Code Reuse Attacks (II)

Slide credits: some slides and figures adapted from David Brumley, AC Chen, and others
Signature-Based Detection

- CRAs have a distinctive signature
  - Gadgets-gadgets-gadgets

- Some early work attempted to detect anomalies in the branch predictor

- “DROP: Detecting ROP malicious code”
  - P. Chen et al. ICISS 2009

\[
\begin{align*}
\text{if} \ (# \ \text{instructions between returns} \leq 5) \\
\text{then it is a gadget} \\
\text{if} \ (# \ \text{consecutive gadgets} \geq 3) \\
\text{then it is a ROP attack}
\end{align*}
\]
DROP

Consecutive Gadgets

min # gadgets to launch a CRA

S=3

max tolerable gadget length for CRA

N=5

Regular workloads

CRAs

• Implemented in software
• 5x slowdown
• Can we do better?
Building on DROP

- Apply the same idea to
  - Indirect jumps
  - Indirect calls

- Adjust the thresholds
Extending DROP for JOP

- Added dispatcher gadgets
- Indirect jumps/calls are not that frequent

- min # gadgets to launch a CRA
- Regular workloads

- max tolerable gadget length for CRA
Case of Long Gadgets

- **Side effects**
  - Overwrite registers/memory locations
  - May cause exceptions
    - `adc [ebx-0x74EBDBAC], cl`
  - Limited number of registers
    - One for the dispatcher
    - One for jumping back to the dispatcher
    - Only 6 left (in x86)
Gadget Length and State Changes

<table>
<thead>
<tr>
<th>Gadget Length</th>
<th>State Changes ≤ 2</th>
<th>State Changes ≤ 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>30%</td>
<td>5%</td>
</tr>
<tr>
<td>3</td>
<td>15%</td>
<td>5%</td>
</tr>
<tr>
<td>4</td>
<td>10%</td>
<td>5%</td>
</tr>
<tr>
<td>5</td>
<td>10%</td>
<td>5%</td>
</tr>
<tr>
<td>6</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>7</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>8</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>9</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>10</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>11</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Percentage of Total Gadgets Found

Gadget Length
Defeating DROP

- Make the attack pattern less predictable
  - Perform some unnecessary computation
  - Without harming attack state

- Call a function
  - Execute many instructions
  - Return to the same point in the attack
  - Caller saved registers stay intact
    - ebx, esi, edi, esp, ebp
Delay Gadgets

Dispatcher Gadget → Functional Gadget → Delay Gadget

atoi()

pop eax
jmp eax

add [ebx], edx
jmp esi

call [ecx-0x56000a00]
add bl, bh
inc ebx
add dh, bh
jmp edi
Summary of DROP Limitations

- Implemented in software
  - 5x slowdown
- Tight margin for detecting ROP
  - False positives/negatives
- Easily defeated by using delay gadgets
Another proposal: SCRAP

- **Signature-Based CRA Protection**
- Detects CRAs with delay gadgets
- No false positives for regular workloads
- Implemented in hardware
  - Low overhead
  - Protects legacy binaries
Attack Signatures

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>Indirect Jump</td>
</tr>
<tr>
<td>x</td>
<td>Indirect Call</td>
</tr>
<tr>
<td>y</td>
<td>Call</td>
</tr>
<tr>
<td>z</td>
<td>Return</td>
</tr>
<tr>
<td>a</td>
<td>All Other</td>
</tr>
</tbody>
</table>

- **DROP-like Signature Definition:**

\[ R_{N,S} = (a\{0, N\}(w|x))\{S, } \]

Up to N instructions

S or more gadgets
SCARP Attack Signatures

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>Indirect Jump</td>
</tr>
<tr>
<td>x</td>
<td>Indirect Call</td>
</tr>
<tr>
<td>y</td>
<td>Call</td>
</tr>
<tr>
<td>z</td>
<td>Return</td>
</tr>
<tr>
<td>a</td>
<td>All Other</td>
</tr>
</tbody>
</table>

