

PtrSplit: Supporting General Pointers in Automatic Program Partitioning

Shen Liu Gang Tan Trent Jaeger

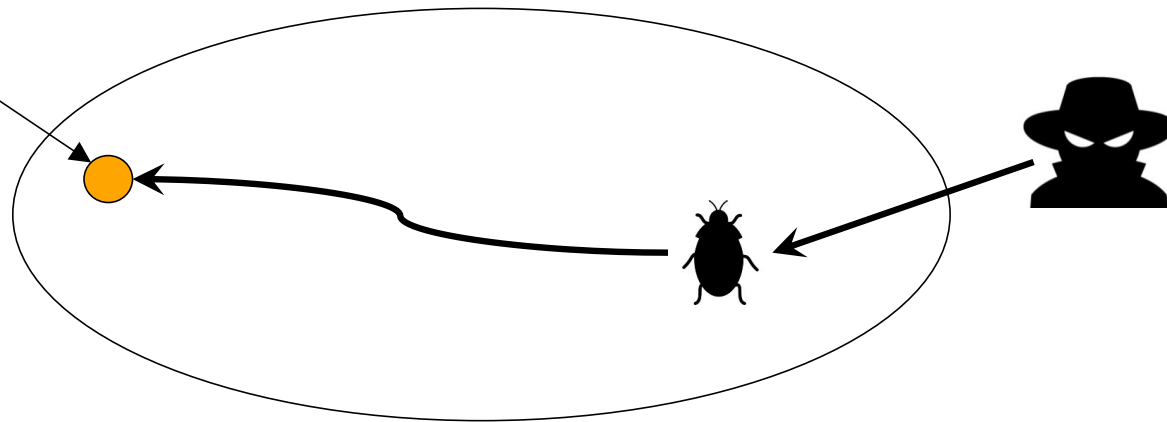
Computer Science and Engineering Department

The Pennsylvania State University

11/02/2017

Motivation for Partitioning

Sensitive data

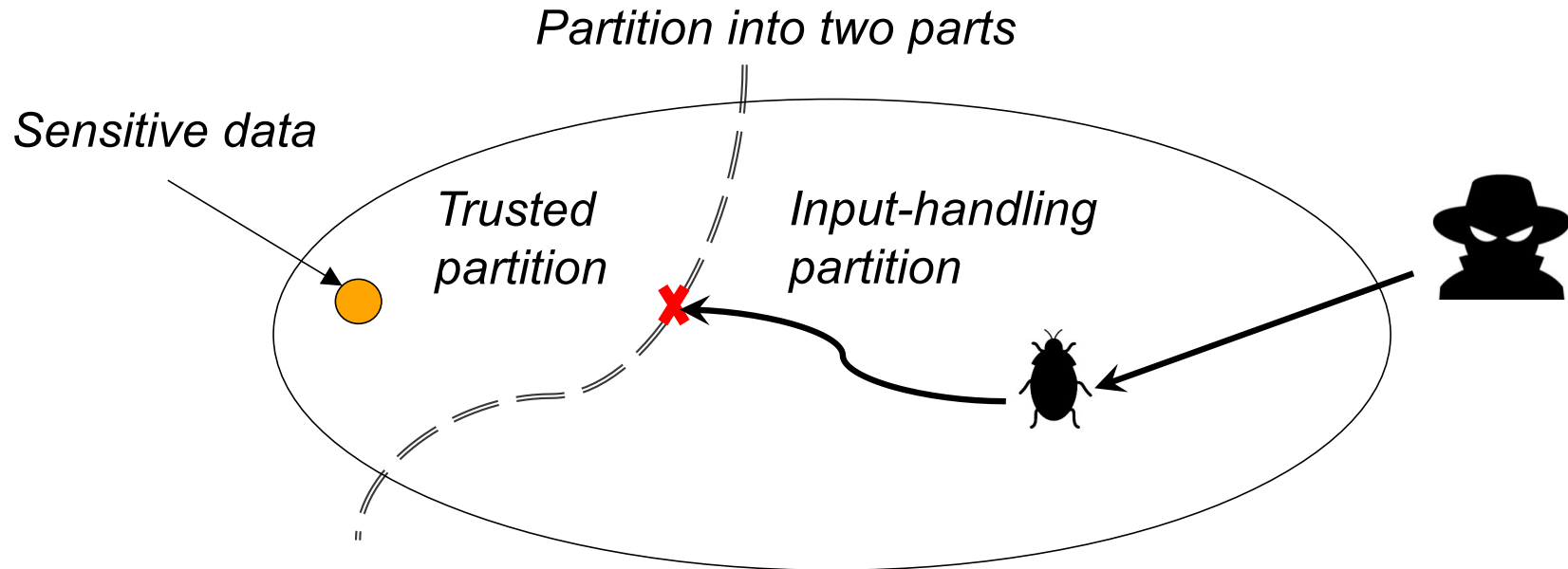


A monolithic, security-sensitive program

A single bug would defeat the security of the whole application

Motivation for Partitioning

- Split the application into multiple partitions
- Each partition is isolated using some isolation mechanism such as OS processes



Although some partition of a program has been hijacked, sensitive data can still be protected

