Effective Memory Protection Using Dynamic Tainting

James Clause
Alessandro Orso
(software)

and

Ioannis Doudalis
Milos Prvulovic
(hardware)

College of Computing
Georgia Institute of Technology

Supported in part by:
NSF awards CCF-0541080 and CCR-0205422 to Georgia Tech,
DHS and US Air Force Contract No. FA8750-05-2-0214
Index

- Conference & Authors
- Motivation
- Technique
- Implementation
- Evaluation
- References
The *International Conference on Automated Software Engineering (ASE)* brings together researchers and practitioners to share ideas on the foundations, techniques, tools, and applications of automated software engineering.
James Clause

- Ph.D. student at the Georgia Institute of Technology.
- Paper
  
  **LEAKPOINT: Pinpointing the Causes of Memory Leaks**
  International Conference on Software Engineering (ICSE 2010)

  **Penumbra: Automatically Identifying Failure-Relevant Inputs Using Dynamic Tainting**
  International Symposium on Software Testing and Analysis (ISSTA 2009)

  **Effective Memory Protection Using Dynamic Tainting**
  International Conference on Automated Software Engineering (ASE 2007)
Ioannis Doudalis

- Ph.D. student at the Georgia Institute of Technology.

Paper

**HARE: Hardware Assisted Reverse Execution.**

**MemTracker: An Accelerator for Memory Debugging and Monitoring**
ACM Transactions on Architecture and Code Optimization (TACO), June 2009

**FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation**

**Effective Memory Protection Using Dynamic Tainting**
International Conference on Automated Software Engineering (ASE 2007)
Conference & Authors

Alex Orso

• Associate Professor at the Georgia Institute of Technology.
  A member of the ACM and the IEEE Computer Society

• Paper
  
  "**Leakpoint: Pinpointing the Causes of Memory Leaks**" (ICSE 2010).
  "**Automated Behavioral Regression Testing**" (ICST 2010).
  "**PENUMBRA: Automatically Identifying Failure-Relevant Inputs Using Dynamic Tainting**" (ISSTA 2009).
  "**Precise Interface Identification to Improve Testing and Analysis of Web Applications**" (ACM Distinguished Paper Award at ISSTA 2009).
  "**Using Positive Tainting and Syntax-Aware Evaluation to Protect Web Applications**" (FSE 2006).

......
Conference & Authors

Milos Prvulovic

- Associate Professor at the Georgia Institute of Technology.
- Paper


void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7. *(buf + i) = rand()%10;
   ...
}

Illegal memory accesses (IMA)
void main() {
    1. int *np, n, i, *buf;

    2. np = &n;

    3. printf("Enter size: ");
    4. scanf("%d", np);

    5. buf = malloc(n * sizeof(int));

    6. for(i = 0; i <= n; i++)
    7.     *(buf + i) = rand() % 10;
    ...
}

Illegal memory accesses (IMA)
void main() {
    1. int *np, n, i, *buf;
    2. np = &n;
    3. printf("Enter size: ");
    4. scanf("%d", np);
    5. buf = malloc(n * sizeof(int));
    6. for(i = 0; i <= n; i++)
    7.     *(buf + i) = rand()%10;
    ...
}
```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand() % 10;

    ...
}
```
void main() {
1. int *np, n, i, *buf;
2. np = &n;
3. printf(“Enter size: “);
4. scanf(“%d”, np);
5. buf = malloc(n * sizeof(int));
6. for(i = 0; i <= n; i++)
7. *(buf + i) = rand()%10;
...}

Illegal memory accesses (IMA)
void main() {
  1. int *np, n, i, *buf;

  2. np = &n;

  3. printf("Enter size: ");
  4. scanf("%d", np);

  5. buf = malloc(n * sizeof(int));

  6. for(i = 0; i <= n; i++)
  7.   *(buf + i) = rand()%10;
      ...
}
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;
...
void main() {
1. int *np, n, i, *buf;
2. np = &n;
3. printf("Enter size: ");
4. scanf("%d", np);
5. buf = malloc(n * sizeof(int));
6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;
...
void main() {
1. int *np, n, i, *buf;
2. np = &n;
3. printf("Enter size: ");
4. scanf("%d", np);
5. buf = malloc(n * sizeof(int));
6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand() % 10;
... 
}
```c
void main() {
1. int *np, n, i, *buf;
2. np = &n;
3. printf("Enter size: ");
4. scanf("%d", np);
5. buf = malloc(n * sizeof(int));
6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand() % 10;
... 
}
```
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf(“Enter size: “);
4. scanf(“%d”, np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;

