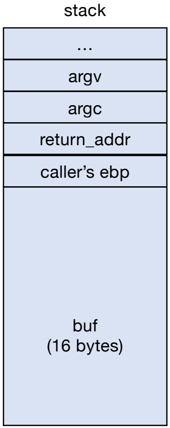
SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting

Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang

USENIX Security 2021

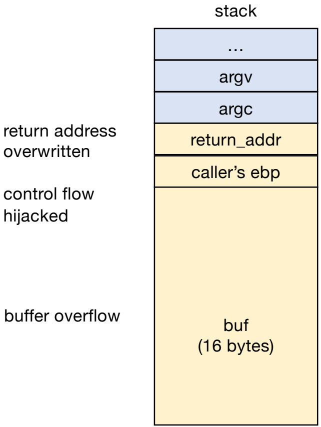
Dynamic Taint Analysis

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



Dynamic Taint Analysis

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



stack

...

argv

argc

return_addr

caller's ebp

buf
(16 bytes)

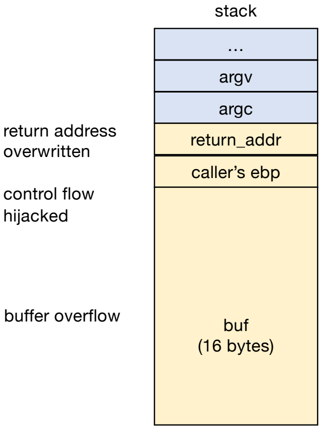
return address
overwritten

control flow
hijacked

buffer overflow

Dynamic Taint Analysis

```
include <string.h>
void main(int argc, char **argv){  <-- Taint Source
char buf[16];
strcpy(buf, argv[1]);            <-- Taint Propagation
return;                          <-- Taint Sink
}
```


stack

...

argv

argc

return_addr

caller's ebp

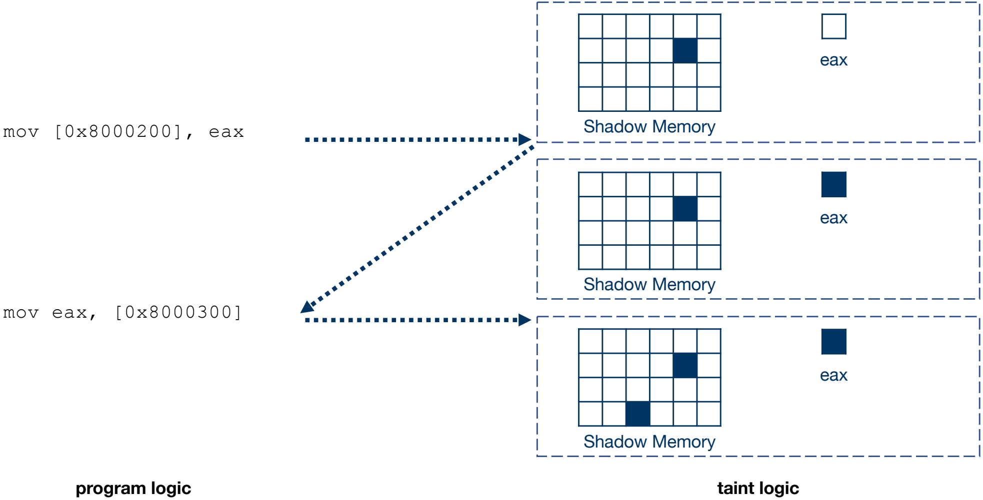
buf
(16 bytes)

return address
overwritten

control flow
hijacked

buffer overflow

Dynamic Taint Analysis



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

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

program logic

taint logic

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

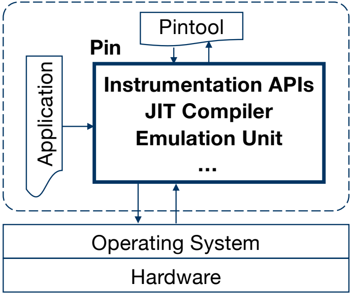
Performance

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

Example

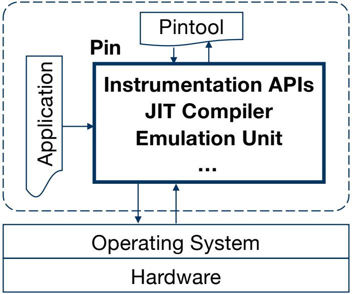
A dynamic taint analysis framework called `libdft` imposes about **4x** slowdown for `gzip` when compressing a file.

