

# SYMBOLIC OF THE FUTURE

James Anderson

# SYMBOLIC OF THE FUTURE

6 reasons why symbolic  
execution is better than IDA

Number 5 will blow you mind!

# SYMBOLIC OF THE FUTURE

6 reasons why symbolic  
execution is better than IDA

Number 5 will blow you mind!

James Anderson

**BUZZFEED  
EDITION**

Get the presentation:  
[malwr.co/symbolic](https://malwr.co/symbolic)

(I swear it's not malware)

## Who am I?:

- Malware reverse engineer 5 years
- Security Engineer 2 years

# #1

Reverse engineering is great  
but if it takes more than a day it  
takes **too long.**

Grab the pitchforks lads!



# Grab the pitchforks!

## Hear me out first...

(besides this is buzzfeed worthy clickbait so now you have to listen)



# How every task starts



# Let's do this



# How every task starts

Looks pretty straight forward

The screenshot displays the IDA Pro interface for the file 'G:\Downloads\TheMatrix.exe'. The 'Functions window' on the left lists several subroutines, with 'sub\_4057E0' selected. The 'Graph overview' window at the bottom left shows a simplified control flow graph with five nodes connected by arrows. The main assembly view shows the following code:

```
sub_4057E0 proc near
push    ebx
push    esi
push    edi
test    dl, dl
jz      short loc_4057EF

add     esp, 0FFFFFFF0h
call    sub_402AC8

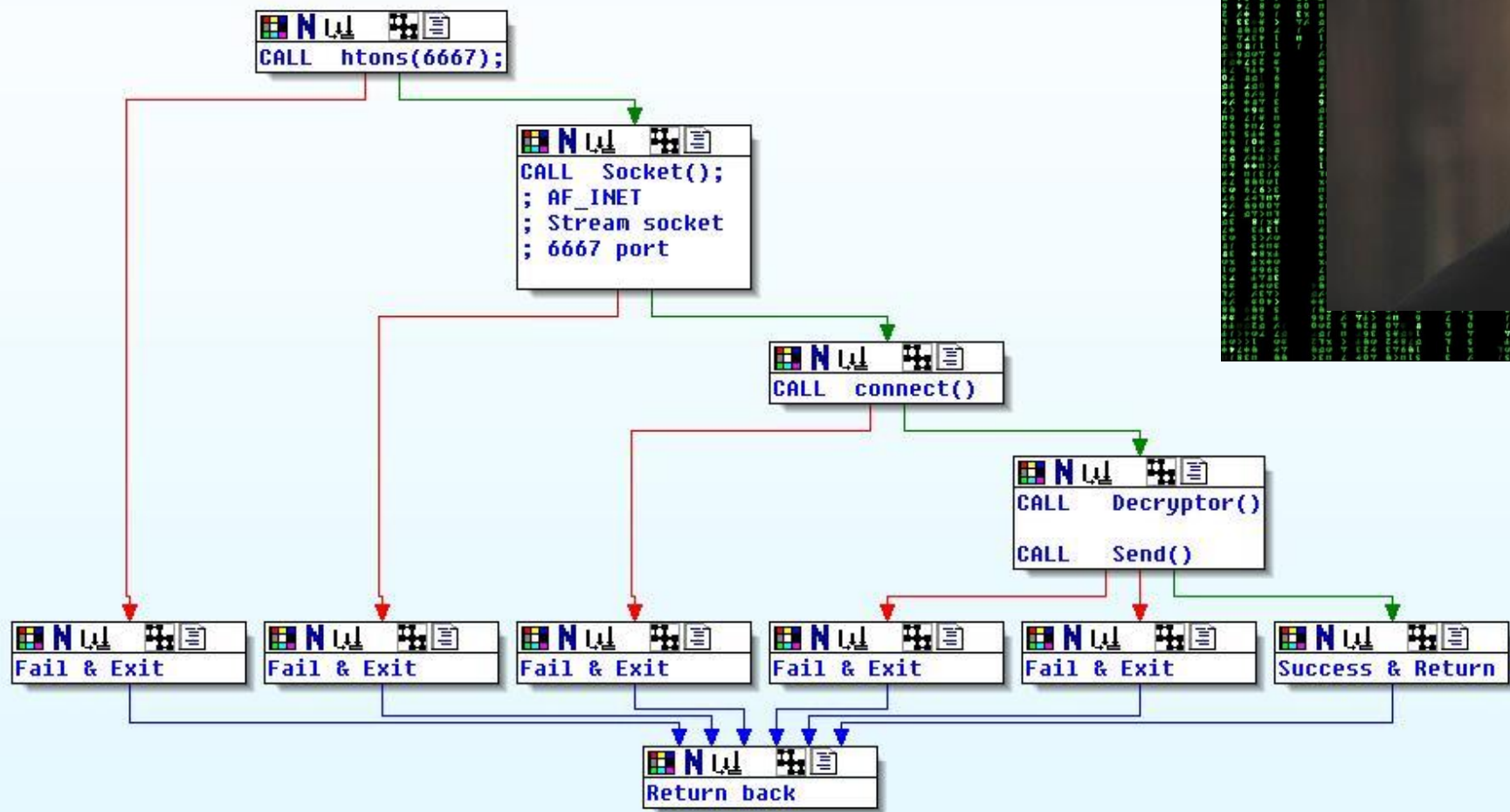
loc_4057EF:
mov     esi, ecx
mov     ebx, edx
mov     edi, eax
lea    eax, [edi+4]
mov     edx, esi
call    sub_4031E8
test    bl, bl
jz      short loc_405800

pop     large dword ptr fs:0
add     esp, 0Ch

loc_405800:
mov     eax, edi
pop     edi
pop     esi
pop     ebx
retn
sub_4057E0 endp ; sp-analysis failed
```

The status bar at the bottom indicates the current address is 100.00% [(-421, -56) | (1118, 781) | 00004BE2 | 004057E2: sub\_4057E0+2].

# I'm seeing into the matrix...



# The Investigation

```
add     edx, eax
mov     [ebp+var_89C], edx
mov     edx, [ebp+var_874]
movsx   eax, byte ptr [ecx+0Fh]
sub     edx, eax
lea     eax, [ebp+var_348]
mov     [ebp+var_874], edx
mov     [ebp+var_28], 0
mov     [ebp+var_24], 0
mov     [ebp+var_20], 0
mov     [ebp+var_1C], 0
mov     [ebp+var_18], 0
mov     [ebp+var_14], 0
push    eax
push    3
mov     eax, [ebp+arg_4]
push    eax
push    ebx
call    sub_401B30
add     esp, 10h
mov     [ebp+var_87C], eax
mov     eax, [ebp+arg_10]
test    eax, eax
jle     loc_40378F
```

```
mov     [ebp+var_8A0], 0
mov     [ebp+var_894], 0
mov     [ebp+var_890], 0
mov     [ebp+var_88C], 0
mov     [ebp+var_884], 0
mov     [ebp+var_880], 0
mov     [ebp+var_878], 0
```

Wait a second!

What does this call do?