Toy Example

```
char* cipher;
```

```
char* key;
```

Sensitive data

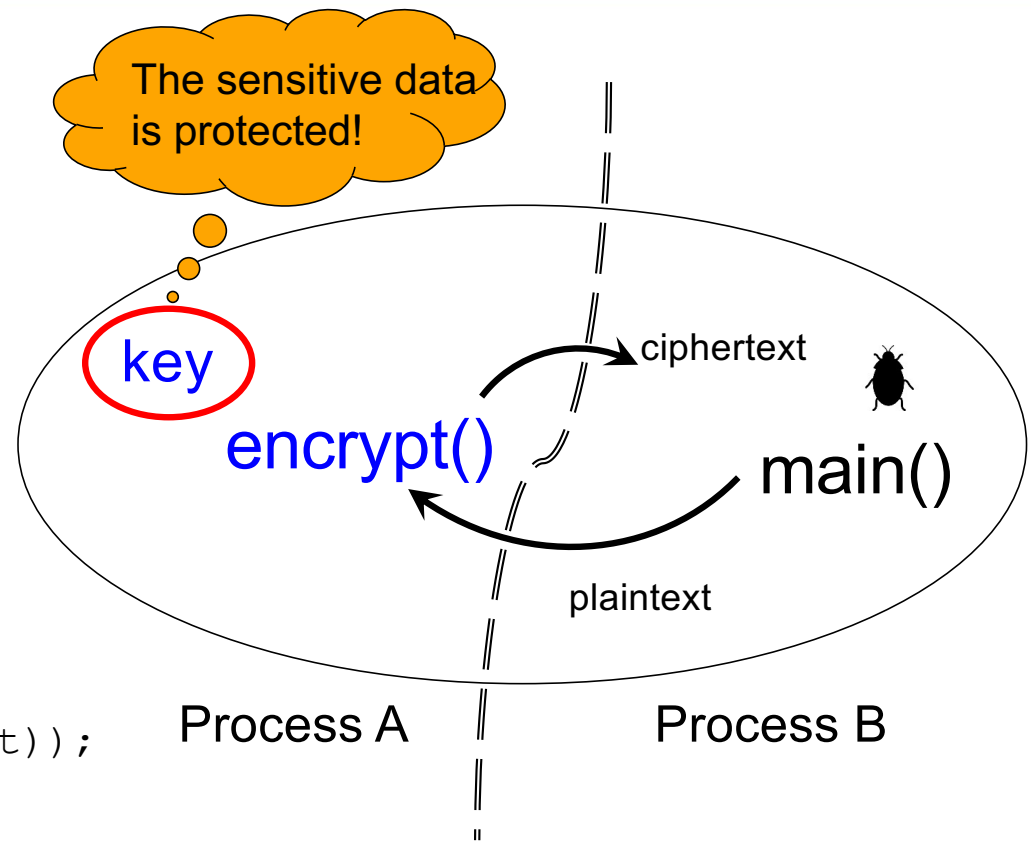
```
void encrypt(char *plain, int n){  
    cipher =(char*)malloc(n);  
    for (i = 0; i < n; i++)  
        cipher[i] = plain[i] ^ key[i];  
}
```

```
void main () {  
    char plaintext[1024];  
    scanf("%s",plaintext);  
    encrypt(plaintext,strlen(plaintext));  
    ...  
}
```

Buffer overflow

Toy Example

```
char* cipher;  
char* key;  
  
void encrypt(char *plain, int n){  
    cipher =(char*)malloc(n);  
    for (i = 0; i < n; i++)  
        cipher[i] = plain[i] ^ key[i];  
}  
  
void main (){  
    char plaintext[1024];  
    scanf("%s",plaintext);  
    encrypt(plaintext,strlen(plaintext));  
    ...  
}
```



Solution

- Manual partitioning
 - do **code review** and extract the sensitive components
 - The amount of code for analysis may be huge...
- Automatic partitioning
 - Given some security criteria, do partitioning based on **static program analysis**
 - Reduce manual effort and errors

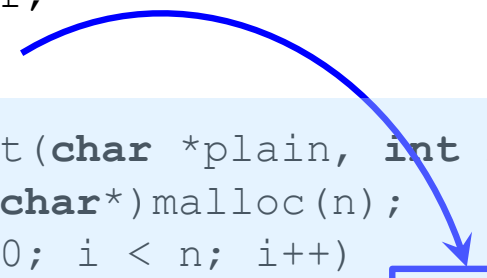
Background: static program analysis

■ Static analysis

- Analyzing code without executing it
- Static analysis can be considered as automated code review
- e.g., Annotate a sensitive variable key, we can find all the statements that key can reach.

```
char* cipher;  
char* key;
```

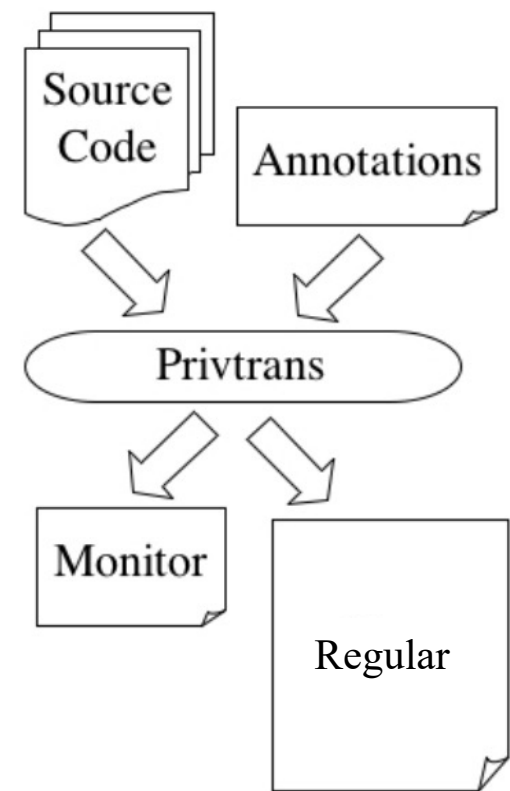
```
void encrypt(char *plain, int n){  
    cipher = (char*)malloc(n);  
    for (i = 0; i < n; i++)  
        cipher[i] = plain[i] ^ key[i];  
}
```



```
void main () {  
    char plaintext[1024];  
    scanf("%s", plaintext);  
    encrypt(plaintext, strlen(plaintext));  
    ...  
}
```

Previous Work: Privtrans (2004)

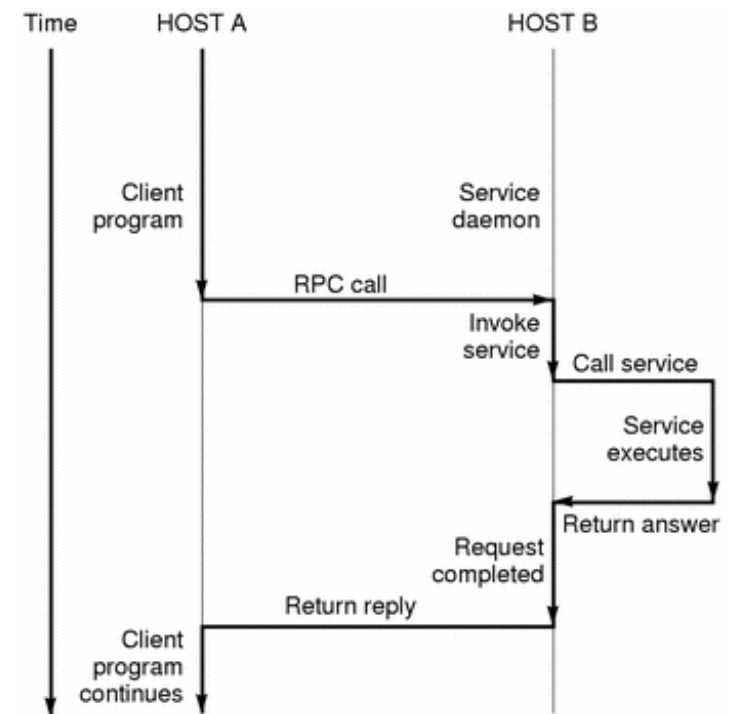
- Privtrans automatically incorporates privilege separation into source code by partitioning it into two programs
 - A **monitor** program which handles privileged operations
 - A **regular** program which executes everything else
 - Users need to manually add a few annotations to help Privtrans decide how to partition
 - The inter-process communication between partitions is implemented by Remote Procedure Calls (RPCs)