Reason 1: Dynamic Instruction Instrumentation



Architecture of Intel Pin

Reason 1: Dynamic Instruction Instrumentation



Architecture of Intel Pin

Insight 1

Taint logic can be instrumented **statically** via static binary rewriting.

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?

$\Downarrow$

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

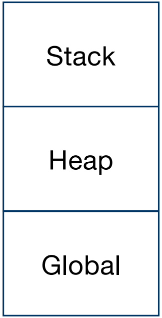
Value set analysis

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.

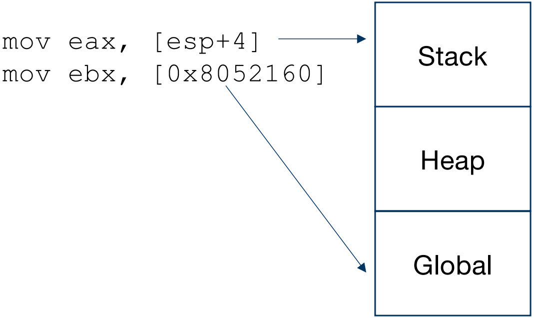


Value Set Analysis

Value Sets

VSA computes the region and value sets based on:

1. instruction semantics



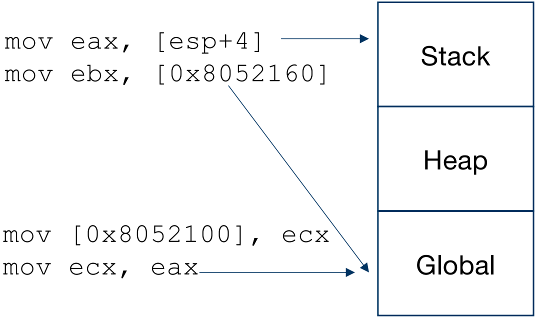
```
mov eax, [esp+4]
mov ebx, [0x8052160]
```

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?

$\Downarrow$

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?

SelectiveTaint Approach

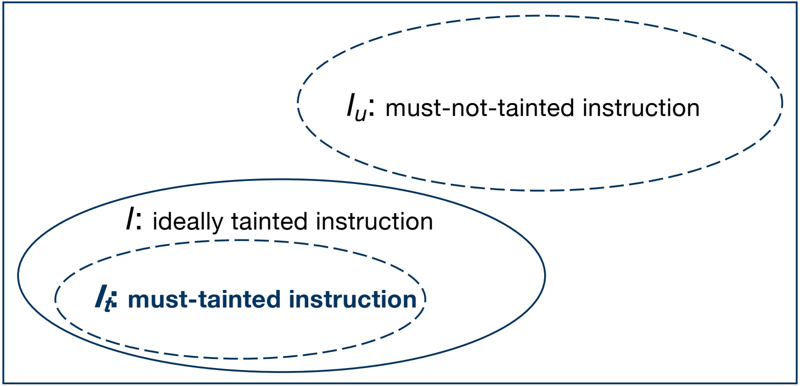
Strawman approach

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

Our approach

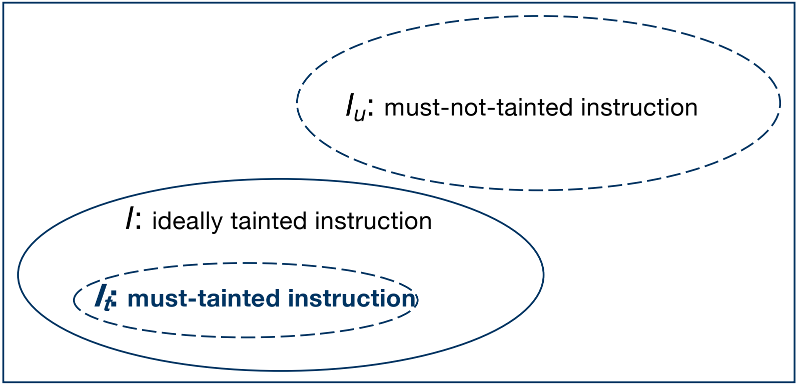
Our approach **conservatively** identifies a **must-not-tainted** instruction set I_u using VSA and taint the others.