# The Investigation

IDA - G:\Downloads\TheMatrix.idb (TheMatrix.exe)

File Edit Jump Search View Debugger Options Windows Help

Functions window

Function name	Sea
sub_407B0C	COC
sub_407B8C	COC
sub_407C18	COC
sub_407CDC	COC
sub_407D44	COC
sub_407D8C	COC
sub_407E18	COC
sub_407E3C	COC
sub_407E70	COC
sub_407F34	COC
sub_407FD8	COC
sub_407FE4	COC
sub_408024	COC
sub_408064	COC
sub_4080C4	COC
sub_40814C	COC
sub_408190	COC
sub_4081D4	COC
sub_408200	COC
sub_408274	COC
sub_4082EC	COC
sub_40836C	COC
sub_408A08	COC

Line 395 of 403

Graph overview

64.00% | (-389,603) | (370,470) | 00008B24 | 00409724: sub\_409724

Output window

Python

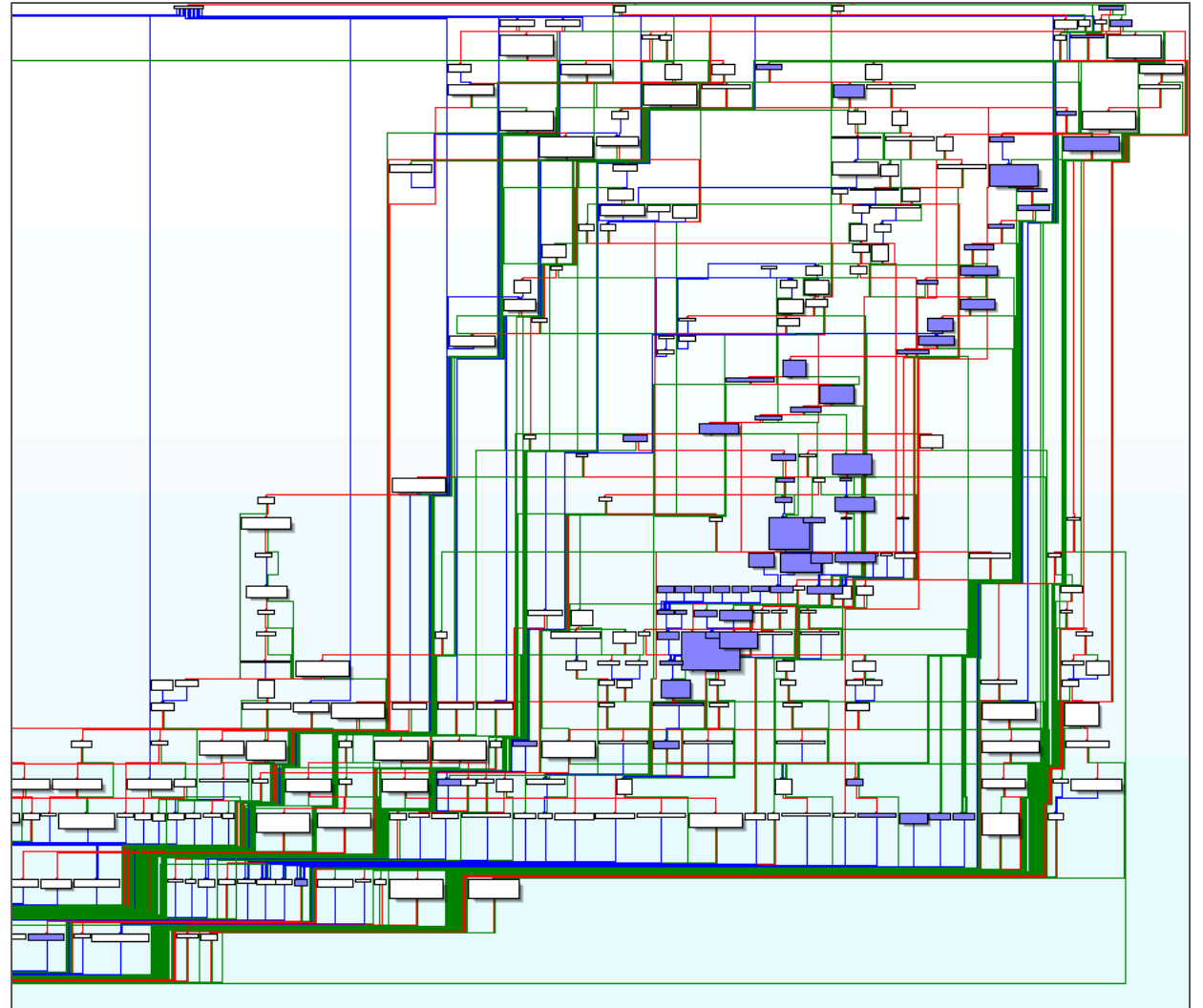
AU: idle | Down | Disk: 194GB

What is this!  
Even more calls

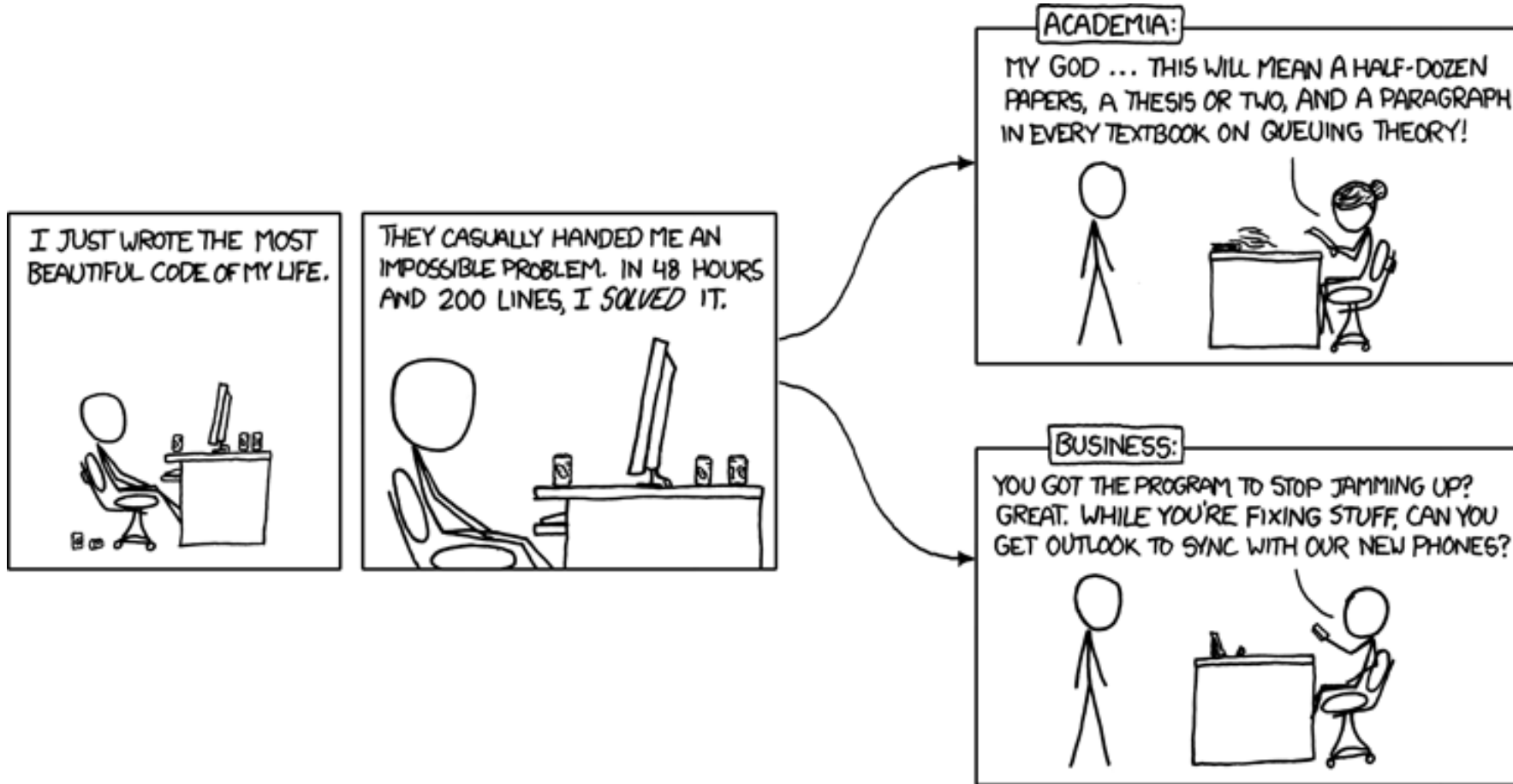
# Plz Help...