Privtrans' principle (copied from the paper)

Background: Remote Procedure Call(RPC)

- RPC enables a program to call procedures in a different address space
 - Programmers need to tell RPC what functions will be called remotely and define the interfaces
 - In an interface definition language (IDL) file
 - IDL compiler can generate code to transmit data between the client and servers (i.e., via RPCs)
 - Data transmission method depends on communication media between processes (network, IPC)



How RPC works(copied from the TI-RPC manual)

Previous Work

- Systems for automatic program partitioning
 - **Privman** by Kilpatrick (USENIX ATC 2003)
 - **Privtrans** by Brumley and Song (USENIX Security 2004)
 - **Wedge** by Bittau, Marchenko, Handley, and Karp (USENIX NSDI 2008)
 - **ProgramCutter** by Wu, Sun, Liu, and Dong (ASE 2013)
- Major limitation: lack of automatic support for pointers
 - Pointers prevalent in C/C++ applications
 - Previous work
 - Lack sound reasoning of pointers [to find functions that reference sensitive data](#)
 - Require manual intervention when pointers are passed across partition boundaries – [to find the size of the referenced memory region to copy](#)

Determine All the Functions in a Partition

- We aim to include all the functions that may operate on the sensitive data within the same sensitive partition
 - Which functions are those?
 - Any function that has access to the sensitive data
 - I.e., any function with a pointer that may point to (alias) the sensitive data
- For sound program partitioning, we have to reason about **all program executions**
 - Need to know what control flows a program may take
 - Which pointers may alias which memory objects
 - And which data depends on which other data
 - Need a **global alias analysis** for tracking data dependence

Background: Aliases

- What will happen when two pointers refer to the same memory location

Example 1:

```
int x;  
p = &x;  
q = p; // <*p,*q>, <x,*p> and <x,*q> are all aliases now
```

Example 2:

```
int i,j, a[100];  
i = j; // a[i] and a[j] are aliases now
```

- Alias analysis is undecidable (G. Ramalingam, TOPLAS 1994)
 - For large programs, alias analysis can identify many possible aliases for some memory locations (e.g., Linux kernel or browser)

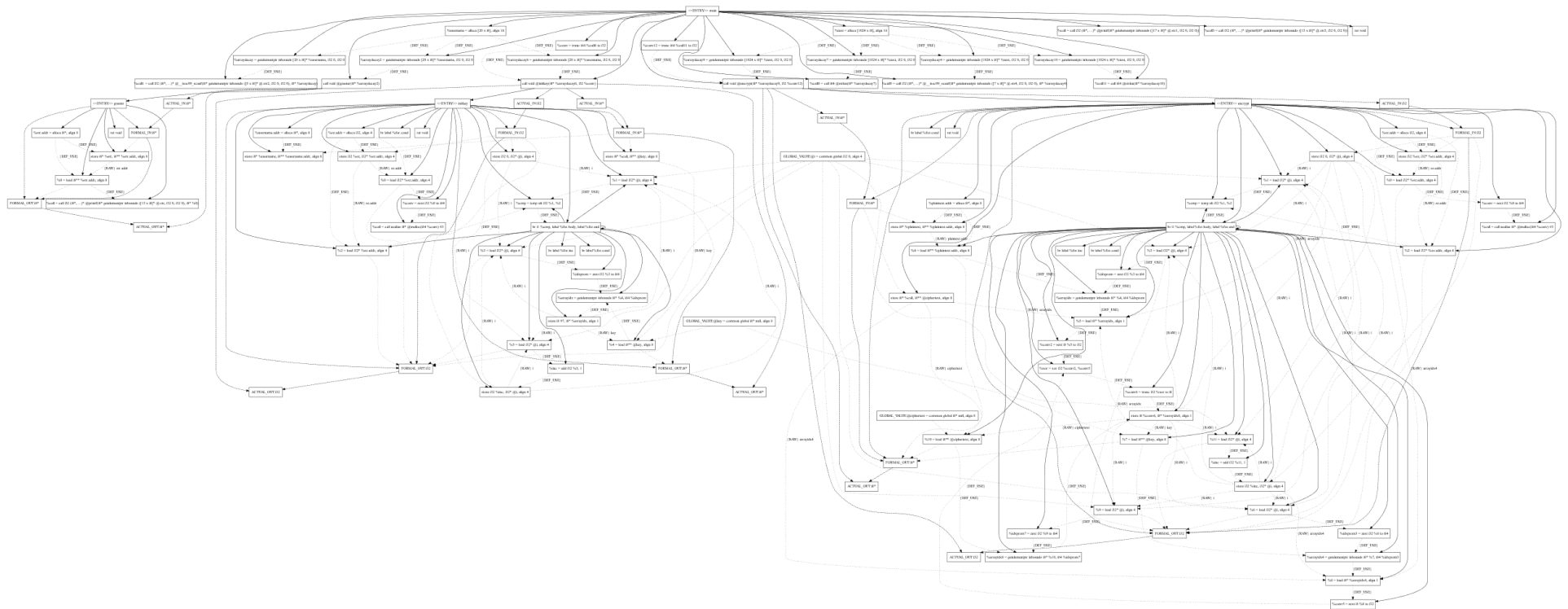
Lack of Bounds Information with Pointers

- What happens when pointers are passed across boundaries?
 - Passing pointers alone insufficient when caller and callee are in two different address spaces
 - Need to copying the data referenced by the pointer passed
 - We use **deep copying**: passing pointers to structures and reachable substructures
 - **Problem**: Pointers may reference data or fields with ambiguous sizes
 - Is an `int*` pointer referencing a single integer or an array?
 - How large is a `char *` buffer referenced?
 - Limitations
 - C-style pointers do not carry bounds information
 - Do not know the sizes of the underlying buffers

