 跳转地址：检查污点数据是否用于跳转对象，如返回地址、函数指针、函数指针偏移等。具体操作是在每个跳转类指令（如call、ret、jmp等）执行前进行监控分析，保证跳转对象不是污点数据所在的内存地址。
  - 格式化字符串：检查污点数据是否用作printf系列函数的格式化字符串参数。
  - 系统调用参数：检查特殊系统调用的特殊参数是否为污点数据。
  - 标志位：跟踪标志位是否被感染，及被感染的标志位是否用于改变程序控制流。
  - 地址：检查数据移动类指令的地址是否被感染。

Further Reading

- **Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software,** valgrind.org/docs/newsome2005.pdf
- **TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones, OSDI 2010, http://www.appanalysis.org/tdroid10.pdf**