Plz halp, I've gone too deep.

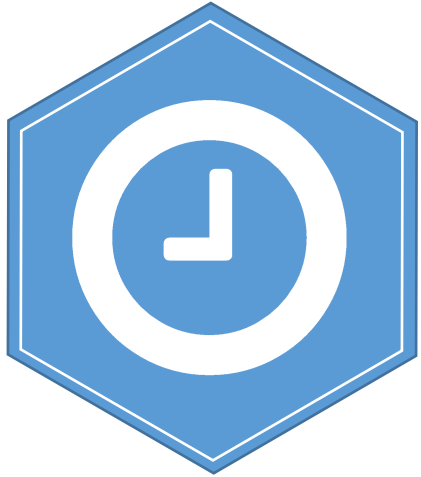


# Speed is the key





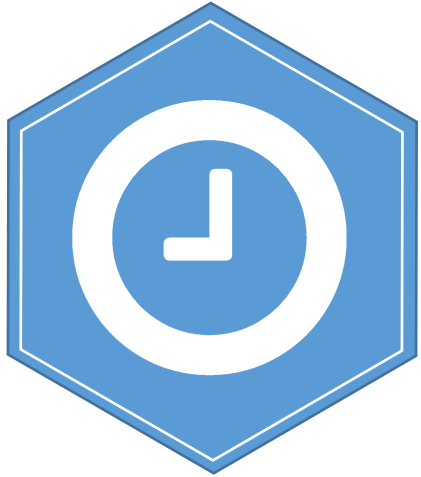
# What was I looking at? Ah nevermind...



## **Time Matters**

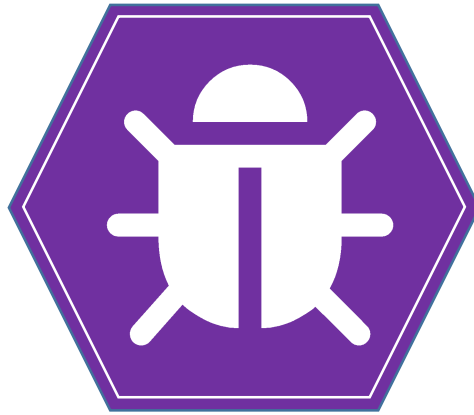
If others are waiting on the results then they will move on quickly

# What was I looking at? Ah nevermind...



## Time Matters

If others are waiting on the results then they will move on quickly



## Tracking Bugs

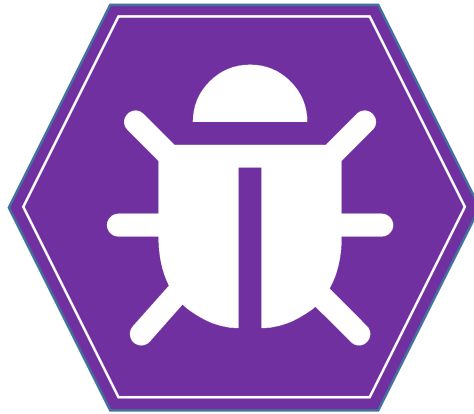
How long can we do it for? I want results but how soon?

# What was I looking at? Ah nevermind...



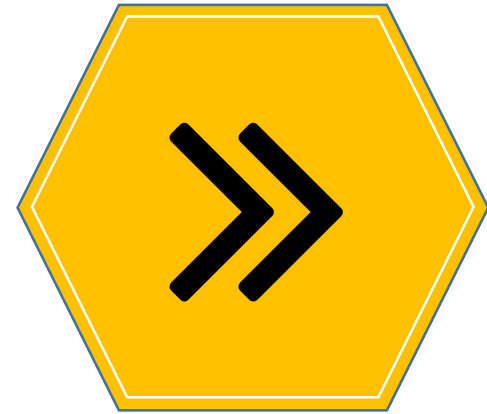
## Time Matters

If others are waiting on the results then they will move on quickly



## Tracking Bugs

How long can we do it for? I want results but how soon?



## Faster at the game

If others are waiting on the results then they will move on quickly

# #2

Dynamic is fast and efficient,  
but you always lose out on the  
details.

# No one is surprised by this



Dynamic execution like cuckoo gives us some of the answers but it only gets us part of the way.

Quick Overview   Static Analysis   Behavioral Analysis   Network Analysis   Dropped Files   Admin

Process Tree

- upclicker.exe 1156
- Explorer.EXE 2004
  - iexplore.exe 1944

Search upclicker.exe Explorer.EXE iexplore.exe

nnr999rn Search

Results

network filesystem registry process services synchronization

Process: upclicker.exe (1156)

2015-03-04 16:02:51,253	NtCreateMutant	Handle: 0x00000088 InitialOwner: 0 MutexName: nnr999rn	success	0x00000000		
----------------------------	----------------	---	---------	------------	--	--

# No one is surprised by this

## **While sandboxes give good detail they lose the details.**

- How are the comms encrypted?
- What other persistence methods are there?
- Does it have any anti-vm/ anti debugger techniques
- What did we miss?

## **Static gives the detail but can be slow**

- Many unresolved symbols
- Debugger side by side to help analysis

What if there was another way?



# Symbolic Execution



Symbolic execution is testing technique to aid the generation of test data and in proving the program quality