Our Work: PtrSplit

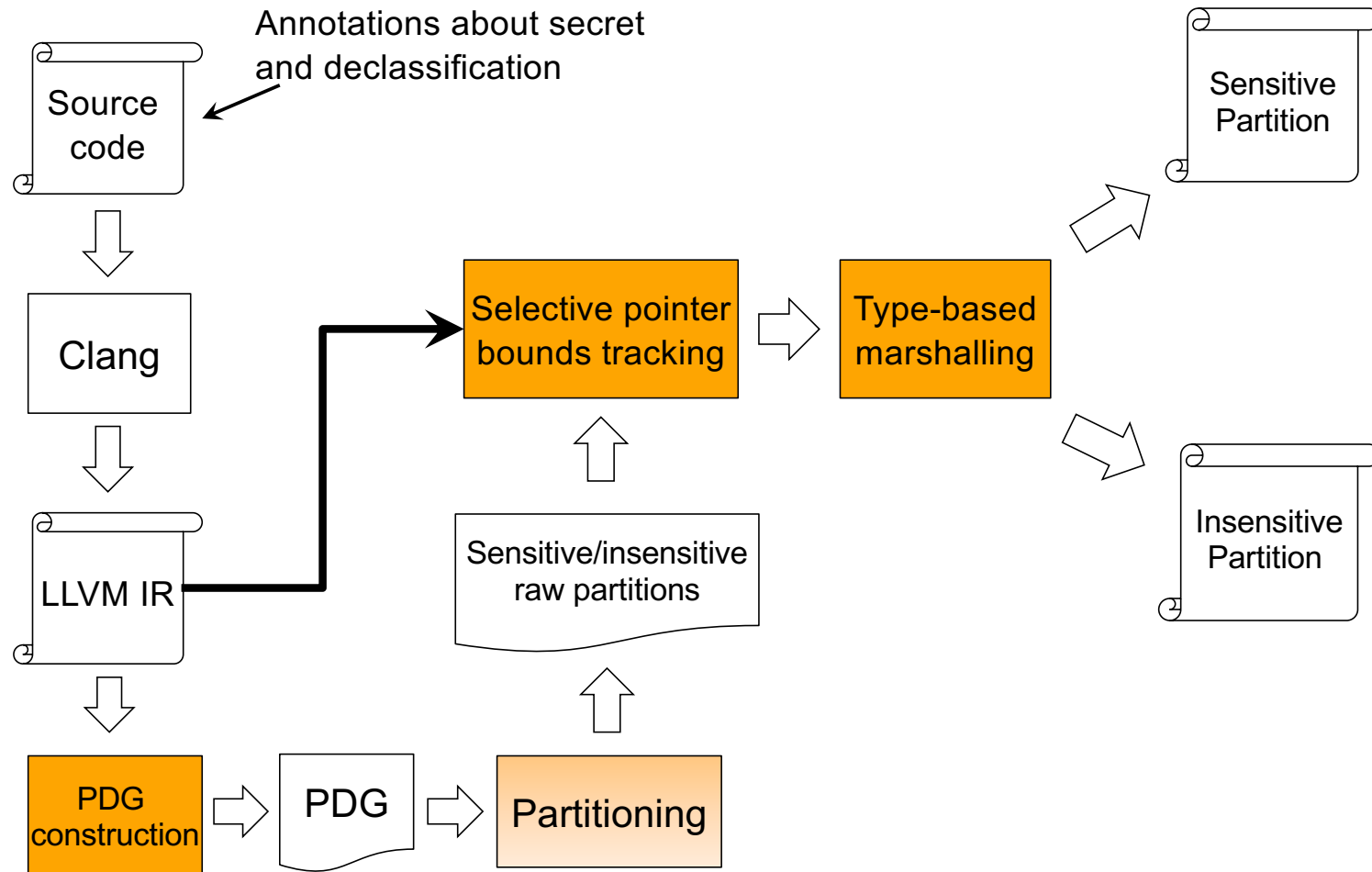
- PtrSplit provides automatic support for program partitioning with pointers
 - Perform program partitioning based on Program Dependence Graphs (PDG), which tracks control and data dependence
- **Parameter-tree**-based PDG
 - Avoid global pointer analysis
 - Modular construction of program dependence graphs by function
 - Determine all the functions needed to be included in a partition to avoid leakage/tampering
- Automated marshalling/unmarshalling for cross-boundary data, even with pointers
 - **Selective pointer bounds tracking**: track bounds only for necessary pointers
 - Avoid high overhead
 - **Type-based marshalling/unmarshalling**: use bounds information to perform deep copying

A Parameter-tree-based PDG



Program Dependency Graph for 'Test' function

Basic Workflow



Program Dependence Graph (PDG) Construction

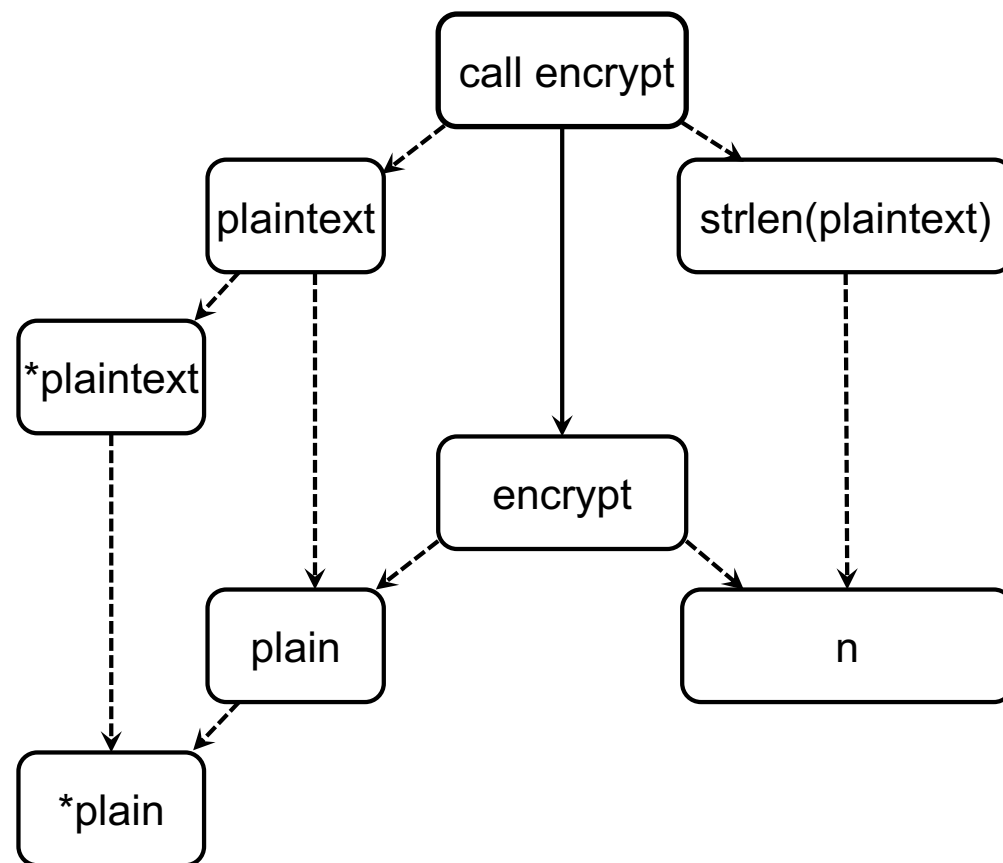
- We build a **parameter-tree**-based PDG
 - Represent a program's data and control dependence in a single graph
 - Sound representation of a program's control/data dependence
 - Modular construction through parameter trees