...}

Illegal memory accesses (IMA)
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand() % 10;

...}

Illegal memory accesses (IMA)
void main() {
  1. int *np, n, i, *buf;
  2. np = &n;
  3. printf("Enter size: ");
  4. scanf("%d", np);
  5. buf = malloc(n * sizeof(int));
  6. for(i = 0; i <= n; i++)
  7.   *(buf + i) = rand()%10;
...}

Illegal memory accesses (IMA)

Memory

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i: 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>n: 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>np:</td>
<td></td>
</tr>
</tbody>
</table>
void main() {
  1. int *np, n, i, *buf;
  2. np = &n;
  3. printf(“Enter size: “);
  4. scanf(“%d”, np);
  5. buf = malloc(n * sizeof(int));
  6. for (i = 0; i <= n; i++)
  7.     *(buf + i) = rand() % 10;
  ...
}

Illegal memory accesses (IMA)

Memory

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>buf:</td>
<td>i: 2</td>
<td>n: 3</td>
<td>np:</td>
<td></td>
</tr>
</tbody>
</table>

i <= n → i < n
void main() {
1. int *np, n, i, *buf;
2. np = &n;
3. printf("Enter size: ");
4. scanf("%d", np);
5. buf = malloc(n * sizeof(int));
6. for(i = 0; i <= n; i++)
7. *(buf + i) = rand()%10;
...}

Illegal memory accesses (IMA)

Memory

buf:

i: 2

n: 3

np:
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand() % 10;
...
}

Illegal memory accesses (IMA)

Memory

<table>
<thead>
<tr>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

buf:

i:

n:

np:
void main() {
  1. int *np, n, i, *buf;

  2. np = &n;

  3. printf("Enter size: ");
  4. scanf("%d", np);

  5. buf = malloc(n * sizeof(int));

  6. for(i = 0; i <= n; i++)
  7.   *(buf + i) = rand()%10;

  ...
}

Illegal memory accesses (IMA)

Memory
void main() {
    1. int *np, n, i, *buf;

    2. np = &n;

    3. printf("Enter size: ");
    4. scanf("%d", np);

    5. buf = malloc(n * sizeof(int));

    6. for(i = 0; i <= n; i++)
    7.  *(buf + i) = rand() % 10;

... }

Illegal memory accesses (IMA)

Memory

- np: 9
- i: 3
- n: 3
- buf:
void main() {
1. int *np, n, i, *buf;
2. np = &n;
3. printf("Enter size: ");
4. scanf("%d", np);
5. buf = malloc(n * sizeof(int));
6. for (i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;
8. ...}

Illegal memory accesses (IMA)

- Caused by common programming mistakes
- Cause non-deterministic failures
- Cause security vulnerabilities

Illegal memory accesses

Memory

| 9 |

buf:

np:

i:

n:
Previous work

Static techniques

• Language based
  e.g., Jim et al. 02, Necula et al. 05

• Analysis based
  e.g., Dor et al. 03, Hallem et al. 02, Heine and Lam 03, Xie et al. 03

Dynamic techniques

• Analysis based
  e.g., Dhurjati and Adve 06, Ruwase and Lam 04, Xu et al. 04, Hastings and Joyce 92, Seward and Nethercote 05

• Hardware based
  e.g., Qin et al. 05, Venkataramani et al. 07, Crandall and Chong 04, Dalton et al. 07, Vachharajani et al. 04
Previous work

Static techniques

• Language based
e.g., Jim et al. 02, Necula et al. 05

• Analysis based
e.g., Dor et al. 03, Hallem et al. 02, Heine and Lam 03, Xie et al. 03

Dynamic techniques

• Analysis based
e.g., Dhurjati and Adve 06, Ruwase and Lam 04, Xu et al. 04, Hastings and Joyce 92, Seward and Nethercote 05

• Hardware based
e.g., Qin et al. 05, Venkataramani et al. 07, Crandall and Chong 04, Dalton et al. 07, Vachharajani et al. 04
Previous work