- **Signatures with delay gadgets (N=5, S=3):**

  - Attack has 3 parts
  - Each part has a Gadget and Delays
  - A Gadget has up to 5 instructions
  - Delay is a function call
  - A function call can be a Delay as long as it is not an Attack itself

  \[
  \text{Attack} \rightarrow P P P \\
  P \rightarrow \text{Gadget Delays} | \text{Delays Gadget} \\
  \text{Gadget} \rightarrow \text{Indirect} | a \text{ Indirect} | a a \text{ Indirect} | a a a \text{ Indirect} | a a a a \text{ Indirect} \\
  \text{Indirect} \rightarrow w | x \\
  \text{Delays} \rightarrow \text{Delay Delays} | \varepsilon \\
  \text{Delay} \rightarrow \text{Call Body Return} \\
  \text{Call} \rightarrow x | y \\
  \text{Return} \rightarrow z \\
  \text{Body} \rightarrow \text{Delays Body} | \text{Body Delays} \\
  \text{Body} \rightarrow a \text{ Body} | \text{Body a} | \varepsilon \\
  \text{Body} \rightarrow \text{NotGadget NotAttack} \\
  \text{NotGadget} \rightarrow a a a a a \text{ Indirect} | a \text{ NotGadget} \\
  \text{NotAttack} \rightarrow \varepsilon | P | P P \\
\]
## Example Signatures

<table>
<thead>
<tr>
<th>Signature</th>
<th>Detected by DROP-like?</th>
<th>Detected by SCRAP?</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaawaawaaw</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>awaaxaaaaw</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>awaxaaaaaaazaxaw</td>
<td>✖</td>
<td>✔</td>
</tr>
<tr>
<td>awaxaaayaaazaazaxaw</td>
<td>✖</td>
<td>✔</td>
</tr>
</tbody>
</table>
Pushdown Automata

\[
\begin{array}{|c|c|}
\hline
\text{saturating counter} & \begin{align*}
\text{a:} & \quad \text{increase counter} \\
\text{w, x:} & \quad \text{if} \ \text{counter} < t_1, \ \text{output} \ S \\
\text{else} & \quad \text{output} \ L \\
\text{x, y:} & \quad \text{push the state} \\
\text{z:} & \quad \text{pop the state}
\end{align*} \\
\text{t}_1 & \begin{align*}
0 & \quad 0 & \quad 0 & \quad 0 & \quad 0 & \quad 0
\end{align*} \\
\hline
\end{array}
\]
Simple Implementation

- **Augment the stack of PDA with Secure Call Stack**
  - Save the PDA state for calls
  - Restore for returns
  - Modify for indirect jumps

- **Commit throttling**
  - Only one stack operation for a commit window
  - Negligible slowdown
  - Simplifies the circuitry
SCRAP Hardware
False Positives for SCRAP

Consecutive Gadgets

140 Shell-storm CRA exploits

Spec 2006 benchmarks

Gadget Length
Performance of SCRAP

![Graph showing the slowdown for various benchmarks including astar, bzip2, gcc, gbmk, h264ref, hmer, lbm, mcf, mlc, namd, omnetpp, perfbench, povray, sjeng, soplex, sphinx3, xalan-c-bmk, and the average slowdown. The x-axis represents different benchmarks, and the y-axis represents slowdown in percentage.](image-url)
Conclusions

- Signature-based detection of CRAs is promising
  - No source code, simple, low-overhead, effective

- Naïve approaches can be defeated
  - Delay gadgets

- But is SCRAP really more secure?
  - Are there other ways to hide?
  - Paper introduces another detector that tolerates longer gadgets
    - Becomes harder to avoid false positives
  - Attacks can be short
Gadget Length

- Long gadget means more intermediate instructions
Two threshold SCRAP

<table>
<thead>
<tr>
<th>saturating counter</th>
<th>$a$: increase counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0.0.0.0</td>
<td>$w, x$: if $counter &lt; t_1$, output S</td>
</tr>
<tr>
<td></td>
<td>else if $counter &lt; t_2$, output M</td>
</tr>
<tr>
<td></td>
<td>else output L</td>
</tr>
<tr>
<td>$t_1$ 0.0.1.0.0.0</td>
<td>$x, y$: push the state</td>
</tr>
<tr>
<td>$t_2$ 1.1.0.0.1.0</td>
<td>$z$: pop the state</td>
</tr>
</tbody>
</table>

**Diagram:**

- States: $q_0, q_1, q_2, q_3, q_a, q'_1, q'_2, q'_3$
- Transitions:
  - From $q_0$: $S, L$ to $q_1$
  - From $q_1$: $S$ to $q_2$
  - From $q_2$: $S$ to $q_3$
  - From $q_3$: $S$ to $q_a$
  - From $q'_1$: $S$ to $q'_2$
  - From $q'_2$: $S$ to $q'_3$
  - From $q'_3$: $S$ to $q_a$
- Labels: $S, L, M, M, L$

UCR
Our story so far...