Parameter Tree: Example

```
char* cipher;  
char* key;
```

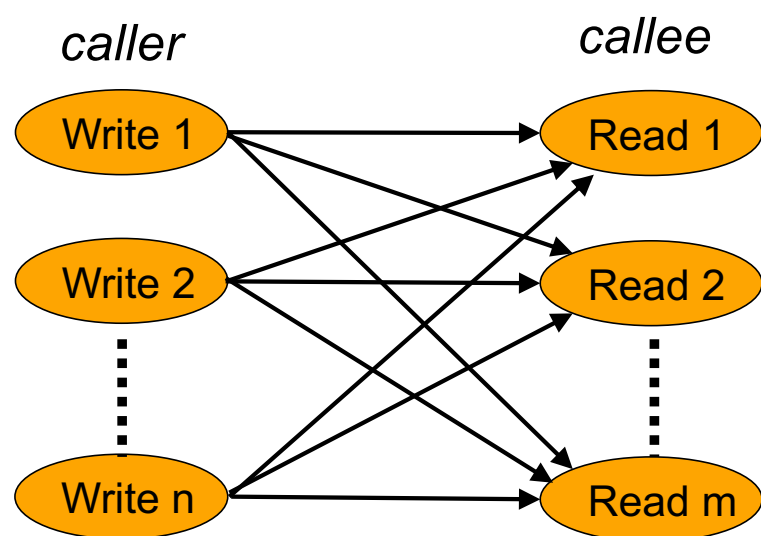
```
void encrypt(char *plain, int n){  
    cipher = (char*)malloc(n);  
    for (i = 0; i < n; i++)  
        cipher[i] = plain[i] ^ key[i];  
}
```

```
void main (){  
    char plaintext[1024];  
    scanf("%s",plaintext);  
    encrypt(plaintext,strlen(plaintext));  
    ...  
}
```

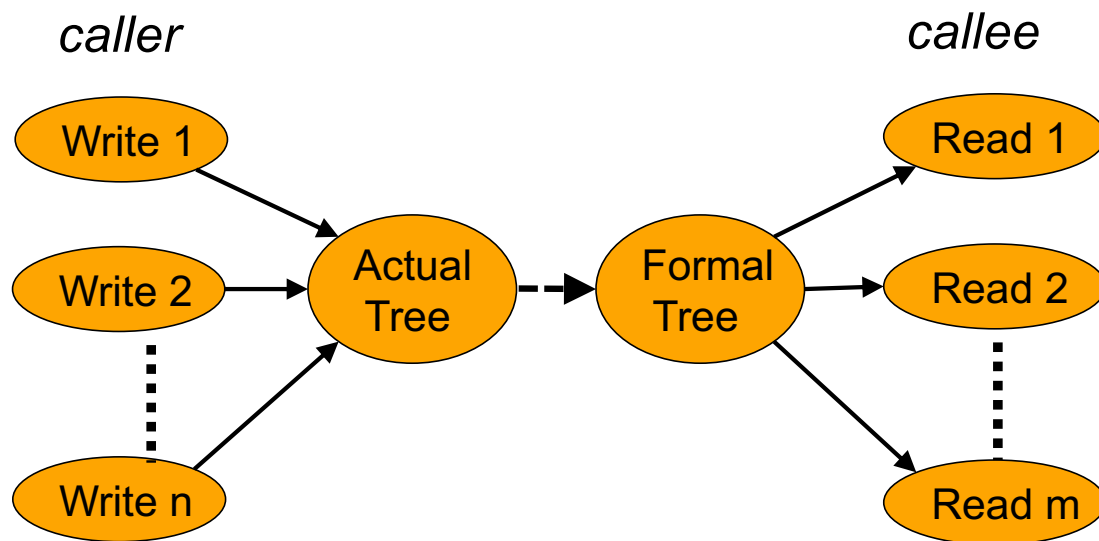


Benefits of Parameter Trees

- Avoid global pointer analysis
 - only intra-procedural pointers analysis is needed
- Reduce the number of dependence edges: suppose n writes and m reads



No parameter trees: $O(n*m)$ edges



With parameter tree: $O(n+m)$ edges

PDG-based Partitioning

- After the PDG construction, we perform PDG-based partitioning
- **Input:** sensitive and declassification nodes
- **Output:** two partitions
 - each partition is a set of functions and global variables
- **Potential problem:** only raw partitions can be generated
 - Inter-module communication overhead may be huge...
 - e.g. If we partition a program with 1000 functions into two, we may get a partition with 600 functions and another partition with 400 functions
 - May be many interactions between the two sets of functions