Static techniques

• Language based
e.g., Jim et al. 02, Necula et al. 05

• Analysis based
e.g., Dor et al. 03, Hallem et al. 02, Heine and Lam 03, Xie et al. 03

Dynamic techniques

• Analysis based
e.g., Dhurjati and Adve 06, Ruwase and Lam 04, Xu et al. 04, Hastings and Joyce 92, Seward and Nethercote 05

• Hardware based
e.g., Qin et al. 05, Venkataramani et al. 07, Crandall and Chong 04, Dalton et al. 07, Vachharajani et al. 04

Require source code

Unacceptable overhead
Previous work

Static techniques

• Language based
e.g., Jim et al. 02, Necula et al. 05

• Analysis based
e.g., Dor et al. 03, Hallem et al. 02, Heine and Lam 03, Xie et al. 03

Dynamic techniques

• Analysis based
e.g., Dhurjati and Adve 06, Ruwase and Lam 04, Xu et al. 04, Hastings and Joyce 92, Seward and Nethercote 05

• Hardware based
e.g., Qin et al. 05, Venkataramani et al. 07, Crandall and Chong 04, Dalton et al. 07, Vachharajani et al. 04

Require source code

Unacceptable overhead

Extensive modification
Previous work

Static techniques

- Language based (e.g., Jim et al. 02, Necula et al. 05)
- Analysis based (e.g., Dor et al. 03, Hallem et al. 02, Heine and Lam 03, Xie et al. 03)

Dynamic techniques

- Analysis based (e.g., Dhurjati and Adve 06, Ruwase and Lam 04, Xu et al. 04, Hastings and Joyce 92, Seward and Nethercote 05)
- Hardware based (e.g., Qin et al. 05, Venkataramani et al. 07, Crandall and Chong 04, Dalton et al. 07, Vachharajani et al. 04)

We define our approach to overcome these limitations:

- Operate at the binary level
- Use hardware to reduce overhead
- Minimal, practical modifications

We define our approach to overcome these limitations

- Operate at the binary level
- Use hardware to reduce overhead
- Minimal, practical modifications
Approach overview
Approach overview

1. Assign taint marks
Approach overview

1. Assign taint marks

Diagram:
- P₁
- A

1. Assign taint marks
Approach overview

1. Assign taint marks
Approach overview

1. Assign taint marks
1. Assign taint marks

Approach overview
Approach overview

1. Assign taint marks
Approach overview

1. Assign taint marks
2. Propagate taint marks
1. Assign taint marks

2. Propagate taint marks

Approach overview
Approach overview

1. Assign taint marks
2. Propagate taint marks
Approach overview

1. Assign taint marks
2. Propagate taint marks
3. Check taint marks
1. Assign taint marks
2. Propagate taint marks
3. Check taint marks
Approach overview

1. Assign taint marks
2. Propagate taint marks
3. Check taint marks
Approach overview

1. Assign taint marks
2. Propagate taint marks
3. Check taint marks
Approach overview

1. Assign taint marks

2. Propagate taint marks

3. Check taint marks
1. Assign taint marks
2. Propagate taint marks
3. Check taint marks
Approach overview

1. Assign taint marks
2. Propagate taint marks
3. Check taint marks
Outline

• Our approach
  1. Assigning taint marks
  2. Propagating taint marks
  3. Checking taint marks
• Empirical evaluation
• Conclusions
Assigning taint marks

1. Static memory allocations
   - Identify the ranges of allocated memory
   - Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
}
```

2. Pointers to statically allocated memory
   - Identify pointer creation sites
   - Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
}
```

3. Dynamic memory allocations
   - Identify the ranges of allocated memory
   - Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
}
```

4. Pointers to dynamically allocated memory
   - Identify pointer creation sites
   - Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
}
```
void main() {
    int *np, n, i, *buf;

    np = &n;
    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
void main() {
    int *np, n, i, *buf;

    np = &n;
    printf(“Enter size: “);
    scanf(“%d”, np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand() % 10;
    ...
}
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
Assigning taint marks

1. **Assigning taint marks**

   **Static memory allocations**
   1. Identify the ranges of allocated memory
   2. Assign a unique taint mark to each range

