SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting

Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang

USENIX Security 2021
Dynamic Taint Analysis

1 include <string.h>
2 void main(int argc, char **argv){
3 char buf[16];
4 strcpy(buf, argv[1]);
5 return;
6 }

stack

... argv argc return_addr

caller's ebp

buf (16 bytes)
Dynamic Taint Analysis

1 include <string.h>
2 void main(int argc, char **argv){
3 char buf[16];
4 strcpy(buf, argv[1]);
5 return;
6 }
Dynamic Taint Analysis

1 include <string.h>
2 void main(int argc, char **argv) {
3     char buf[16];
4     strcpy(buf, argv[1]);
5     return;
6 }

---

return address overwritten
control flow hijacked
caller’s ebp

buffer overflow

stack
... argv argc return_addr caller's ebp
buf (16 bytes)
Dynamic Taint Analysis

```
mov [0x8000200], eax
mov eax, [0x8000300]
```

```
mov eax, [0x8000300]
```
High Performance Overhead

Performance

Dynamic taint analysis frameworks often have a high performance overhead, which stop them from deploying in real world computer systems.
High Performance Overhead

Performance
Dynamic taint analysis frameworks often have a high performance overhead, which stop them from deploying in real world computer systems.

Example
A dynamic taint analysis framework called libdft imposes about 4x slowdown for gzip when compressing a file.
Reason 1: Dynamic Instruction Instrumentation

Architecture of Intel Pin
Reason 1: Dynamic Instruction Instrumentation

Insight 1
Taint logic can be instrumented \textit{statically} via static binary rewriting.

Architecture of Intel Pin
Reason 2: Over Instrumentation

Example

test eax, eax

This instruction will not affect any memory location or general register and does not propagate taint.
Reason 2: Over Instrumentation

Example

test eax, eax

This instruction will not affect any memory location or general register and does not propagate taint.

Insight 2

Taint logic can be instrumented *selectively* via value set analysis.
Static and Selective Instrumentation

Static Taint Analysis

Selective and static instrumentation is performed at compile time, which is equivalent to perform static taint analysis.

Research Questions

RQ: How to perform this static taint analysis?

RQ: How to reason about aliasing relation in binary code?
Static and Selective Instrumentation

**Static Taint Analysis**

Selective and static instrumentation is performed at **compile time**, which is equivalent to perform **static taint analysis**.

**Research Questions**

RQ: How to perform this static taint analysis?

⇒

RQ: How to reason about aliasing relation in binary code?
Value set analysis

Value set analysis (VSA) is a static binary analysis technique, which over-approximates the set of possible values for data objects at each program point.
Value Set Analysis

Memory Regions

VSA separates the memory space into three disjoint memory spaces: global, stack, heap regions.
Value Set Analysis

Value Sets

VSA computes the region and value sets based on:

- instruction semantics

Example:

```c
mov eax, [esp+4]
mov ebx, [0x8052160]
```
Value Set Analysis

Value Sets
VSA computes the region and value sets based on:
1. instruction semantics
2. data flow analysis

```
mov eax, [esp+4]
mov ebx, [0x8052160]
mov [0x8052100], ecx
mov ecx, eax
```
Static and Selective Instrumentation

Static Taint Analysis

Selective and static instrumentation is performed at compile time, which is equivalent to perform static taint analysis.

Research Questions

RQ: How to perform this static taint analysis?

RQ: How to reason about aliasing relation in binary code?
Static and Selective Instrumentation

Static Taint Analysis

Selective and static instrumentation is performed at **compile time**, which is equivalent to perform **static taint analysis**.

Research Questions

**RQ: How to perform this static taint analysis?**

**RQ: How to reason about aliasing relation in binary code?**
**Strawman approach**

Strawman approach identifies a **must-tainted** instruction set $I_t$ using VSA. However, VSA loses precision due to incomplete CFG and aliasing.

**Our approach**

Our approach **conservatively** identifies a **must-not-tainted** instruction set $I_u$ using VSA and taint the others.
**SELECTIVE_TAINT** Approach

$I$: ideally tainted instruction

$I'$: must-tainted instruction

$I_u$: must-not-tainted instruction

must-tainted analysis → imprecise
**SelectiveTaint** Approach

$I$: ideally tainted instruction

$\hat{I}$: must-tainted instruction

$I_u$: must-not-tainted instruction

Conservative must-tainted analysis $\rightarrow$ under-taint
SelectiveTaint Approach

\[ I: \text{ideally tainted instruction} \]
\[ I_t: \text{must-tainted instruction} \]
\[ I_u: \text{must-not-tainted instruction} \]

must-not-tainted analysis $\rightarrow$ imprecise
Selectivetaint Approach

$I$: ideally tainted instruction

$I_t$: must-tainted instruction

$I_u$: must-not-tainted instruction

Conservative must-not-tainted analysis $\rightarrow$ over-taint
We perform a conservative must-tainted analysis and taint the rest.
Identification Policy