Control Flow Hijacks

Attack
- Buffer Overflows
- Format String Vulnerabilities
  - More Buffer Overflows
  - Mem Read
  - Mem Write

Defense
- Computation
- DEP/NX
  - return
  - libc
  - Return-Oriented Programming

Unauthorized Control Information Tampering

http://propercourse.blogspot.com/2010/05/i-believe-in-duct-tape.html
How to really solve the problem?

- Prevent buffer overflows? How? Later

- Prevent illegal control flow? Hmm....

- Control Flow Integrity (CFI), Abadi et al, CCS 2005
  - Important paper in security
  - Control flow: how your program counter changes
  - Control flow altering instructions: branches, calls, returns, ...
  - CFI: How to specify and preserve legal control flow in a program
    - How does this help with CRAs?
    - Discussion: how do we approach this problem?
CFI: Control-Flow Integrity

Main idea: pre-determine control flow graph (CFG) of an application
- Static analysis of source code
- Static binary analysis ← CFI
- Execution profiling
- Explicit specification of security policy

Execution must follow the pre-determined control flow graph
Control Flow Integrity

- protects against powerful adversary
  - with full control over entire data memory
- widely-applicable
  - language-neutral; requires binary only
- provably-correct & trustworthy
  - formal semantics; small verifier
- efficient
  - hmm... 0-45% in experiments; average 16%
CFI Adversary Model

**CAN**
- Overwrite any data memory at any time
  - stack, heap, data segs
- Overwrite registers in current context

**CANNOT**
- Execute Data
  - NX takes care of that
- Modify Code
  - text seg usually read-only
- Write to %ip
  - true in x86
- Overwrite registers in other contexts
  - kernel will restore regs
CFI Overview

Invariant: Execution must follow a path in a control flow graph (CFG) created ahead of run time.

Method:

- build CFG statically, e.g., at compile time
- instrument (rewrite) binary, e.g., at install time
  - add IDs and ID checks; maintain ID uniqueness
- verify CFI instrumentation at load time
  - direct jump targets, presence of IDs and ID checks, ID uniqueness
- perform ID checks at run time
  - indirect jumps have matching IDs
CFI: Binary Instrumentation

- Use binary rewriting to instrument code with runtime checks

- Inserted checks ensure that the execution always stays within the statically determined CFG
  - Whenever an instruction transfers control, destination must be valid according to the CFG

- Goal: prevent injection of arbitrary code and invalid control transfers (e.g., return-to-libc)
  - Secure even if the attacker has complete control over the thread’s address space
CFG Example

```c
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len) {
    sort( a, len, lt );
    sort( b, len, gt );
}
```
CFI: Control Flow Enforcement

- For each control transfer, determine statically its possible destination(s)

- Insert a **unique bit pattern at every destination**
  - Two destinations are equivalent if CFG contains edges to each from the same source
    - This is imprecise *(why?)*
    - Use same bit pattern for equivalent destinations

- Insert binary code that at runtime will check whether the bit pattern of the target instruction matches the pattern of possible destinations
CFI: Example of Instrumentation

**Original code**

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Destination Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF E1</td>
<td>jmp ecx ; computed jump</td>
<td>8B 44 24 04 mov eax, [esp+4] ; dst</td>
</tr>
</tbody>
</table>

**Instrumented code**

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Destination Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>B8 77 56 34 12</td>
<td>mov eax, 12345677h inc eax cmp [ecx+4], eax jne error_label jmp ecx</td>
<td>3E 0F 13 05 prefetchnta</td>
</tr>
</tbody>
</table>

Jump to the destination only if the tag is equal to “12345678”

Abuse an x86 assembly instruction to insert “12345678” tag into the binary
CFI: Preventing Circumvention

- **Unique IDs**
  - Bit patterns chosen as destination IDs must not appear anywhere else in the code memory except ID checks

- **Non-writable code**
  - Program should not modify code memory at runtime
    - What about run-time code generation and self-modification?

- **Non-executable data**
  - Program should not execute data as if it were code

- **Enforcement: hardware support + prohibit system calls that change protection state + verification at load-time**
Improving CFI Precision

- Suppose a call from A goes to C, and a call from B goes to either C, or D *(when can this happen?)*
  - CFI will use the same tag for C and D, but this allows an “invalid” call from A to D
  - Possible solution: duplicate code or inline
  - Possible solution: multiple tags

- **Function F is called first from A, then from B; what’s a valid destination for its return?**
  - CFI will use the same tag for both call sites, but this allows F to return to B after being called from A
  - Solution: *shadow call stack*
CFI: Security Guarantees

- Effective against attacks based on illegitimate control-flow transfer
  - Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge

- Does not protect against attacks that do not violate the program’s original CFG
  - Incorrect arguments to system calls
  - Substitution of file names
  - Other data-only attacks
Possible Execution of Memory

Possible flow destination
Safe code/data

Data memory

Code memory for function A

Code memory for function B

x86  x86/NX  RISC/NX  x86/CFI

[Erliingsson]
Control Flow Integrity

- Generate Control Flow Graph
  - Unique labels for edges

- Instrument the code with checks
  - Each indirect branch checks target for a label

```assembly
cmp [ecx], 12345678h
jne error_label
lea ecx, [ecx+4]
jmp ecx
```

```
jmp ecx
... 
```

```
... 
mov eax, [esp+4]
... 
```

```
<data 12345678h>
mov eax, [esp+4]
... 
```

Source

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>jmp ecx</td>
<td>mov eax, [esp+4]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
CFI Limitations

- Need to construct Control Flow Graph
  - Sometimes difficult; especially for obfuscated code
  - No publicly available tools for constructing CFG from binaries
    - This is changing

- Runtime performance overhead due to extra instructions
  - About 20% for SPEC 2K6 Benchmarks

- Problems with dynamic linking

- Indirect branches are a headache

- Does not handle unintended instructions

- Imprecise
Where Do Jump Targets Go?

- **Typical use of indirect jump instruction**
  - To efficiently implement switch-case statements
    - Target is in the same function
  - To support dynamic linking
    - Target is a function entry point

**Key Observation:**

Legitimate jump targets ARE NOT in the middle of another function
Branch Regulation

- **Enforce the following rules:**
  - Returns go to the correct addresses
  - Jumps target same functions or function entry points
  - Calls target function entry points

- **Use hardware checks for enforcement**
  - Low performance overhead (around 2%)
  - Unintended instructions are handled
  - No CFG needed
  - Requires minimal binary annotations

- **Security shown for common libraries**
How Does BR Mitigate JOP Attacks?

- Stack frame for `main()`
  - `xor ecx, ecx`  
  - `mul ecx`  
  - `jmp esi`  
  - `lea ebx, [esp+8]`  
  - `jmp esi`  
  - `mov al, 11`  
  - `int 0x80`

- Stack frame for `vulnerable()`
  - `pop ecx`  
  - `jmp esi`  
  - `pop edi`  
  - `lea ebx, [esp+8]`  
  - `mov ecx, [esp+8]`  
  - `mov edi, ecx`  
  - `lea ebx, [esp+8]`  
  - `jmp esi`  
  - `xor ecx, ecx`  
  - `pop ecx`  
  - `jmp esi`  
  - `popa`  
  - `jmp esi`

- Stack frame for `function1()`

- Stack frame for `function2()`

- Stack growth from `0x0000` to `0xFFFF`
Implementing Branch Regulation

- Use a Secure Call Stack

- Annotate function entry points with function size info

- Compare jump targets with function bounds and allow if checks pass

- Store bounds in secure call stack

- Cache a few secure stack entries for performance
Inserting Code Annotations

Symbol Table

```
... 08048410 g F .text 005a __libc_csu_init
080483e4 g F .text 001c main
080482b8 g F .init 0000 _init
...
```

Original Code

```
080483e4 <main>:
  80483e4: push ebp
  80483e5: mov ebp, esp
  ...
  80483f9: mov eax, 0
  80483fe: leave
  80483ff: ret
```

Annotated Code

```
080483d2 <main>:
  80483d2: <annotation> <size=001c>
  80483e4: push ebp
  80483e5: mov ebp, esp
  ...
  80483f9: mov eax, 0
  80483fe: leave
  80483ff: ret
```
Executing Call Instruction

```
Fetch -> Decode -> Execute -> Commit

<annotation>  0098

Instruction Cache

8048232: call eax
8048236: ...
80482bc: <annotation> 0098
80482be: push esp
80482c0: ...
8048300: jmp ecx
8048354: ret
```

Function Bounds Stack

<table>
<thead>
<tr>
<th>Base</th>
<th>Bound</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>80482bc</td>
<td>8048354</td>
<td>8048236</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

BR Check
Executing Jump Instruction

Fetch → Decode → Execute → Commit

call eax
<annotation> 0x98
push esp
...  
jmp ecx
...  
jmp ecx
...  
ret
...  

Instruction Cache

0x80483AA

<table>
<thead>
<tr>
<th>Base</th>
<th>Bound</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048354</td>
<td>0x80482BC</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Function Bounds Stack

BR Check
Executing Return Instruction

Fetch | Decode | Execute | Commit

ret

0x8048236: call eax
0x80482BC: <annotation> 0x98
0x80482BE: push esp
0x80482C0: ...