   ```c
   void main() {
     int *np, n, i, *buf;
     np = &n;
     printf("Enter size: ");
     scanf("%d", np);
     buf = malloc(n * sizeof(int));
     for(i = 0; i <= n; i++)
       *(buf + i) = rand()%10;
     ...
   }
   ``

   **Dynamic memory allocations**
   1. Identify the ranges of allocated memory
   2. Assign a unique taint mark to each range

   ```c
   void main() {
     int *np, n, i, *buf;
     np = &n;
     printf("Enter size: ");
     scanf("%d", np);
     buf = malloc(n * sizeof(int));
     for(i = 0; i <= n; i++)
       *(buf + i) = rand()%10;
     ...
   }
   ```

   **Pointers to statically allocated memory**
   1. Identify pointer creation sites
   2. Assign the pointer the same taint mark as the memory it points to

   ```c
   void main() {
     int *np, n, i, *buf;
     np = &n;
     printf("Enter size: ");
     scanf("%d", np);
     buf = malloc(n * sizeof(int));
     for(i = 0; i <= n; i++)
       *(buf + i) = rand()%10;
     ...
   }
   ``

   **Pointers to dynamically allocated memory**
   1. Identify pointer creation sites
   2. Assign the pointer the same taint mark as the memory it points to

   ```c
   void main() {
     int *np, n, i, *buf;
     np = &n;
     printf("Enter size: ");
     scanf("%d", np);
     buf = malloc(n * sizeof(int));
     for(i = 0; i <= n; i++)
       *(buf + i) = rand()%10;
     ...
   }
   ```
Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand() % 10;
    ...
}
```
Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int n, i, *np, *buf;
    np = &n;

    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...}
```
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
...
Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
...
}
```
Pointers to statically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```
Assigning taint marks

Static memory allocations
1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```

Dynamic memory allocations
1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```

Pointers to statically allocated memory
1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```

Pointers to dynamically allocated memory
1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```
Dynamic memory allocations

1. Identify the ranges of allocated memory

2. Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```
Dynamic memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;

    ...
}
```
Dynamic memory allocations

1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf([ret, ret + arg0]
        buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```
Dynamic memory allocations

1. Identify the ranges of allocated memory

2. Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```
Dynamic memory allocations

1. Identify the ranges of allocated memory

2. Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```
1 Assigning taint marks

**Static memory allocations**
1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```

**Pointers to statically allocated memory**
1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```

**Dynamic memory allocations**
1. Identify the ranges of allocated memory
2. Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```

**Pointers to dynamically allocated memory**
1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```
Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```

- `malloc` returns the address of the allocated memory block. This address can be used as a pointer.
- When allocating memory, use `malloc` to obtain the address of the allocated block.
- When using the allocated memory, use the address returned by `malloc` to access its content.

Remember to free the memory using `free` when you are done with it to prevent leaks.
Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;

    ...
}
Pointers to dynamically allocated memory

1. Identify pointer creation sites
2. Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```
1 Assigning taint marks

Static memory allocations
1 Identify the ranges of allocated memory
2 Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; ++i)
        *(buf + i) = rand()%10;
    ...
}
```

Static memory allocations
1 Identify the ranges of allocated memory
2 Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; ++i)
        *(buf + i) = rand()%10;
    ...
}
```

Dynamic memory allocations
1 Identify the ranges of allocated memory
2 Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; ++i)
        *(buf + i) = rand()%10;
    ...
}
```

Dynamic memory allocations
1 Identify the ranges of allocated memory
2 Assign a unique taint mark to each range

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; ++i)
        *(buf + i) = rand()%10;
    ...
}
```

Pointers to statically allocated memory
1 Identify pointer creation sites
2 Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```

Pointers to statically allocated memory
1 Identify pointer creation sites
2 Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```

Pointers to dynamically allocated memory
1 Identify pointer creation sites
2 Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```

Pointers to dynamically allocated memory
1 Identify pointer creation sites
2 Assign the pointer the same taint mark as the memory it points to

```c
void main() {
    int *np, n, i, *buf;
    np = &n;
    printf("Enter size: ");
    scanf("%d", np);
    buf = malloc(n * sizeof(int));
    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
    ...
}
```
Propagating taint marks

Overview

Addition, Subtraction

AND

Multiplication, Division, OR, XOR
Propagating taint marks

Overview

Addition, Subtraction

AND

Multiplication, Division, OR, XOR

+, −, ×, ÷, and, or, xor, ...