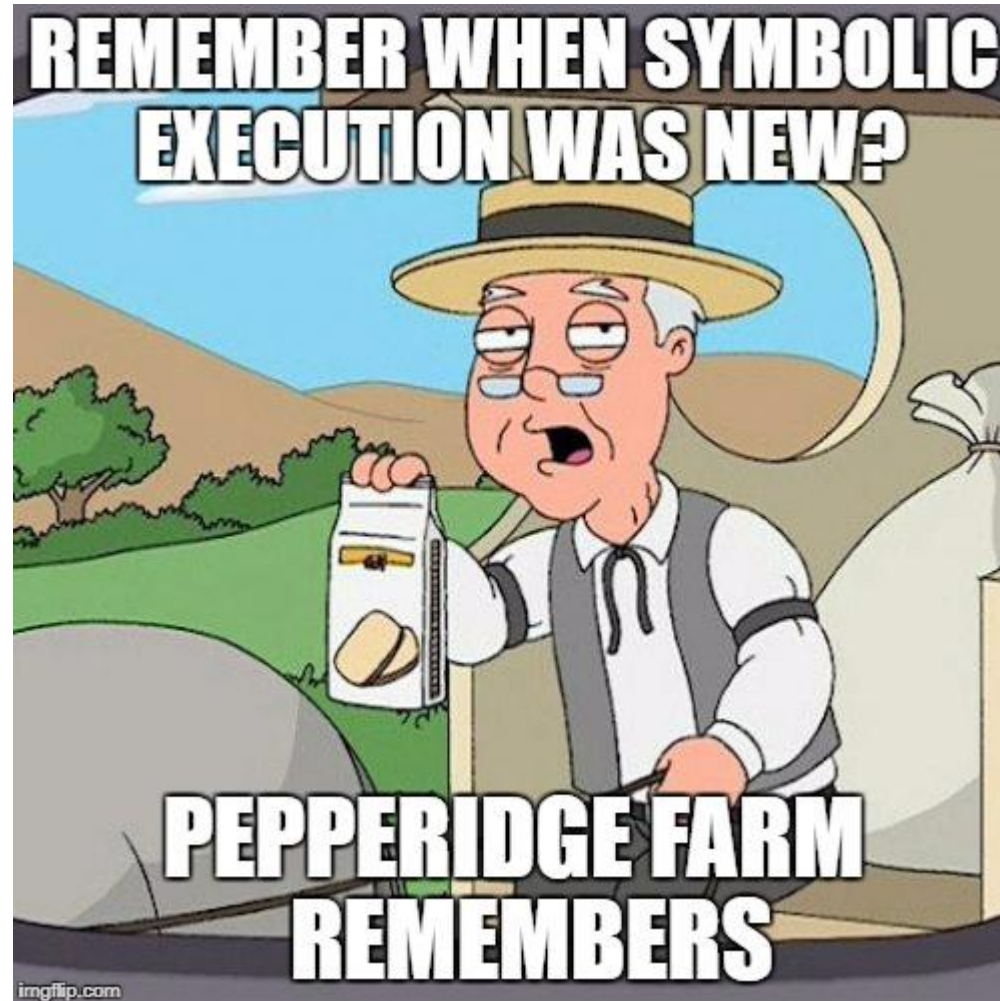


Symbolic execution is simply making some data 'symbolic' in the same way we do for formulas.

$$2x + 7 = 23$$



This is nothing new...



# #3

Symbolic execution is the 80%  
solution you **need**.



# Symbolic Execution



## **Remember we are trying to get answers fast**

- We aren't trying to work out everything
- Just the core details
- URL's, comms , persistence, lateral movement, second stage

**If we can get the detail sooner even if we don't fully understand it we still get what we (and others) need.**

# Symbolic Execution



## Concrete Execution

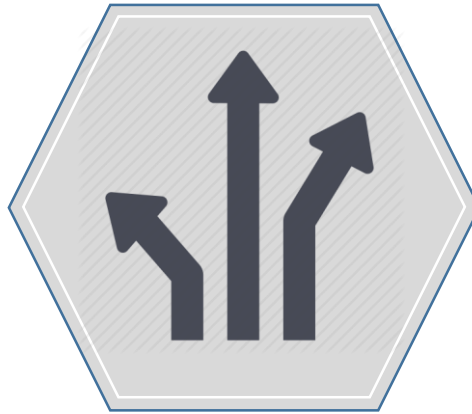
Standard execution of a program with defined variables (i.e.  $x=5$ )

# Symbolic Execution



## Concrete Execution

Standard execution of a program with defined variables (i.e.  $x=5$ )



## Symbolic Execution

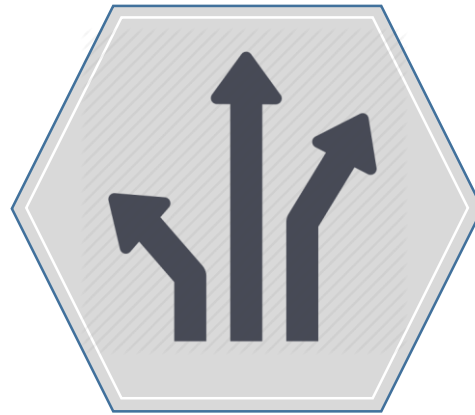
Execute a program through all possible execution paths, thus achieving all possible path conditions

# Symbolic Execution



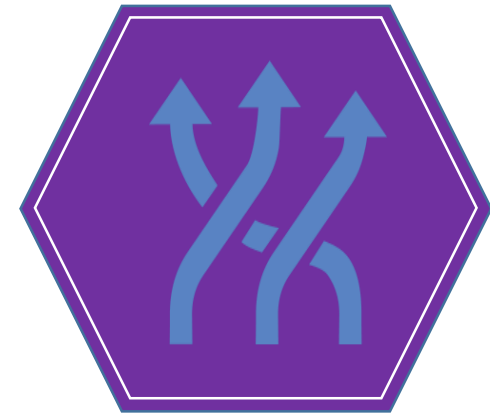
## Concrete Execution

Standard execution of a program with defined variables (i.e.  $x=5$ )



## Symbolic Execution

Execute a program through all possible execution paths, thus achieving all possible path conditions



## Concolic Execution

Concolic execution is a mix between CONCrete execution and symbOLIC execution, guiding it through a specific execution path

# Concrete Execution



## Concrete execution

- Perform actions based on the hard data that is passed in.
- X and y are hard values (i.e x =5, y=7)

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

# Symbolic Execution - Primer



## **Symbolic Execution**

Maintains a symbolic states of registers and part of memory at each program point.

- a table of symbolic registers states
- a map of symbolic memory states
- a global set of all symbolic references



# Dynamic forward symbolic execution



Dynamic forward symbolic execution builds a logical formula describing a program execution path

# Dynamic forward symbolic execution



Dynamic forward symbolic execution builds a logical formula describing a program execution path

Symbolic execution proceeds along both branches, by "forking" two paths. Each path gets assigned a copy of the program state at the branch instruction as well as a path constraint

$\lambda * 2 == 12$  for the then branch and  $\lambda * 2 != 12$

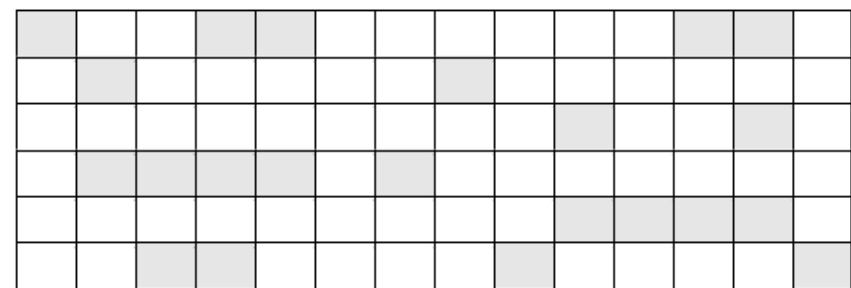
```
1 int f() {
2
3     y = read();
4     z = y * 2;
5     if (z == 12) {
6         fail();
7     } else {
8         printf("OK");
9     }
10 }
```

# Taint Analysis



The purpose of dynamic taint analysis is to track the information flow from the sources (usually user inputs) to the targets (such as control-flow value).

It is thus capable of analyzing which region of the memory and registers are controllable by user inputs.