0x8048300: jmp ecx
0x8048354: ret

Instruction Cache

Function Bounds Stack

BR Check

<table>
<thead>
<tr>
<th>Base</th>
<th>Bound</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80482Bc</td>
<td>0x8048354</td>
<td>0x8048236</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Return Address from Stack
Security of Branch Regulation

- Effectively reduces the exploitable code base to the current function

- An attacker needs to find in same function:
  - A vulnerability to exploit
  - Sufficient functional gadgets
  - Dispatcher gadget
  - A system call instruction
## Number of Vulnerable Functions

<table>
<thead>
<tr>
<th>Library</th>
<th>Total Number of Functions</th>
<th>Functions with Dispatcher Gadget</th>
<th>Functions with Dispatcher Gadget and System Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc</td>
<td>7775</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>libm</td>
<td>1077</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libcrypto</td>
<td>7991</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>libgcrypt</td>
<td>1135</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>libssl</td>
<td>1017</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Unintended Instructions

- Mostly garbage instructions
- Usually more side effects than intentional
Additional Security Analysis in the Paper

- Analysis of available gadgets
- Dispatcher gadget discovery algorithms
- Analysis of gadget side effects
  - Gadget length
  - Impact of unintended gadgets
Increase of Executable Size

- Control Flow Integrity
- Branch Regulation
Impact of Secure Stack Cache Size

Slowdown vs Cache size graph showing a decrease in slowdown as cache size increases.
BR Performance with 4-Entry Cache

Control Flow Integrity vs. Branch Regulation

Slowdown
Another CFI Solution point: CFI Mon

- Some solutions try to reuse existing hardware
  - Use performance monitoring unit on x86
  - kBouncer: use Last Branch Record (LBR)
  - CFIMon: use the branch target store to implement a CFI reference monitor
Precision Mode of Intel CPU
---Branch Trace Store (BTS) Mechanism

- Record all control transfer precisely into a predefined buffer
  - jump, call, return, interrupt and exception
  - also record the addresses of branch source and target

- Let a monitor get the trace in a batch
  - an interrupt will be delivered when the buffer is nearly full

- Obtain all the branch information of a running application, help users locate the vulnerabilities
Main Idea

- The CFI of an application can be maintained if we can
  - get a legal set of branch target addresses for every branch
  - check whether the target address of every branch is within the corresponding legal set at runtime
Branch Classification in X86 ISA
---Direct Branch & Its Target Address

- Direct Branch
  - Direct jump
    - jnz c2ef0 <__write>
  - Direct call
    - callq 34df0 <abort>

- Since the code is read-only and cannot be modified during runtime, both the direct jump and direct call are considered safe one
Branch Classification in X86 ISA
---Indirect Branch & Its Target Address

- Indirect Branch
  - Indirect jump
    - jmpq *%rdx
    - not possible to gain the whole target address set just by static analysis
  - Indirect call
    - callq *%rax
    - its target address could be obtained by statically scanning the binary code of the application and the libraries it uses
  - Return
    - retq
    - its target address could also be obtained by scanning the binary code.

  In general, the target address of a return has to be the one next to a call

- Dynamic Training
  A call can only transfer control to the start of a function.
CFIMon: 2 Phases

- **Offline phase**
  - build a *legal set* of target addresses for each branch instruction

- **Online phase**
  - diagnose possible attacks with legal sets following a number of rules
    - determine the status of the branch as *legal*, *illegal* or *suspicious*
Offline Analysis

--- obtain legal set: ret_set, call_set

- Scans the binary of application and dynamic libraries to get
  - ret_set
    - contains all addresses of the instructions next to each call
  - special cases
  - call_set
    - contains all addresses of the first instruction of each function

```c
int add (int a, int b){
    printf("1st inst.");
    ...
}
```
Online Detection

\(<\text{source}, \text{target}>\)

- special case?
  - yes: legal
  - no: illegal/suspicious

- \(<\text{source}>\) is direct branch?
  - yes: legal
  - no: illegal/suspicious

- \(<\text{source}>\) is return?
  - yes: ret_set
    - yes: legal
    - no: illegal
  - no: call_set/

- \(<\text{source}>\) is indirect call?
  - yes: call_set
    - yes: legal
    - no: illegal
  - no: train_set

Consider the state of a branch depending on \(<\text{target}>\)

slide-window mechanism
Architecture

A kernel extension