Unreachable instructions
Removed from must-not-tainted set

<version_etc_arn>:
804b7a0: push ebp

Potentially tainted instructions
Removed from must-not-tainted set

8055c3c: call 8048f30 <__IO_getc@plt>
8055c41: mov eax, edx

Untainted operand instructions
Added to must-not-tainted set

8096a07: inc ebp

None taint-propagation instructions
Added to must-not-tainted set

8062456: jmp 806238b <mbslen+0x8b>
Identification Policy

Unreachable instructions
Removed from must-not-tainted set

<version_etc_arn>:
804b7a0: push ebp

Potentially tainted instructions
Removed from must-not-tainted set

8055c3c: call 804b8f30 <__IO_getc@plt>
8055c41: mov eax, edx

Untainted operand instructions
Added to must-not-tainted set

8096a07: inc ebp

None taint-propagation instructions
Added to must-not-tainted set

8062456: jmp 806238b <mbslen+0x8b>
Identification Policy

**Unreachable instructions**
Removed from must-not-tainted set

<version_etc_arn>:
804b7a0: push ebp

**Potentially tainted instructions**
Removed from must-not-tainted set

8055c3c: call 8048f30 <__IO_getc@plt>
8055c41: mov eax, edx

**Untainted operand instructions**
Added to must-not-tainted set

8096a07: inc ebp

**None taint-propagation instructions**
Added to must-not-tainted set

8062456: jmp 806238b <mbslen+0x8b>
Identification Policy

**Unreachable instructions**
Removed from must-not-tainted set

\(<version\_etc\_arn>:\)  
804b7a0: push ebp

**Potentially tainted instructions**
Removed from must-not-tainted set

8055c3c: call 8048f30 <__IO_getc@plt>  
8055c41: mov eax, edx

**Untainted operand instructions**
Added to must-not-tainted set

8096a07: inc ebp

**None taint-propagation instructions**
Added to must-not-tainted set

8062456: jmp 806238b <mbslen+0x8b>
Formal Proof of Must-not-tainted Analysis

Primary Inference Rules

**Unreachable**

\[ \exists i_s \in \text{source}, \ i_s \sim i, \ i \sim i_s \]

\[ \mathcal{I}_u := \{i\} \]

**UnknownOperand**

\[ \exists o \in \text{op}(i), \ V[o] = (\perp, \perp, \perp) \]

\[ \mathcal{I}_u := \{i\} \]

**UntaintedOperand**

\[ \forall o \in \text{op}(i), \ V[o] \subseteq \mathcal{V}_u \]

\[ \mathcal{I}_u \cup= \{i\} \]

**NonPropagateOpcode**

\[ \forall o \in \text{op}(i), \ V[o] \equiv V[o] \]

\[ \mathcal{I}_u \cup= \{i\} \]
### Formal Proof of Must-not-tainted Analysis

#### Auxiliary Inference Rules

**Control-flows:**

- **Reachable**
  \[
  \frac{suc(i_1, i_2)}{i_1 \sim i_2}
  \]

- **TransReachable**
  \[
  \frac{suc(i_1, i_2) \cdot suc(i_2, i_3)}{i_1 \sim i_3}
  \]

**Operands:**

- **LiteralOperand**
  \[
  \frac{l \in op(i)}{V_u \cup= V[l]}
  \]

- **LabelOperand**
  \[
  \frac{l \in op(i)}{V_u \cup= V[l]}
  \]

- **TaintSource**
  \[
  \frac{o \in \text{taintedop}(i_s) \cdot i_s \in \text{source}}{V_u \leftarrow V[o]}
  \]

- **TaintPropagate**
  \[
  \frac{o_1 \in \text{sourceop}(i) \cdot o_2 \in \text{destop}(i)}{V_u \leftarrow V[o_1] \subseteq V_u}
  \]

**Opcodes:**

- **PCRegChangeOpcode**
  \[
  \frac{V[pc] \cdot V[pc] \cdot \forall o \in \text{op}(i), V[o] \leftarrow V[o]}{I_u \leftarrow \{i\}}
  \]

- **StatusRegChangeOpcode**
  \[
  \frac{V[status] \cdot V[status] \cdot \forall o \in \text{op}(i), V[o] \leftarrow V[o]}{I_u \leftarrow \{i\}}
  \]
Formal Proof of Must-not-tainted Analysis

Theorem 1
Must-not-tainted analysis is sound, except for the precision loss due to imprecise CFG and VSA results.

Proof
We prove this theorem with induction.
1. In the first iteration, $I_u$ is $\emptyset$, must-not-tainted analysis is sound.
2. We next prove if the kth iteration, must-not-tainted analysis is sound, it also holds for the (k+1)th iteration.
Design

Selective Binary Taint Analysis

Original Binary → CFG Reconstruction → Value Set Analysis → Taint Instruction Identification → Binary Rewriting → Rewritten Binary
Performance Evaluation

The diagram shows the slowdown (normalized runtime) for various tools and systems. The x-axis represents different tools such as `tar`, `gzip`, `bzip2`, `scp`, `cat`, `comm`, `cut`, `grep`, `head`, `nl`, `od`, `ptx`, `shred`, `tail`, `truncate`, `uniq`, and `average`. The y-axis represents the slowdown ranging from 0 to 6.