P₁

1

P₂
Propagating taint marks

Overview

Addition, Subtraction
AND
Multiplication, Division, OR, XOR

Should the result be tainted? If so, how?
2 Propagating taint marks

Overview

Addition, Subtraction

AND

Multiplication, Division, OR, XOR

• Propagation must take into account both operation semantics and programmer intent

Should the result be tainted? If so, how?
**2 Propagating taint marks**

**Overview**

- Addition, Subtraction
- AND
- Multiplication, Division, OR, XOR

**Should the result be tainted? If so, how?**

- Propagation must take into account both operation semantics and programmer intent.
- Our policy is based on knowledge of C/C++/assembly and patterns observed in real software.
Most common use of addition and subtraction is to add or subtract a pointer and an offset.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>no taint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
2. Propagating taint marks

Overview

Addition, Subtraction

AND

Multiplication, Division, OR, XOR

**A & B = C**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1 or no taint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

The result of anding a pointer and a mask should be treated differently depending on the value of the mask.

\[
c = a \& 0xffffffff00 - \text{base address}
\]

\[
c = a \& 0x000000ff - \text{offset}
\]
Propagating taint marks

Overview

Addition, Subtraction

Multiplication, Division, OR, XOR

We found zero cases where the result of any of these operations was a pointer.
### Checking taint marks

When memory is accessed through a pointer: compare the memory taint mark and the pointer taint mark.

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Memory</th>
<th>IMA?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>yes</td>
</tr>
</tbody>
</table>
void main() {
    1. int *np, n, i, *buf;
    
    2. np = &n;
    
    3. printf("Enter size: ");
    4. scanf("%d", np);
    
    5. buf = malloc(n * sizeof(int));
    
    6. for(i = 0; i <= n; i++)
    7.    *(buf + i) = rand() % 10;
    ...
}
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;
   ...
}

Preventing IMAs
void main() {
    1. int *np, n, i, *buf;
    2. np = &n;
    3. printf("Enter size: ");
    4. scanf("%d", np);
    5. buf = malloc(n * sizeof(int));
    6. for(i = 0; i <= n; i++)
    7.    *(buf + i) = rand()%10;
    ...
}
### Preventing IMAs

```c
void main() {

1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand() % 10;
...
}
```
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand() % 10;
   ...
}

buf: 4
i: 3
n: 2
np: 1
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand() % 10;
... 
}
void main() {
    int *np, n, i, *buf;

    np = &n;

    printf("Enter size: ");
    scanf("%d", np);

    buf = malloc(n * sizeof(int));

    for(i = 0; i <= n; i++)
        *(buf + i) = rand()%10;
...
}

Preventing IMAs
Preventing IMAs

```c
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;
...
}
```
Preventing IMAs

```c
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand() % 10;
...
}
```
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand() % 10;
...
}
void main() {
  1. int *np, n, i, *buf;

  2. np = &n;

  3. printf("Enter size: ");
  4. scanf("%d", np);

  5. buf = malloc(n * sizeof(int));

  6. for(i = 0; i <= n; i++)
  7.   *(buf + i) = rand() % 10;
  ...
}
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;
...
}
Preventing IMAs

```c
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf(“Enter size: “);
4. scanf(“%d”, np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand() % 10;

...}
```
void main() {
1. int *np, n, i, *buf;
2. np = &n;
3. printf("Enter size: ");
4. scanf("%d", np);
5. buf = malloc(n * sizeof(int));
6. for(i = 0; i <= n; i++)
7. *(buf + i) = rand()%10;
...}

Preventing IMAs
void main() {
  1. int *np, n, i, *buf;

  2. np = &n;

  3. printf(“Enter size: “);
  4. scanf(“%d”, np);

  5. buf = malloc(n * sizeof(int));

  6. for(i = 0; i <= n; i++)
  7.   *(buf + i) = rand() % 10;

  ...}

void main() {
    1. int *np, n, i, *buf;

    2. np = &n;

    3. printf("Enter size: ");
    4. scanf("%d", np);

    5. buf = malloc(n * sizeof(int));

    6. for(i = 0; i <= n; i++)
    7.    *(buf + i) = rand() % 10;

}
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf(“Enter size: “);
4. scanf(“%d”, np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;

}

Preventing IMAs
void main() {
1. int *np, n, i, *buf;
2. np = &n;
3. printf("Enter size: ");
4. scanf("%d", np);
5. buf = malloc(n * sizeof(int));
6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;
...}

void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;

