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**

```c
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));
    ...
}
```

- **Sensitive data**: `cipher` and `key`
- **Buffer overflow**: `scanf` without ensuring the array is large enough to hold the input.
Toy Example

```c
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));
    ...
}
```

The sensitive data is protected!
Manual partitioning
- do code review and extract the sensitive components
- The amount of code for analysis may be huge…

Automatic partitioning
- Given some security criterions, do partitioning based on static program analysis
- Reduce manual effort and errors
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 to.

```c
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));
    ...
}
```
Privtrans automatically incorporate privilege separation into source code by partitioning it into two programs
- A **monitor** program which handles privileged operations
- A **slave** 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 monitor and slave is implemented by Remote Procedure Call (RPC)
RPC allows a program to call procedures that run in a different address space

- Programmers need to tell RPC what functions will be called remotely, and define the interfaces(IDL file)
- RPC can generate code to transmit data between the client and servers
- Data transmission is done through the network

Background: Remote Procedure Call(RPC)

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

- Systems for automatic program partitioning
  - Privman by Kilpatrick (USENIX ATC 2003)
  - Ptrimtrans 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)

- One major limitation: lack automatic support for pointers
  - Pointers prevalent in C/C++ applications
  - Previous work
    - Lack sound reasoning of pointers for partitioning
    - Require manual intervention when pointers are passed across partition boundaries
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 will be a disaster (e.g. Linux kernel)
Claim: For sound program partitioning, has to reason about program dependence with aliasing
– Need global pointer analysis for tracking dependence on programs with pointers
– Global pointer analysis is complex and unscalable

What happens when pointers are passed across boundaries?
– Passing pointers alone insufficient when caller and callee are in two different address spaces
– We use deep copying: passing pointers as well as their underlying buffers
  • However, 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 track program dependences
- **Parameter-tree**-based PDG
  - Avoid global pointer analysis
  - Modular construction of the dependence graph
- 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 marshaling/unmarshalling**: use bounds information to perform deep copying
**Background: Program Dependence Graph (PDG)**

- **PDG** is a **graphical representation** of the program
  - Program statements are represented as "nodes"
  - The dependencies among different statements are represented as "edges"

- In a PDG there exist two kinds of dependence
  - **Control dependence** describes the control relationships caused by conditional statements (if-else/switch) and circular statements (for/while loops)
  - **Data dependence** describes the relationship caused by assignment statements
void sum{
    int sum = 0;
    int i = 1;
    while (i < 10) {
        sum = sum + i;
        i = i + 1;
    }
}
A Parameter-tree-based PDG

Once we have such a graph, it’s easy to apply many graph-based algorithms…
Basic Workflow

Annotations about secret and declassification

Source code

Clang

LLVM IR

PDG construction → PDG → Partitioning

Selective pointer bounds tracking

Type-based marshalling

Sensitive/insensitive raw partitions

Sensitive Partition

Insensitive Partition
<table>
<thead>
<tr>
<th>Program Dependence Graph (PDG) Construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>- We build a <strong>parameter-tree</strong>-based PDG</td>
</tr>
<tr>
<td>- Represent a program’s data and control dependence in a single graph</td>
</tr>
<tr>
<td>- Sound representation of a program’s control/data dependence</td>
</tr>
<tr>
<td>- Modular construction through parameter trees</td>
</tr>
</tbody>
</table>
Motivation of Parameter Trees

- Pointers make building dependence graphs hard
- Inter-procedural dependences require global pointer analysis
- However, global pointer analysis is complex and unscalable

```c
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));
    ...  
}```
Parameter Trees

- Goal: make the PDG construction efficient and sound
  - For each parameter of a function, we build a formal parameter tree according to the parameter’s type
  - Similarly, at a call site of a function, we build a parameter tree for every argument
  - A caller and its callee can be connected by connecting the corresponding nodes in the actual and formal parameter trees

- Our tree representation generalizes the object-tree approach and deals with circular data structures resulting from pointers
  - Prior work did not cover pointers at the language level
```c
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
PDG-based partitioning may give us a very awkward result
– e.g. a sort function inside a 3-level loop is called remotely

To balance the security and performance, we use declassification to prevent some sensitive dataflows

Example:  

```c
bool authenticate(char* s1, char* s2){...
...
for(...) {
    if(authenticate(password,input) == true){...
}
}
```

(We can declassify authenticate’s return value since there isn’t too much sensitive information leakage here – should limit number of calls to authenticate)
PDG-based Partitioning: Example

Sensitive data

Partitioning boundary

Declassification
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

Step 1
Find pointers that are sent across the boundary

Step 2
Do backward propagation to find all BR pointers

We need to track the bounds of only the colored pointers
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
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 SPECCPU2006

**Security testing**
- 4 security-sensitive programs
Example: thttpd

- 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

<table>
<thead>
<tr>
<th>Program</th>
<th>Sensitive Data</th>
<th>Declassifications</th>
<th>Total Functions</th>
<th>Sensitive Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssh</td>
<td>Private key file</td>
<td>2</td>
<td>1235</td>
<td>12</td>
</tr>
<tr>
<td>wget</td>
<td>Downloaded file</td>
<td>2</td>
<td>666</td>
<td>8</td>
</tr>
<tr>
<td>thttpd</td>
<td>Authentication file</td>
<td>1</td>
<td>145</td>
<td>5</td>
</tr>
<tr>
<td>telnet</td>
<td>Received data from server</td>
<td>3</td>
<td>180</td>
<td>11</td>
</tr>
</tbody>
</table>

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

**Selective bounds tacking 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%
Future Work

- Multi-threading support

- More efficient bounds-tracking
  - LowFat Pointer (NDSS 2017).
  - Checked C (still in development)

- Automatic inference of sensitive data and declassifications
  - Automating Security Mediation Placement (ESOP 2010).
Thank you!