- **Native**: The fastest runtime represented by a white bar.
- **nullpin**: Slower than native, represented by a light grey bar.
- **libdft**: Even slower, represented by a medium grey bar.
- **StaticTaintAll**: Slower yet, represented by a dark grey bar.
- **SelectiveTaint**: The slowest, represented by a black bar.

The diagram compares the performance of these tools and systems, highlighting the trade-off between runtime and tainting overhead.
Performance Evaluation

Results

On average 1.7x faster than libdft.
## Functionality Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>Category</th>
<th>Vulnerability</th>
<th>CVE ID</th>
<th>StaticTaintAll</th>
<th>SelectiveTaint</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoX 14.4.2</td>
<td>Sound Processing Utilities</td>
<td>Buffer Overflow</td>
<td>CVE-2019-8356</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TinTin++ 2.01.6</td>
<td>Multiplayer Online Game Client</td>
<td>Buffer Overflow</td>
<td>CVE-2019-7629</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dcraw 9.28</td>
<td>Raw Image Decoder</td>
<td>Buffer Overflow</td>
<td>CVE-2018-19655</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ngiflib 0.4</td>
<td>GIF Format Decoding Library</td>
<td>Buffer Overflow</td>
<td>CVE-2018-11575</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Gravity 0.3.5</td>
<td>Programming Language Interpreter</td>
<td>Buffer Overflow</td>
<td>CVE-2017-1000437</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MP3Gain 1.5.2</td>
<td>Audio Normalization Software</td>
<td>Buffer Overflow</td>
<td>CVE-2017-14411</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NASM 2.14.02</td>
<td>Assembler and Disassembler</td>
<td>Double Free</td>
<td>CVE-2019-8343</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Jhead 3.00</td>
<td>Exif Jpeg Header Manipulation Tool</td>
<td>Integer Underflow</td>
<td>CVE-2018-6612</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Nginx 1.4.0</td>
<td>Web Server</td>
<td>Buffer Overflow</td>
<td>CVE-2013-2028</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
# Functionality Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>Category</th>
<th>Vulnerability</th>
<th>CVE ID</th>
<th>StaticTaintAll</th>
<th>SelectiveTaint</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoX 14.4.2</td>
<td>Sound Processing Utilities</td>
<td>Buffer Overflow</td>
<td>CVE-2019-8356</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TinTin++ 2.01.6</td>
<td>Multiplayer Online Game Client</td>
<td>Buffer Overflow</td>
<td>CVE-2019-7629</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dcraw 9.28</td>
<td>Raw Image Decoder</td>
<td>Buffer Overflow</td>
<td>CVE-2018-19655</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ngiflib 0.4</td>
<td>GIF Format Decoding Library</td>
<td>Buffer Overflow</td>
<td>CVE-2018-11575</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Gravity 0.3.5</td>
<td>Programming Language Interpreter</td>
<td>Buffer Overflow</td>
<td>CVE-2017-1000437</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MP3Gain 1.5.2</td>
<td>Audio Normalization Software</td>
<td>Buffer Overflow</td>
<td>CVE-2017-14411</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NASM 2.14.02</td>
<td>Assembler and Disassembler</td>
<td>Double Free</td>
<td>CVE-2019-8343</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Jhead 3.00</td>
<td>Exif Jpeg Header Manipulation Tool</td>
<td>Integer Underflow</td>
<td>CVE-2018-6612</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Nginx 1.4.0</td>
<td>Web Server</td>
<td>Buffer Overflow</td>
<td>CVE-2013-2028</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

## Results

Detected all nine tested vulnerability as libdft.
Dynamic Taint Analysis

<table>
<thead>
<tr>
<th>Papers</th>
<th>Year</th>
<th>Static</th>
<th>Dynamic</th>
<th>Hardware</th>
<th>Parallel/Offline</th>
<th>Neural Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suh et al. [SLD04]</td>
<td>2004</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Newsome et al. [NS05]</td>
<td>2005</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clause et al. [CLO07]</td>
<td>2007</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bosman et al. [BSB11]</td>
<td>2011</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kemerlis et al. [KPJK12]</td>
<td>2012</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jee et al. [JPK⁺12]</td>
<td>2012</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jee et al. [JKKP13]</td>
<td>2013</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Ming et al. [MWX⁺15]</td>
<td>2015</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ming et al. [MWW⁺16]</td>
<td>2016</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Banerjee et al. [BDCN19]</td>
<td>2019</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>She et al. [SCS⁺20]</td>
<td>2020</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SelectiveTaint [CLZ21]</td>
<td>2021</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Related Work

### Binary Rewriting

- Uroboros [WWW15], Ramblr [WSB17], Multiverse [BLH18], Probabilistic Disassembly [MKS19], Ddisasm [FMS20], dyninst [BM11].

### Alias Analysis on Binary

- Points-to relations with Datalog [BN06], abstract address sets [DMW98], symbolic value sets [ABZT98].
Selective Taint Analysis

- Static and selective instruction instrumentation
- Conservative must-not-tainted analysis

The source code is available at https://github.com/OSUSecLab/SelectiveTaint. Email: {chen.4825, lin.3021}@osu.edu, yinqianz@acm.org
References I