...}

Preventing IMAs
void main() {
    1. int *np, n, i, *buf;
    
    2. np = &n;
    
    3. printf("Enter size: ");
    4. scanf("%d", np);
    
    5. buf = malloc(n * sizeof(int));
    
    6. for(i = 0; i <= n; i++)
    7.     *(buf + i) = rand()%10;
    
    ...
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.   *(buf + i) = rand()%10;

...
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf(“Enter size: “);
4. scanf(“%d”, np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7. *(buf + i) = rand()%10;

...}
void main() {
1. int *np, n, i, *buf;

2. np = &n;

3. printf("Enter size: ");
4. scanf("%d", np);

5. buf = malloc(n * sizeof(int));

6. for(i = 0; i <= n; i++)
7.  *(buf + i) = rand()%10;

}
Limiting the number of taint marks

An unlimited number of taint marks makes a hardware implementation infeasible

- increases the overhead (time and space)
- complicates the design
Limiting the number of taint marks

An unlimited number of taint marks makes a hardware implementation infeasible

- increases the overhead (time and space)
- complicates the design

Assign taint marks from a limited, reusable pool
**Effects on the approach**

⚠️ IMAs are detected probabilistically

With an random assignment of $n$ taint marks the detection probability is:

$$p = 1 - \frac{1}{n}$$
Effects on the approach

⚠️ IMAs are detected probabilistically

With an random assignment of $n$ taint marks the detection probability is:

$$p = 1 - \frac{1}{n}$$

2 marks = 50%, 4 marks = 75%, 16 marks = 93.75%, 256 marks = 99.6%
Effects on the approach

⚠️ IMAs are detected probabilistically

With an random assignment of $n$ taint marks the detection probability is:

$$p = 1 - \frac{1}{n}$$

2 marks = 50%, 4 marks = 75%, 16 marks = 93.75%, 256 marks = 99.6%

1. The technique can be tuned by increasing or decreasing the number of taint marks
Effects on the approach

⚠️ IMAs are detected probabilistically

With an random assignment of \( n \) taint marks the detection probability is:

\[
p = 1 - \frac{1}{n}
\]

2 marks = 50%, 4 marks = 75%, 16 marks = 93.75%, 256 marks = 99.6%

1. The technique can be tuned by increasing or decreasing the number of taint marks

2. In practice the approach is successful with only a small number (2) of taint marks
Empirical evaluation

RQ1: Is the efficiency of our approach sufficient for it to be applied to deployed software?

RQ2: What is the effectiveness of our technique when using limited number of taint marks?
RQ1: experimental method

• Hardware implementation
  • Cycle accurate simulator (SESC)
  • Treat taint marks as first class citizens

• Subjects
  • SPEC CPU2000 benchmark (12 applications)

• Calculate the overhead imposed by our approach for each subject application
RQ1: experimental method

- Hardware implementation
  - Cycle accurate simulator (SESC)
  - Treat taint marks as first class citizens
- Subjects
  - SPEC CPU2000 benchmark (12 applications)
- Calculate the overhead imposed by our approach for each subject application

Current implementation assigns taint marks only to dynamically allocated memory, but propagation and checking are fully implemented.
RQ1: results

The chart shows the percentage overhead (time) for various benchmarks across different marks:

- **2 marks**
- **8 marks**
- **16 marks**
- **256 marks**

The benchmarks include:
- bzip2
- crafty
- eon
- gap
- gcc
- gzip
- mcf
- parser
- perlbench
- twolf
- vortex
- vpr
- average

The x-axis represents the benchmarks, and the y-axis shows the percentage overhead (time) in marks.
Even with 256 marks, the average overhead is in the single digits.
RQ1: results

- Even with 256 marks, the average overhead is in the single digits
- All attacks were detected with two taint marks
**RQ I: results**

- Even with 256 marks, the average overhead is in the single digits.
- All attacks were detected with two taint marks.
- Software-only implementations impose two orders of magnitude more overhead.
RQ2: experimental method

• Software implementation
  • Binary instrumenter (Pin)
  • Use instrumentation to assign, propagate, and check taint marks

• Subjects
  • SPEC CPU2000 benchmark (12 applications)
  • 5 applications with 7 known IMAs

• Run both each applications protected by our software implementation and check that only the known illegal memory accesses are detected (5 times)
### RQ2: results