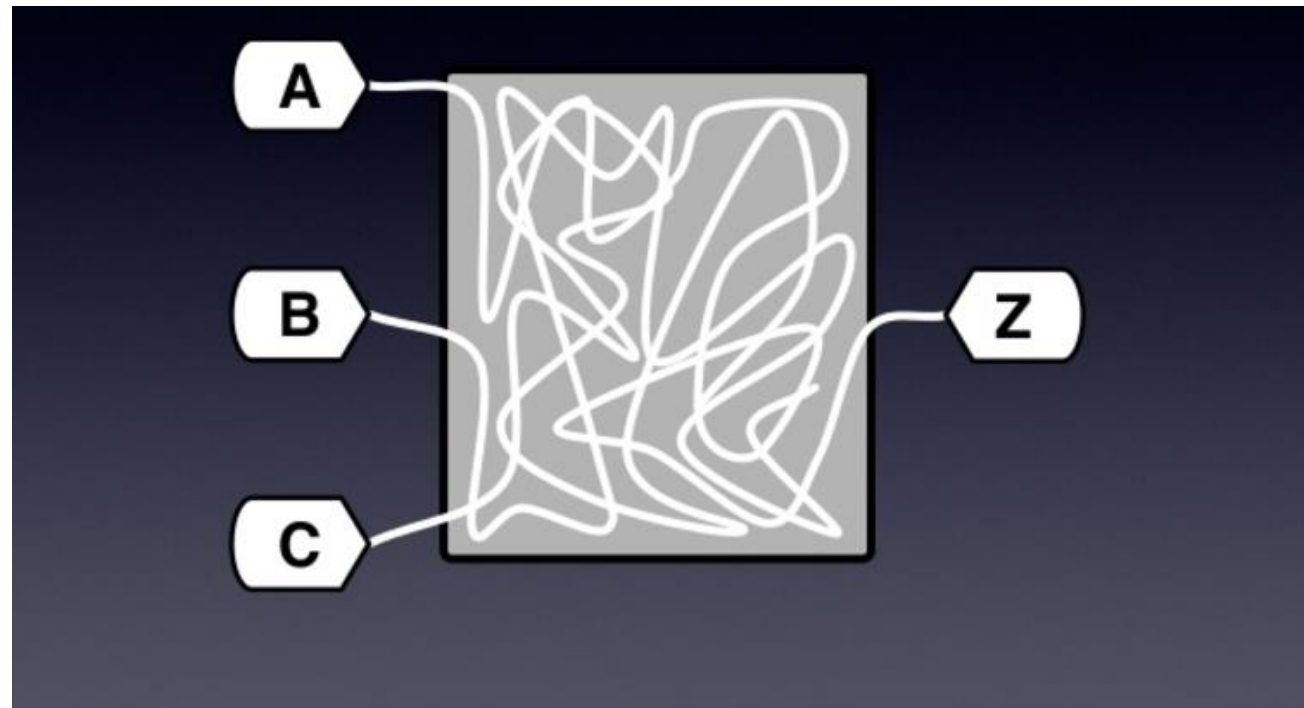
Memory area

- Byte which can be controlled
- Byte which cannot be controlled

# Taint Analysis

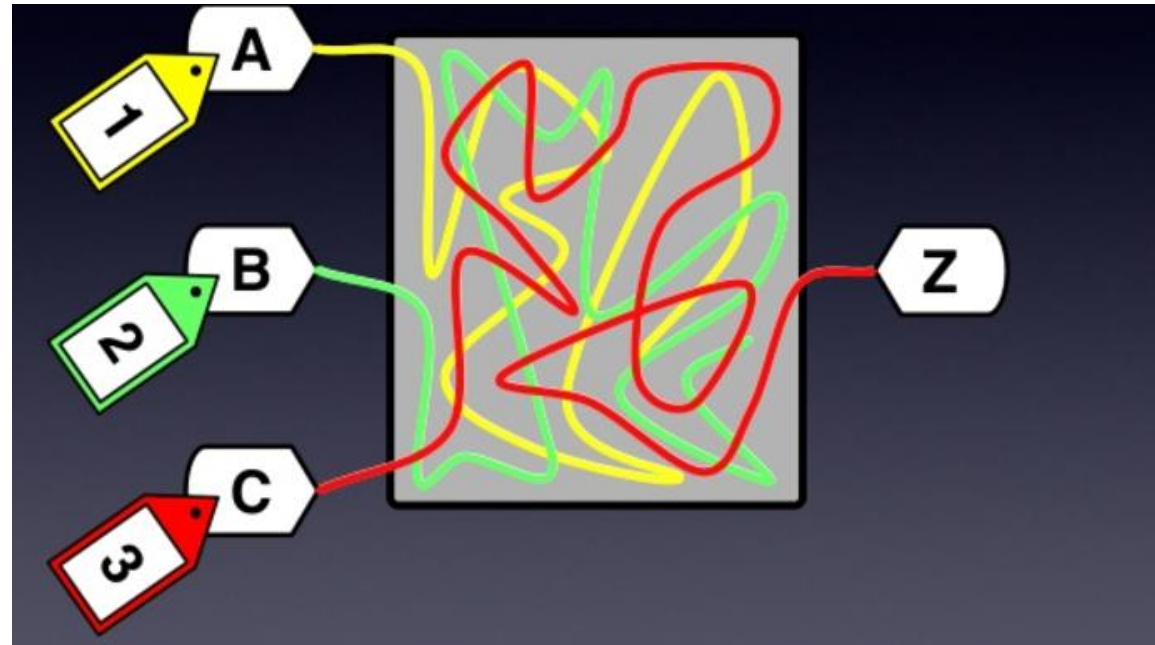


With this method it is possible to check the registers and the memory areas which can be controlled by the user when a crash occurs



# Taint Analysis

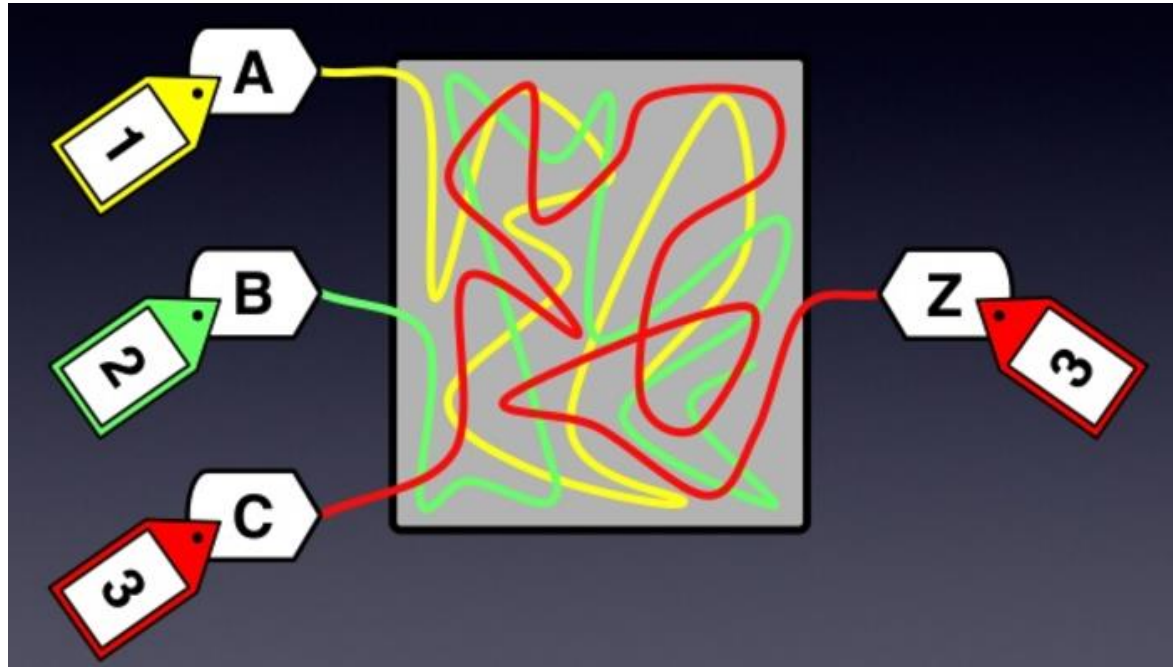
- ❓ With this method it is possible to check the registers and the memory areas which can be controlled by the user when a crash occurs