Leakage (Indirectly)

```
char* cipher;
```

```
char* key;
```

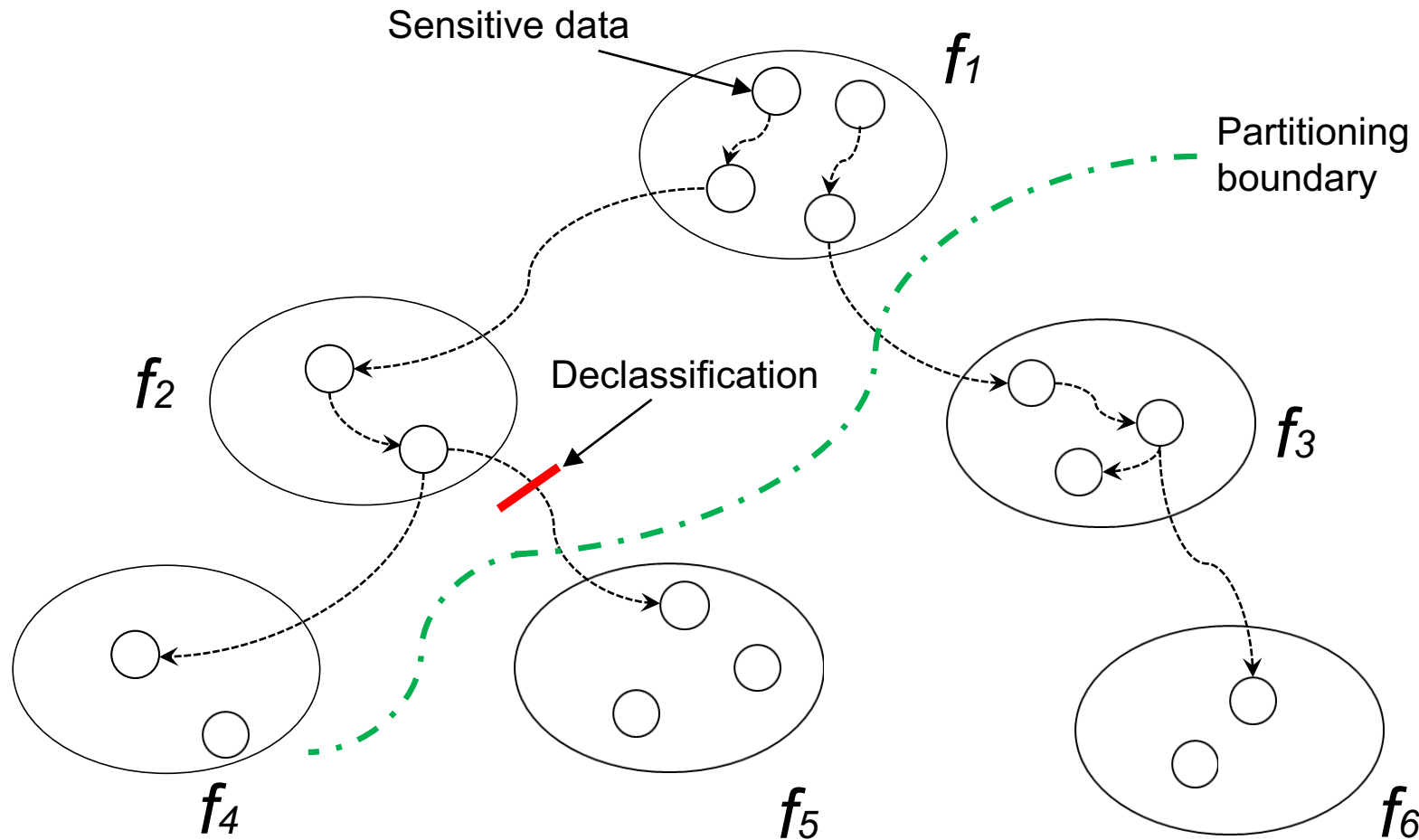
Sensitive data

```
void encrypt(char *plain, int n){  
    cipher =(char*)malloc(n);  
    for (i = 0; i < n; i++)  
        cipher[i] = plain[i] ^ key[i];  
}
```

```
void main () {  
    char plaintext[1024];  
    scanf("%s",plaintext);  
    encrypt(plaintext,strlen(plaintext));  
    ...  
}
```

Buffer overflow

PDG-based Partitioning: Example



Selective Pointer Bounds Tracking

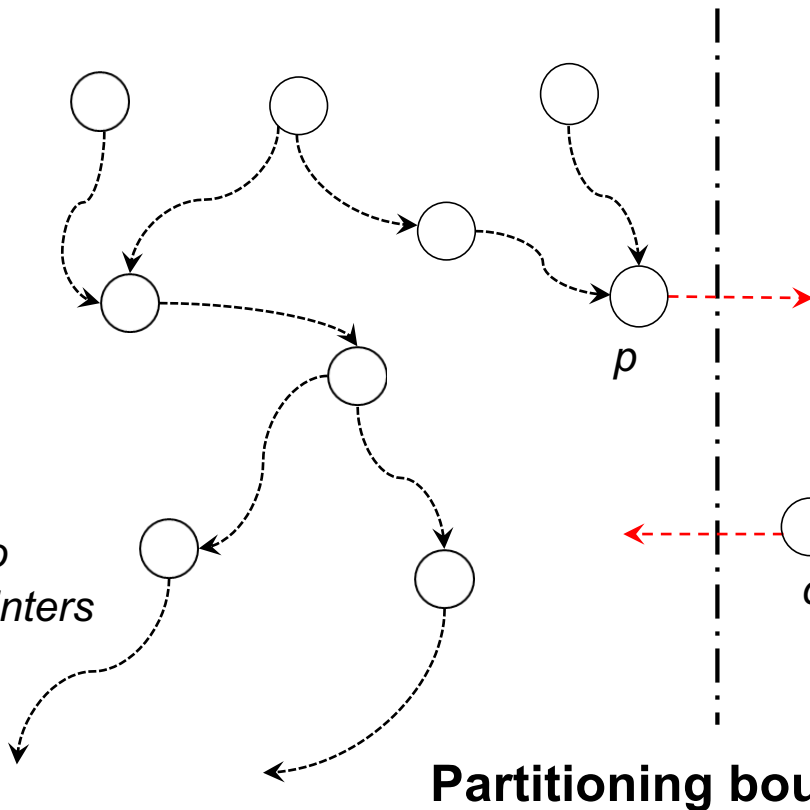
- Why we need to know the buffer size?
 - When pointers are passed across the partition boundary, we deep copy pointers and their underlying buffers
- How to calculate the buffer size?
 - Use bounds tracking tools
- Several tools for enforcing memory safety track bounds at runtime
- However, enforcing memory safety incurs high performance overhead
 - E.g., SoftBound's performance overhead on the SPEC and Olden benchmarks is 67% on average
- Improvement
 - For marshalling and unmarshalling it is necessary to perform **only bounds tracking, but not bounds checking**
 - We care about only the bounds of pointers that can **cross the boundary** of partitions