#### Applications with known IMAs

<table>
<thead>
<tr>
<th>Application</th>
<th>IMA location</th>
<th>Type</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc-1.06</td>
<td>more_arrays: 177</td>
<td>buffer overflow</td>
<td>✓ (5/5)</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>lookup: 577</td>
<td>buffer overflow</td>
<td>✓ (5/5)</td>
</tr>
<tr>
<td>gnupg-1.4.4</td>
<td>parse_comment: 2095</td>
<td>integer overflow</td>
<td>✓ (5/5)</td>
</tr>
<tr>
<td>mutt-1.4.2.li</td>
<td>utf8_to_utf7: 199</td>
<td>buffer overflow</td>
<td>✓ (5/5)</td>
</tr>
<tr>
<td>php-5.2.0</td>
<td>php_char_to_str_ex: 3152</td>
<td>integer overflow</td>
<td>✓ (5/5)</td>
</tr>
<tr>
<td>pine-4.44</td>
<td>rfc882_cat: 260</td>
<td>buffer overflow</td>
<td>✓ (5/5)</td>
</tr>
<tr>
<td>squid-2.3</td>
<td>ftpBuildTitleUrl: 1024</td>
<td>buffer overflow</td>
<td>✓ (5/5)</td>
</tr>
</tbody>
</table>
RQ2: results

Applications with known IMAs

<table>
<thead>
<tr>
<th>Application</th>
<th>IMA location</th>
<th>Type</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc-1.06</td>
<td>more_arrays: 177</td>
<td>buffer overflow</td>
<td>✔  (5/5)</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>lookup: 577</td>
<td>buffer overflow</td>
<td>✔  (5/5)</td>
</tr>
<tr>
<td>gnupg-1.4.4</td>
<td>parse_comment: 2095</td>
<td>integer overflow</td>
<td>✔  (5/5)</td>
</tr>
<tr>
<td>mutt-1.4.2.li</td>
<td>utf8_to_utf7: 199</td>
<td>buffer overflow</td>
<td>✔  (5/5)</td>
</tr>
<tr>
<td>php-5.2.0</td>
<td>php_char_to_str_ex: 3152</td>
<td>integer overflow</td>
<td>✔  (5/5)</td>
</tr>
<tr>
<td>pine-4.44</td>
<td>rfc882_cat: 260</td>
<td>buffer overflow</td>
<td>✔  (5/5)</td>
</tr>
<tr>
<td>squid-2.3</td>
<td>ftpBuildTitleUrl: 1024</td>
<td>buffer overflow</td>
<td>✔  (5/5)</td>
</tr>
</tbody>
</table>

All attacks were detected with two taint marks
RQ2: results

Applications with known IMAs

<table>
<thead>
<tr>
<th>Application</th>
<th>IMA location</th>
<th>Type</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc-1.06</td>
<td>more_arrays: 177</td>
<td>buffer overflow</td>
<td>✔️ (5/5)</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>lookup: 577</td>
<td>buffer overflow</td>
<td>✔️ (5/5)</td>
</tr>
<tr>
<td>gnupg-1.4.4</td>
<td>parse_comment: 2095</td>
<td>integer overflow</td>
<td>✔️ (5/5)</td>
</tr>
<tr>
<td>mutt-1.4.2.li</td>
<td>utf8_to_utf7: 199</td>
<td>buffer overflow</td>
<td>✔️ (5/5)</td>
</tr>
<tr>
<td>php-5.2.0</td>
<td>php_char_to_str_ex: 3152</td>
<td>integer overflow</td>
<td>✔️ (5/5)</td>
</tr>
<tr>
<td>pine-4.44</td>
<td>rfc882_cat: 260</td>
<td>buffer overflow</td>
<td>✔️ (5/5)</td>
</tr>
<tr>
<td>squid-2.3</td>
<td>ftpBuildTitleUrl: 1024</td>
<td>buffer overflow</td>
<td>✔️ (5/5)</td>
</tr>
</tbody>
</table>

All attacks were detected with two taint marks

SPEC Benchmarks (“IMA free”)

<table>
<thead>
<tr>
<th>Application</th>
<th>IMA location</th>
<th>Type</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>vortex</td>
<td>SendMsg: 279</td>
<td>null-pointer dereference</td>
<td>✔️ (5/5)</td>
</tr>
</tbody>
</table>


......