SelectiveTaint Approach



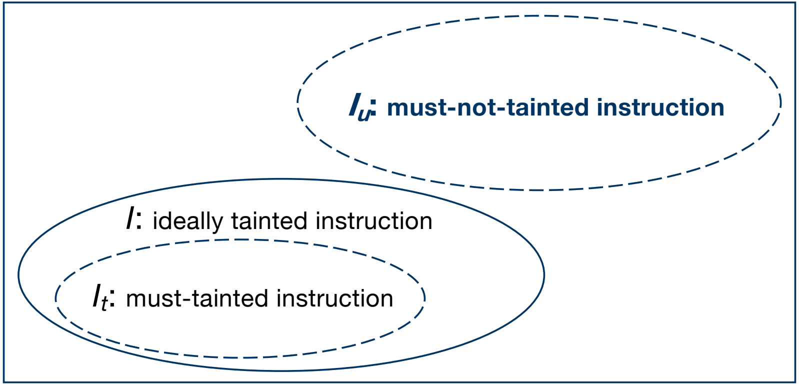
must-tainted analysis → imprecise

SELECTIVETAINT Approach



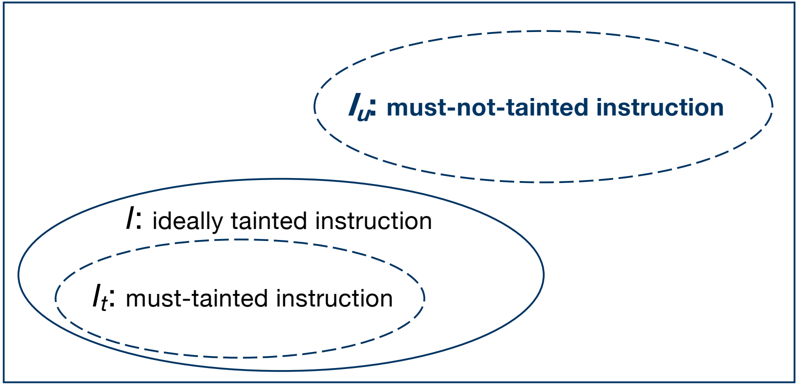
conservative must-tainted analysis → under-taint