Selective Pointer Bounds Tracking

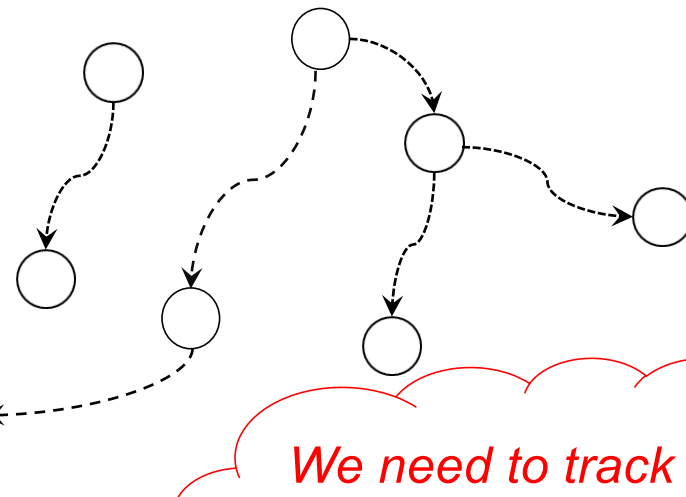
Insensitive Partition

Step 1
Find pointers that are sent across the boundary

Step 2
Do backward propagation to find all BR pointers



Sensitive Partition



We need to track the bounds of only the labeled pointers

Automatic Support of Marshalling and Unmarshalling

- Since partitions are loaded into separate processes, some function calls are turned into Remote Procedure Calls (RPCs)
 - Straightforward for values of most data types, including integers, arrays of fixed sizes, and structs
 - For pointers, the underlying buffer sizes can be tracked with SPBT
- When a pointer is passed across the boundary, we perform deep copying
 - After marshalling, arguments of a function call are encoded as a byte array, which is sent to the receiver via the help of an RPC library

Experiments

- We implemented PtrSplit on LLVM 3.5, which supports both DSA alias analysis and SoftBound
 - SoftBound keeps the bound information as metadata for each pointer
 - All bounds checking operations removed
 - Only BR-pointers are instrumented
 - RPC library: TI-RPC
- Robustness testing
 - 8 benchmarks from SPECCPU 2006
- Security testing
 - 4 security-sensitive programs

Example: tthttpd

- Sensitive data: authentication file
- Declassification: the return result (integer) of function auth_check
- Full pointer bounds tracking overhead : 56.3%
 - Selective pointer bounds tracking overhead: 3.6%
- A total of 5 out of 145 functions are marked sensitive
 - Total overhead: 8.8%

Result: Security-sensitive Programs

Program	Sensitive Data	Declassifications	Total Functions	Sensitive Functions
ssh	Private key file	2	1235	12
wget	Downloaded file	2	666	8
thttpd	Authentication file	1	145	5
telnet	Received data from server	3	180	11

Program	Total/BR pointers	Full PBT overhead	Selective PBT overhead	Total overhead
ssh	21020/591	45.0%	2.6%	7.4%
wget	14939/466	52.5%	3.4%	6.5%
thttpd	3068/189	56.3%	3.6%	8.8%
telnet	2068/233	74.1%	5.1%	9.6%

Selective bounds tracking greatly reduced overhead

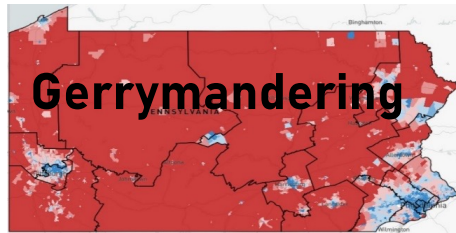
Experiments: SPECCPU 2006 programs

- Not suitable for security experiments, only used for correctness testing
- Use randomly chosen data as the partitioning start
- Average full pointer bounds tracking overhead : 136.2%
 - Average selective pointer bounds tracking overhead: 7.2%
- Average total overhead: 33.8%

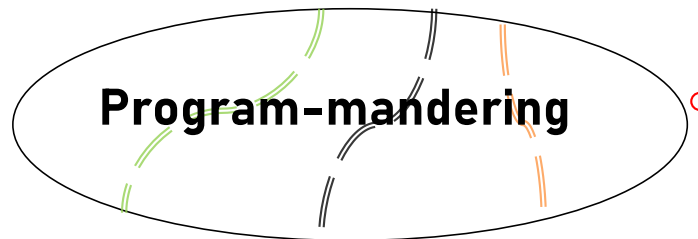
Balance Security and Performance: Program-mandering (PM)

- Program-mandering

- A **quantitative** framework that takes user guidance about how to balance between performance and security and computes partitioning boundaries