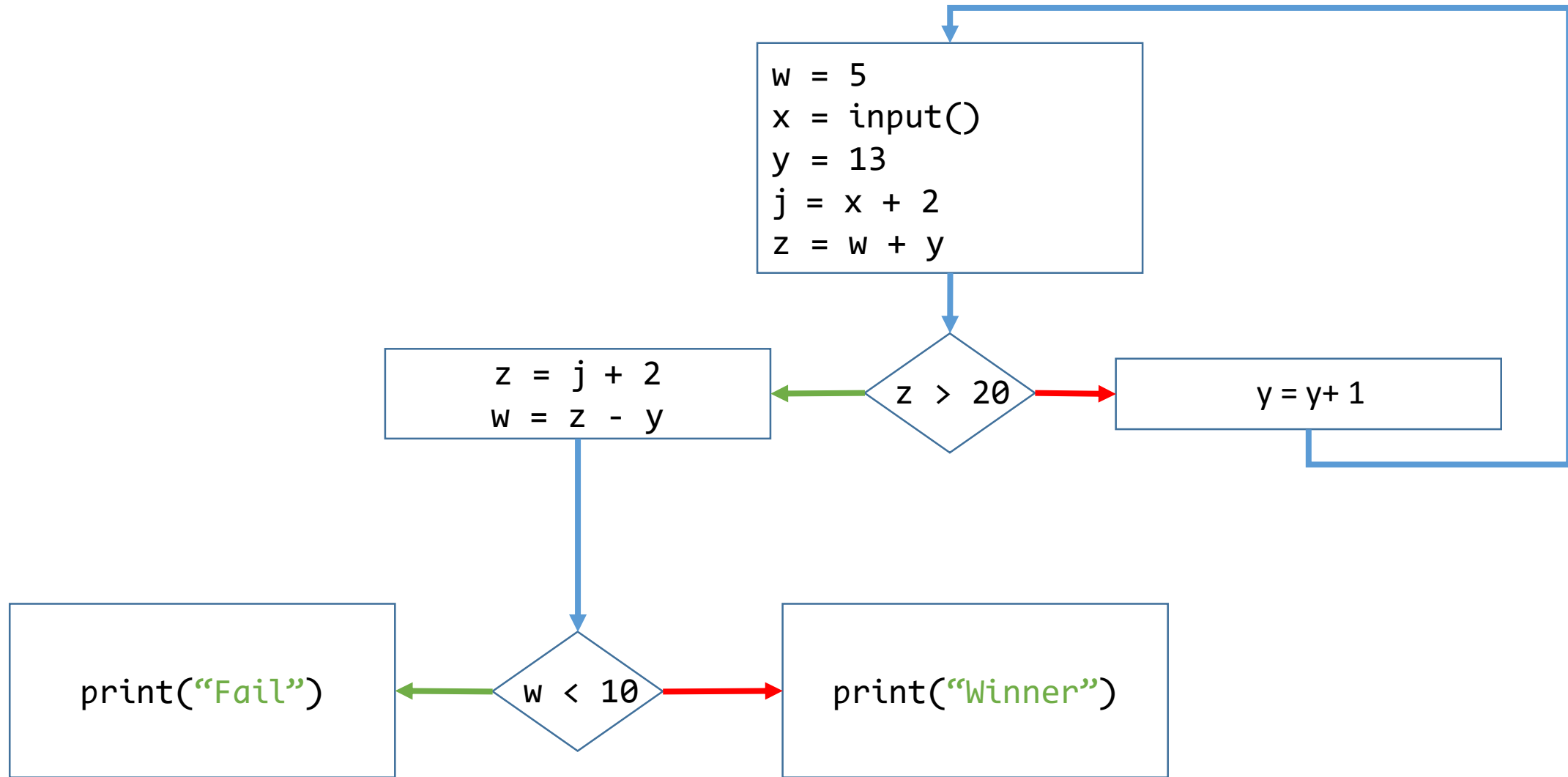
# Taint Analysis



With this method it is possible to check the registers and the memory areas which can be controlled by the user when a crash occurs