SELECTIVETAINT Approach



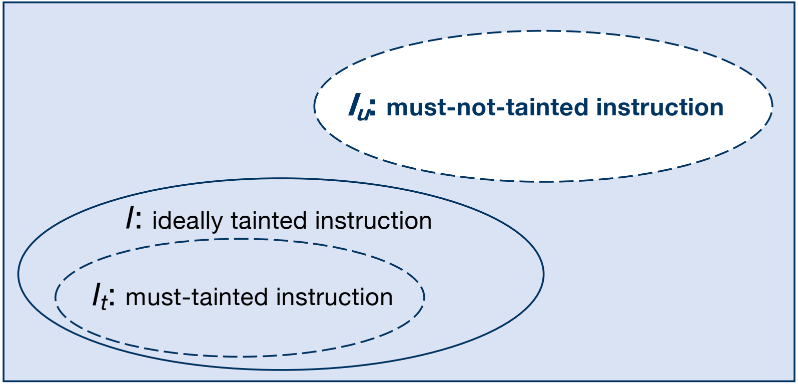
must-not-tainted analysis → imprecise

SELECTIVETAINT Approach



conservative must-not-tainted analysis → over-taint

SELECTIVETAINT Approach



I_u: must-not-tainted instruction

I: ideally tainted instruction

I_t: must-tainted instruction

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 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>
```

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

$$\textsc{Unreachable}\ \frac{\nexists i_s \in source,\ i_s \rightsquigarrow i,\ i \rightsquigarrow i_s}{\mathcal{I}_u\ -\!\!=\ \{i\}} \qquad \textsc{UnknownOperand}\ \frac{\exists o \in op(i),\ V[o] = (\bot,\bot,\bot)}{\mathcal{I}_u\ -\!\!=\ \{i\}}$$

$$\textsc{UntaintedOperand}\ \frac{\forall o \in op(i),\ V[o] \subseteq \mathcal{V}_u}{\mathcal{I}_u\ \cup\!\!=\ \{i\}} \qquad \textsc{NonPropagateOpcode}\ \frac{\forall o \in op(i),\ V[o] \overset{i}{\equiv} V[o]}{\mathcal{I}_u\ \cup\!\!=\ \{i\}}$$

Formal Proof of Must-not-tainted Analysis

Auxiliary Inference Rules

Control-flows:

$$\textsc{Reachable}\ \frac{succ(i_1,\ i_2)}{i_1\ \rightsquigarrow\ i_2} \qquad \textsc{TransReachable}\ \frac{succ(i_1,\ i_2) \quad succ(i_2,\ i_3)}{i_1\ \rightsquigarrow\ i_3}$$

Operands:

$$\textsc{LiteralOperand}\ \frac{l\ \in\ op(i) \quad l\ :\ \texttt{literal}}{\mathcal{V}_u\ \cup\!=\ V[l]} \qquad \textsc{LabelOperand}\ \frac{l\ \in\ op(i) \quad l\ :\ \texttt{label}}{\mathcal{V}_u\ \cup\!=\ V[l]}$$

$$\textsc{TaintSource}\ \frac{o\ \in\ taintedop(i_s) \quad i_s\ \in\ source}{\mathcal{V}_u\ -\!=\ V[o]} \qquad \textsc{TaintPropagate}\ \frac{o_1\ \in\ sourceop(i) \quad o_2\ \in\ destop(i) \quad V[o_1]\ \subseteq\ \mathcal{V}_u}{\mathcal{V}_u\ -\!=\ V[o_2]}$$

Opcodes:

$$\textsc{PCRegChangeOpcode}\ \frac{V[pc]\ {}^{i}\ V[pc] \quad \forall o\ \in\ op(i),\ V[o]\ \overset{i}{\equiv}\ V[o]}{\mathcal{I}_u\ \cup\!=\ \{i\}}$$

$$\textsc{StatusRegChangeOpcode}\ \frac{V[status]\ {}^{i}\ V[status] \quad \forall o\ \in\ op(i),\ V[o]\ \overset{i}{\equiv}\ V[o]}{\mathcal{I}_u\ \cup\!=\ \{i\}}$$

Formal Proof of Must-not-tainted Analysis

Theorem 1

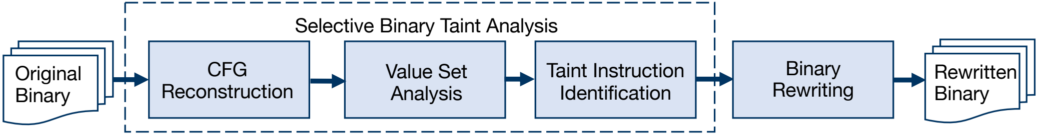
Must-not-tainted analysis is sound, except for the precision loss due to imprecise CFG and VSA results.

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

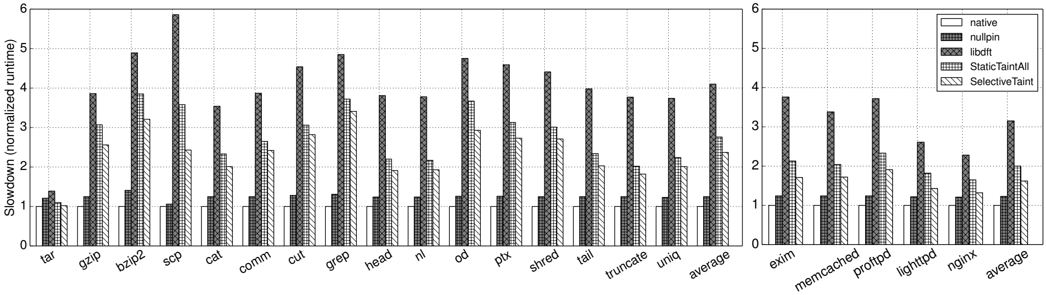
Value Set Analysis

Taint Instruction Identification

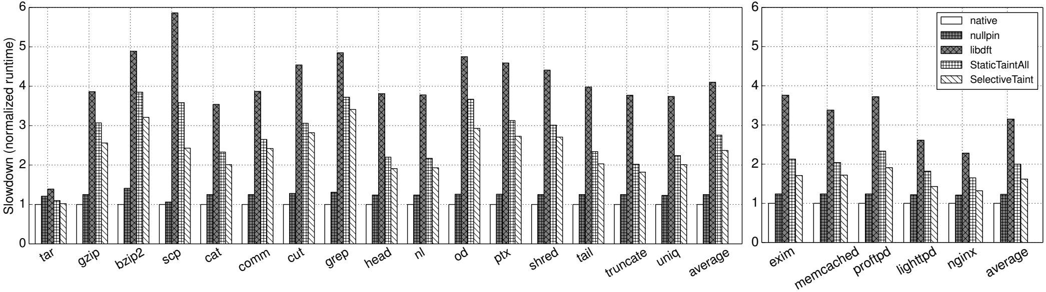
Binary Rewriting

Rewritten Binary

Performance Evaluation



Performance Evaluation



Results

On average **1.7x** faster than libdft.

Functionality Evaluation

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

Functionality Evaluation

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

Results

Detected **all nine** tested vulnerability as libdft.

Dynamic Taint Analysis

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

Related Work

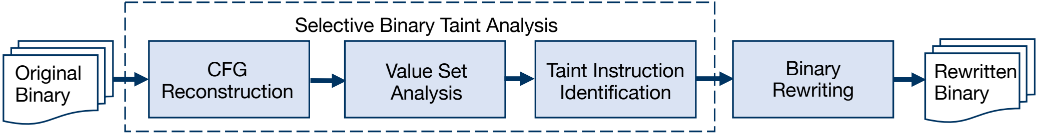
Binary Rewriting

Uroboros [WWW15], Ramblr [WSB+17], Multiverse [BLH18], Probabilistic Disassembly [MKS+19], Ddisasm [FMS20], dyninst [BM11].

Alias Analysis on Binary

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

Summary



SELECTIVETAINT

- 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

Wolfram Amme, Peter Braun, Eberhard Zehendner, and Francois Thomasset, *Data dependence analysis of assembly code*, Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (Washington, DC, USA), PACT '98, IEEE Computer Society, 1998, pp. 340–347.

S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy, *Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis*, Proceedings of the 40th IEEE Symposium on Security and Privacy, SP '19, 2019, pp. 712–726.

Erick Bauman, Zhiqiang Lin, and Kevin Hamlen, *Superset disassembly: Statically rewriting x86 binaries without heuristics*, Proceedings of the 25th Annual Network and Distributed System Security Symposium (San Diego, CA), NDSS '18, Feb. 2018.