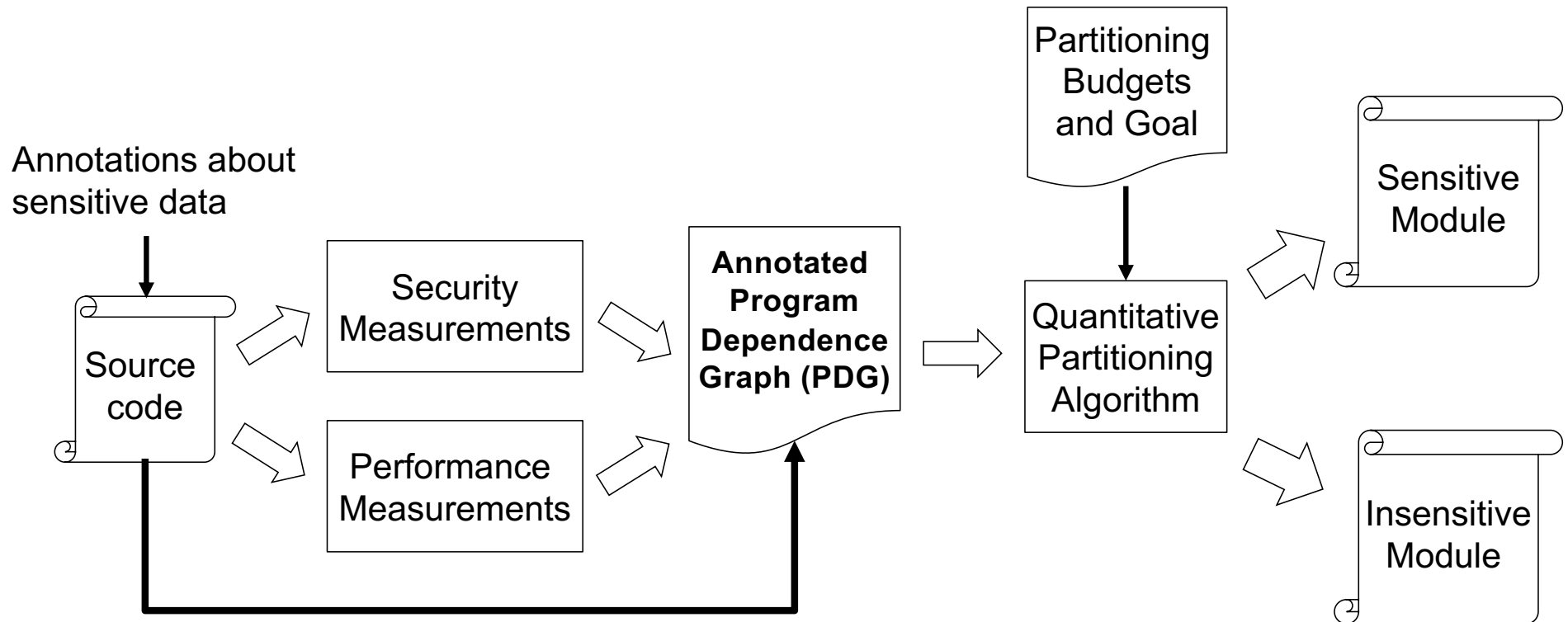


https://pic1.zhimg.com/v2-c1b58313a3b8973fc3b1ce2ff874ae2c_1200x500.jpg



*We can manipulate
the boundary for the good!*

PM System Flow



PM Overview

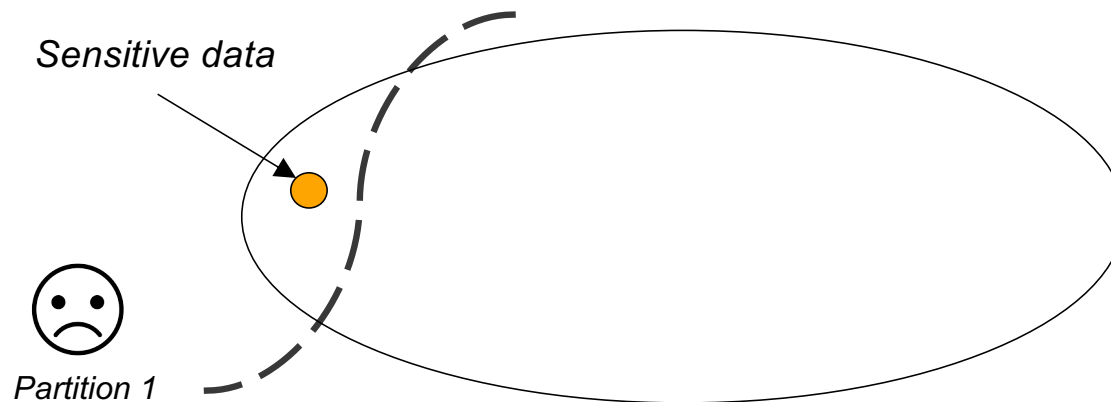
- Propose a set of metrics for security and performance
 - Implement program analysis to automatically collect measurements on a program
- Users specify performance/security budgets and an optimization goal
 - E.g., at most 10 context switches per second and find the partition with the smallest sensitive domain
- Convert the problem of “partitioning a program” into **“an Integer Programming (IP) problem”**
- Use an IP solver to find the optimal partition that satisfies user constraints

Program Partitioning as an Optimization

- User specification
 - **Budgets** (b_c, b_f, b_s, b_x) on sensitive code percentage, the amount of sensitive info flow, context switch frequency, and pointer complexity
 - Unlimited budgets are allowed with “_”
 - **Optimization goal**: which dimension to minimize
 - E.g., (10%, 2*, _, _)
- Conversion to integer programming
 - Encode the annotated PDG, the budgets, and the optimization goal as an integer programming problem
 - Use an IP solver to get the optimal solution

PM: an Interactive Tool

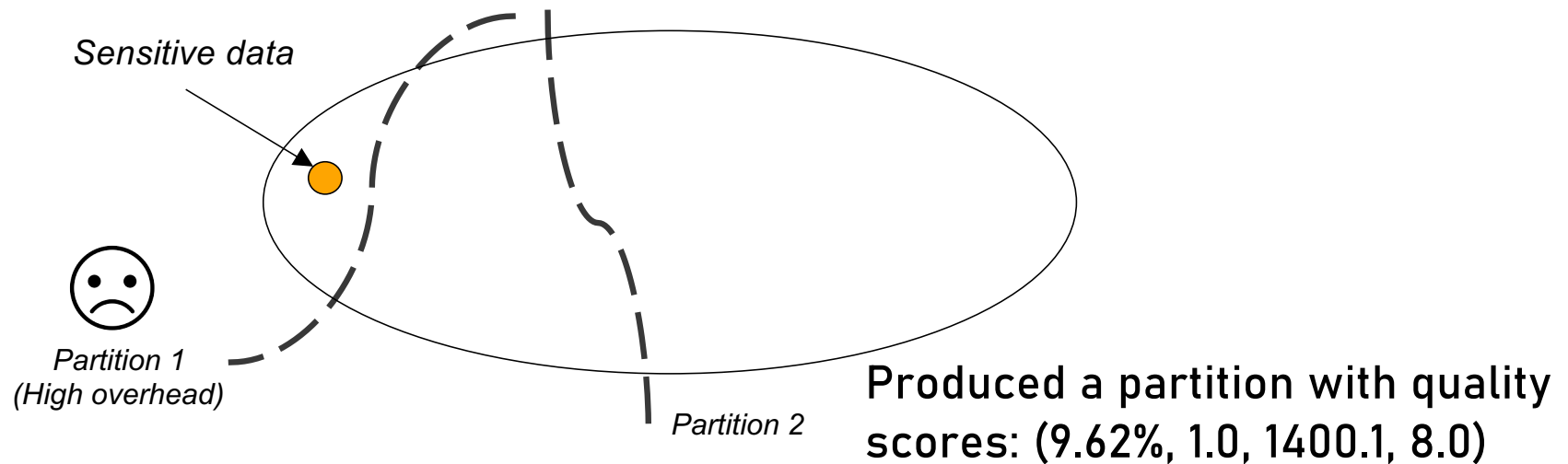
- Start with unlimited budgets and only minimize the sensitive code percentage: ($_*$, $_$, $_$, $_$)



For tthttpd with authentication info as sensitive data, this produced a partition with quality scores: (9.15%, 1.0, **1455.6**, 9.0); **high overhead**

PM: an Interactive Tool

- Partition 1's quality score: (9.15%, 1.0, 1455.6, 9.0)
- New budgets: (10%, 1.0, 1455.5*, 9.0)
 - Decrease the budget on the context-switch frequency and aim to minimize it
 - Increase the budget on sensitive code percentage to 10%



Q&A

Thank you!