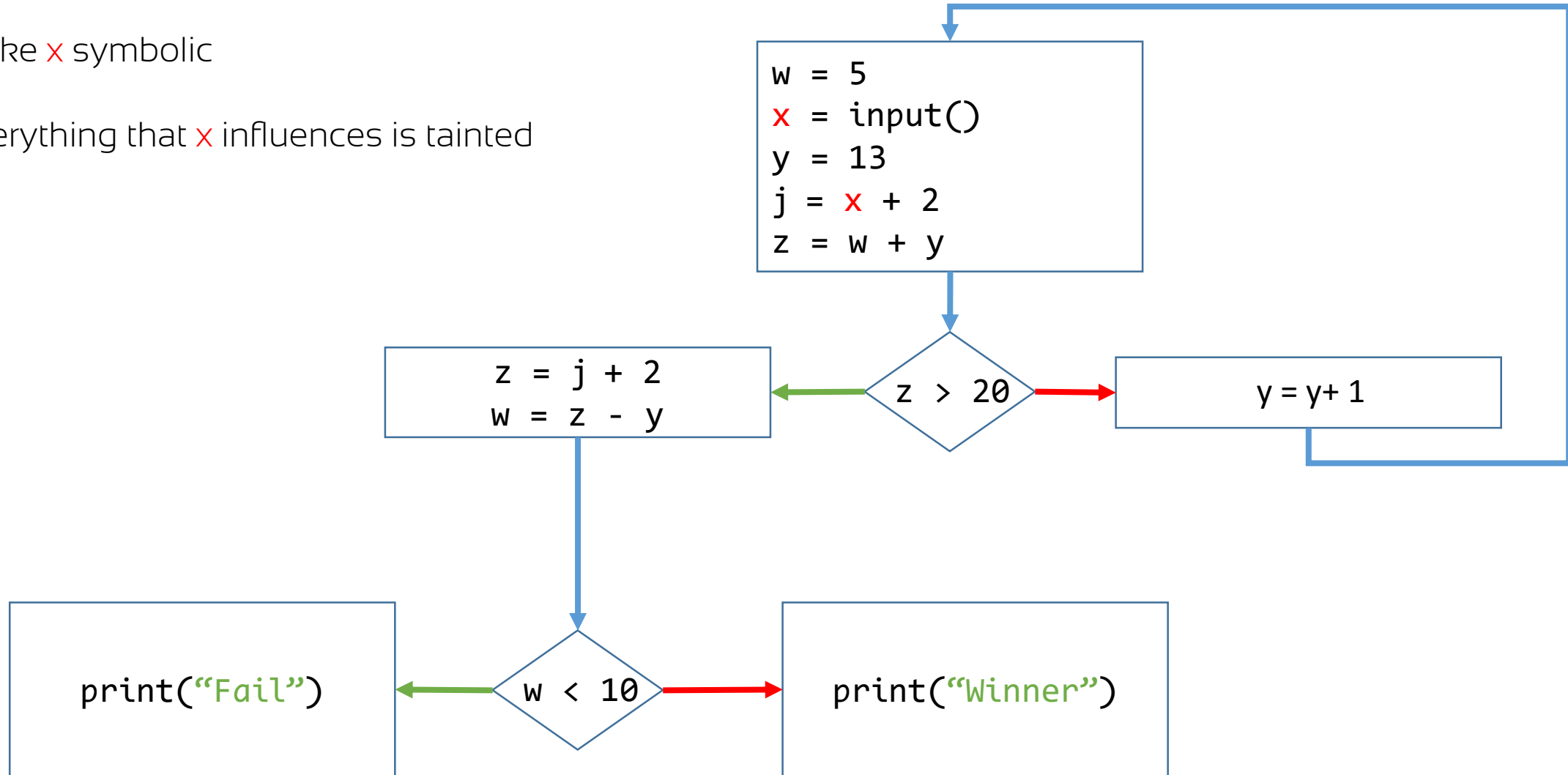
# Symbolic Execution – Taint Analysis



# Symbolic Execution – Taint Analysis

Make  $x$  symbolic

Everything that  $x$  influences is tainted

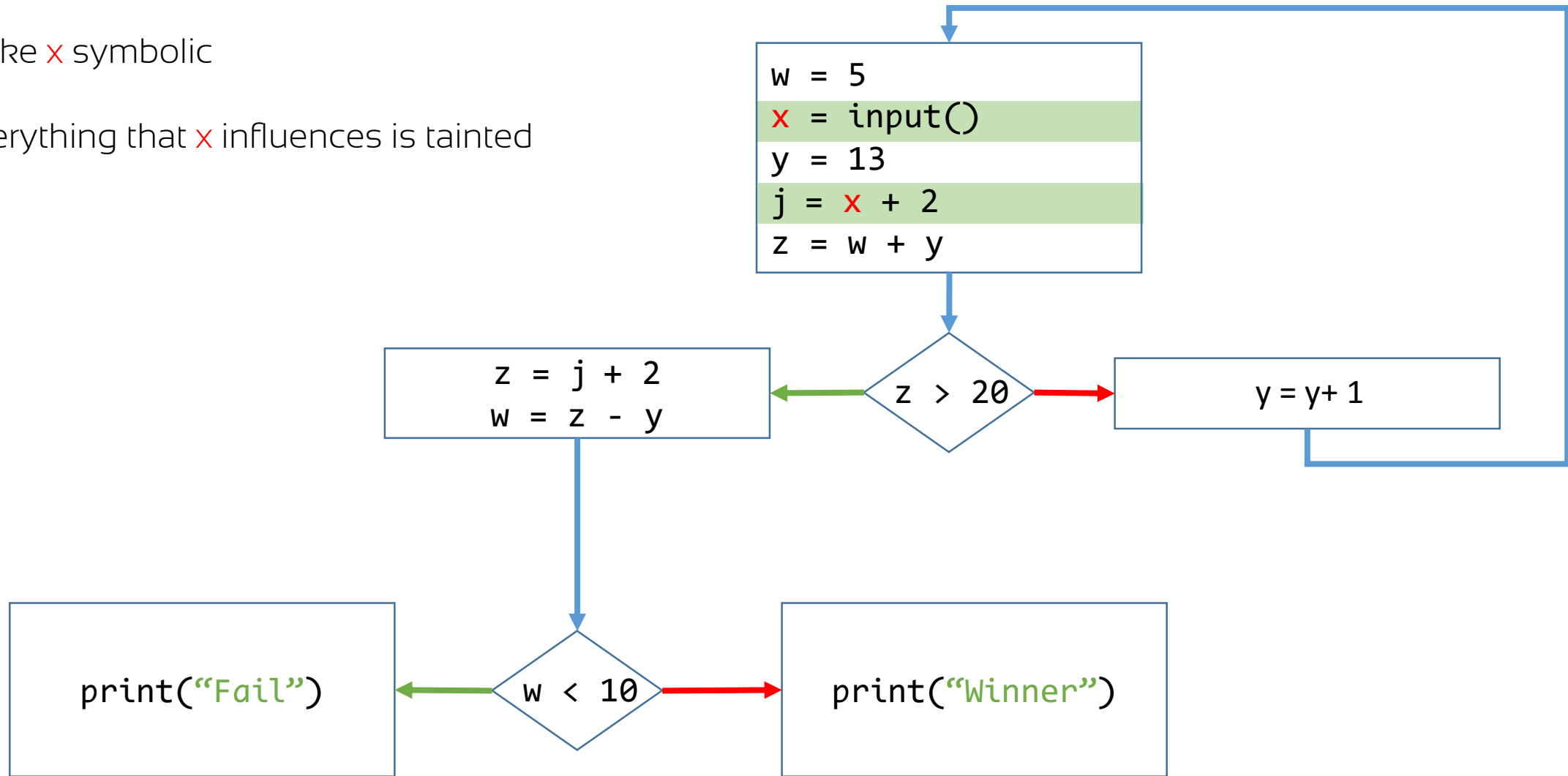




# Symbolic Execution – Taint Analysis

Make  $x$  symbolic

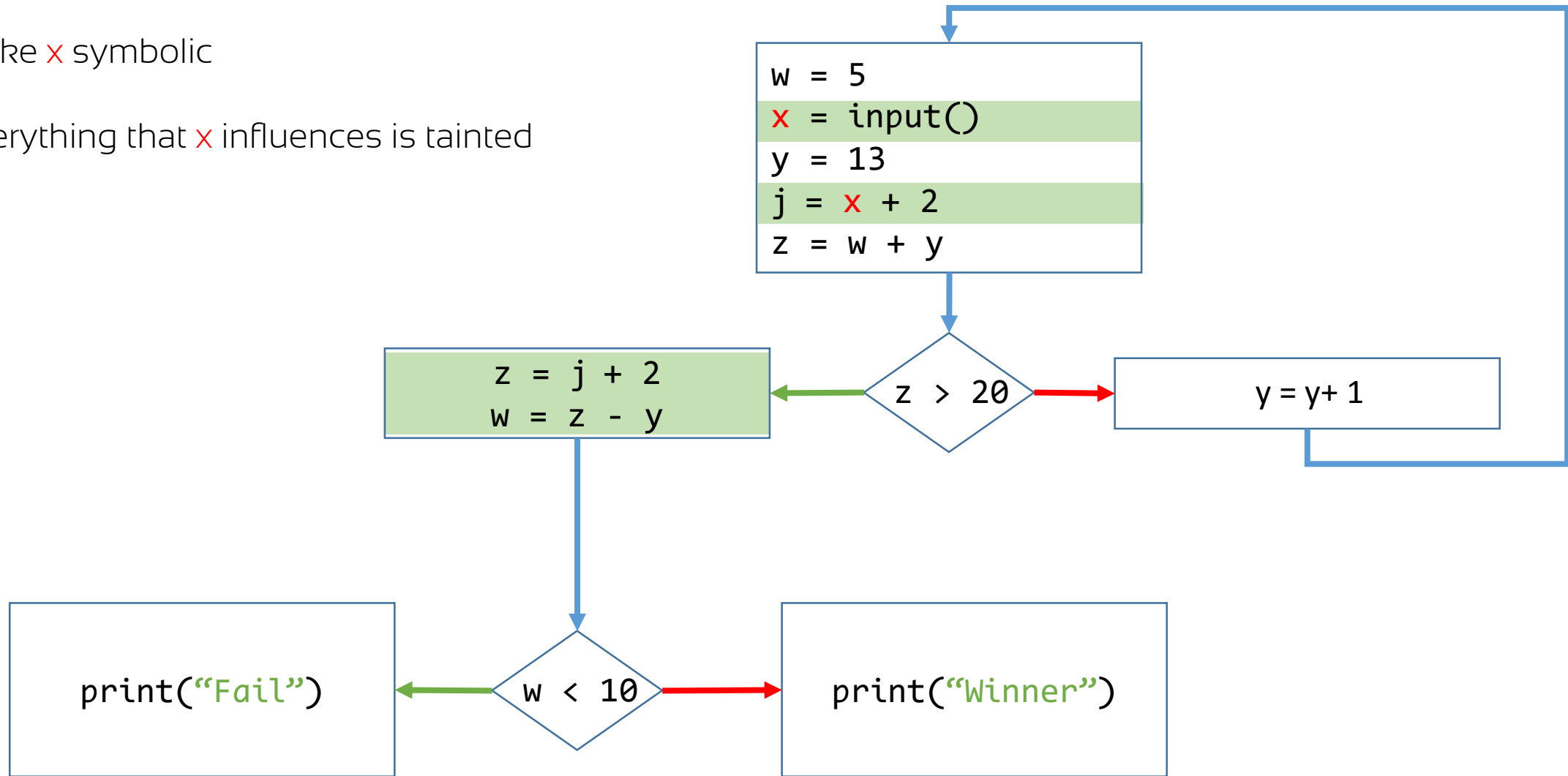
Everything that  $x$  influences is tainted