Andrew R. Bernat and Barton P. Miller, *Anywhere, any-time binary instrumentation*, Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (New York, NY, USA), PASTE '11, ACM, 2011, pp. 9–16.

David Brumley and James Newsome, *Alias analysis for assembly*, Tech. report, Carnegie Mellon University, 2006.

Erik Bosman, Asia Slowinska, and Herbert Bos, *Minemu: The world's fastest taint tracker*, Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (Berlin, Heidelberg), RAID '11, Springer Berlin Heidelberg, 2011, pp. 1–20.

James Clause, Wanchun Li, and Alessandro Orso, *Dytan: A generic dynamic taint analysis framework*, Proceedings of the 2007 International Symposium on Software Testing and Analysis (New York, NY, USA), ISSTA '07, ACM, 2007, pp. 196–206.

Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang, *Selectivetaint: Efficient data flow tracking with static binary rewriting*, 30th USENIX Security Symposium (USENIX Security 21), USENIX Association, August 2021.

References II

Saumya Debray, Robert Muth, and Matthew Weippert, *Alias analysis of executable code*, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (New York, NY, USA), POPL '98, ACM, 1998, pp. 12–24.

Antonio Flores-Montoya and Eric Schulte, *Datalog disassembly*, Proceedings of the 29th USENIX Security Symposium, USENIX Security '20, USENIX Association, August 2020, pp. 1075–1092.

Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis, *ShadowReplica: Efficient parallelization of dynamic data flow tracking*, Proceedings of the 20th ACM Conference on Computer and Communications Security (New York, NY, USA), CCS '13, ACM, 2013, pp. 235–246.

Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis, *A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware*, Proceedings of the 19th Annual Network and Distributed System Security Symposium, NDSS '12, 2012.

Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis, *libdft: Practical dynamic data flow tracking for commodity systems*, Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (New York, NY, USA), VEE '12, ACM, 2012, pp. 121–132.

Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin, *Probabilistic disassembly*, Proceedings of the 41st International Conference on Software Engineering, ICSE '19, IEEE Press, 2019, p. 1187–1198.

Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu, *StraightTaint: Decoupled offline symbolic taint analysis*, Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (New York, NY, USA), ASE '16, ACM, 2016, pp. 308–319.

References III

- Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu, *TaintPipe: Pipelined symbolic taint analysis*, Proceedings of the 24th USENIX Security Symposium (Washington, D.C.), USENIX Security '15, USENIX Association, 2015, pp. 65–80.
- James Newsome and Dawn Song, *Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software*, Proceedings of the 12th Annual Network and Distributed Systems Security Symposium, NDSS '05, 2005.
- Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana, *Neutaint: Efficient dynamic taint analysis with neural networks*, 2020 IEEE Symposium on Security and Privacy (SP), IEEE, 2020, pp. 1527–1543.
- G. Edward Suh, Jaewook Lee, and Srinivas Devadas, *Secure program execution via dynamic information flow tracking*, 11th international conference on Architectural support for programming languages and operating systems, 2004, pp. 85–96.
- Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna, *Ramblr: Making reassembly great again*, Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS '17, 2017.
- Shuai Wang, Pei Wang, and Dinghao Wu, *Reassembleable disassembling*, Proceedings of the 24th USENIX Security Symposium (Washington, D.C.), USENIX Security '15, USENIX Association, 2015, pp. 627–642.