# Symbolic Execution – Taint Analysis

Make  $x$  symbolic

Everything that  $x$  influences is tainted

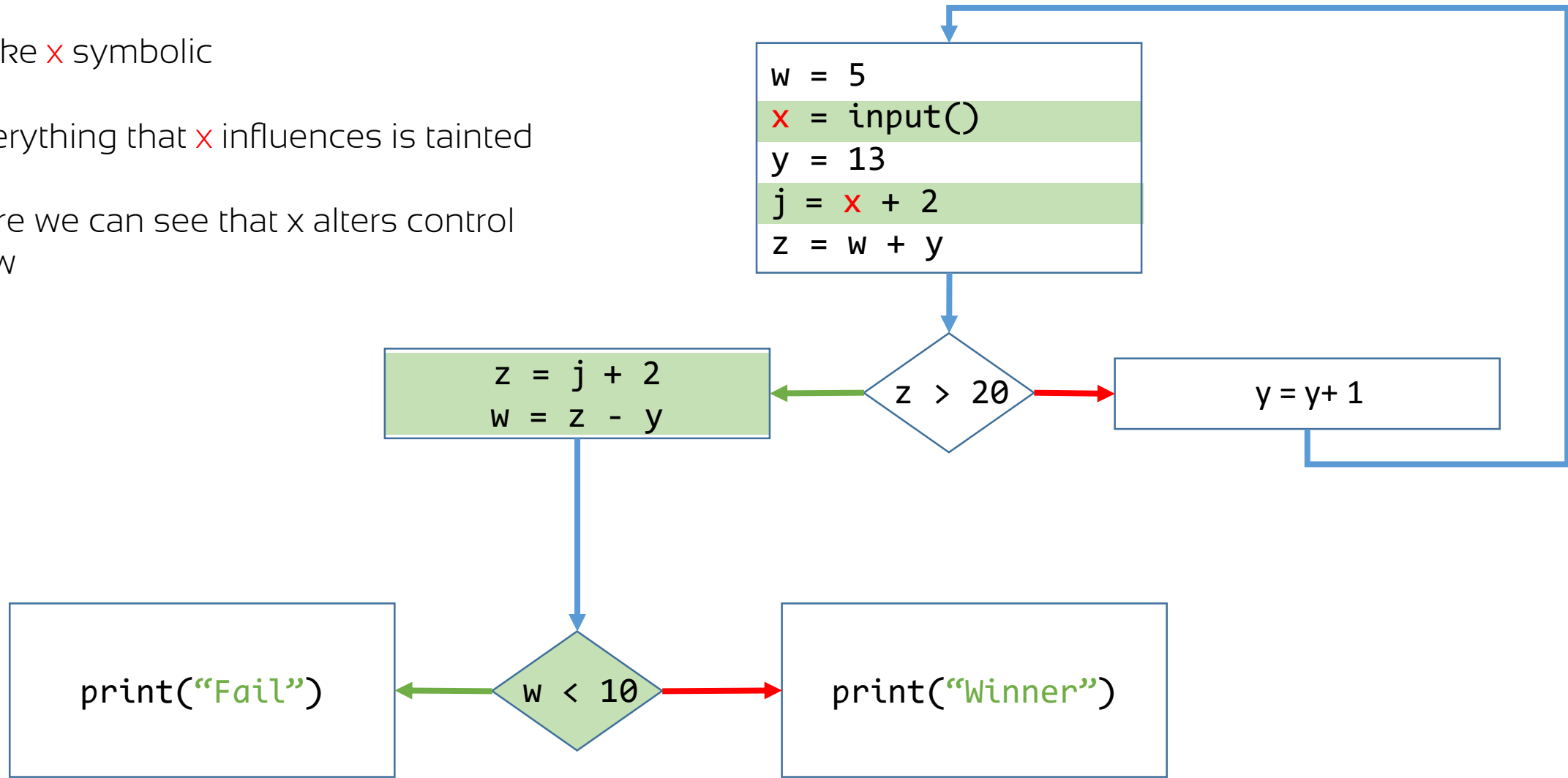


# Symbolic Execution – Taint Analysis

Make  $x$  symbolic

Everything that  $x$  influences is tainted

Here we can see that  $x$  alters control flow



# Symbolic Execution Limitations



## Path Explosion

- Symbolically executing all feasible program paths does not scale to large programs.

# Symbolic Execution Limitations



## Path Explosion

- Symbolically executing all feasible program paths does not scale to large programs.
- The number of feasible paths in a program grows exponentially with an increase in program size.

# Symbolic Execution Limitations



## Path Explosion

- Symbolically executing all feasible program paths does not scale to large programs.
- The number of feasible paths in a program grows exponentially with an increase in program size.



## Environment Interactions

- Symbolic execution that requires a file or user input will have consistency problems.

# Symbolic Execution Limitations



## Path Explosion

- Symbolically executing all feasible program paths does not scale to large programs.
- The number of feasible paths in a program grows exponentially with an increase in program size.



## Environment Interactions

- Symbolic execution that requires a file or user input will have consistency problems.
- File operations are implemented as system calls in the kernel, and are outside the control of the symbolic execution

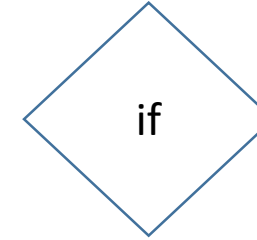
# Concolic Execution



## Concolic execution

- Performs symbolic execution along a concrete execution path
- Simplifying symbolic execution in the example

```
1 void f(int x, int y) {  
2     int z = 2*y;  
3     if (x == 100000) {  
4         if (x < z) {  
5             assert(0); /* error */  
6         }  
7     }  
8 }
```



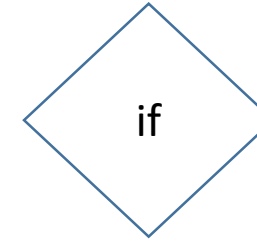


# Concolic Execution



## Concolic execution

- *Let*  $x = 1$   $y = 1$
- $Z = 2$



```
1 void f(int x, int y) {  
2     int z = 2*y;  
3     if (x == 100000) { ←  
4         if (x < z) {  
5             assert(0); /* error */  
6         }  
7     }  
8 }
```

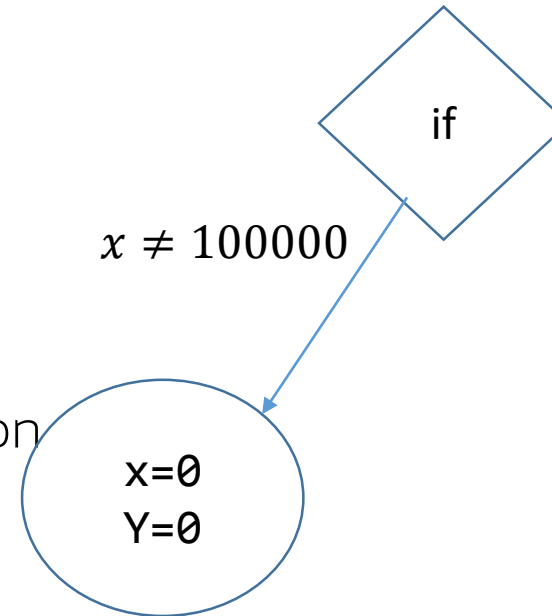
# Concolic Execution



## Concolic execution

- Let  $x = y = 1$
- $Z = 2$
- Line 3 fails because since  $1 \neq 100000$
- From the inequality we create a path condition

```
1 void f(int x, int y) {  
2     int z = 2*y;  
3     if (x == 100000) { ←  
4         if (x < z) {  
5             assert(0); /* error */  
6         }  
7     }  
8 }
```



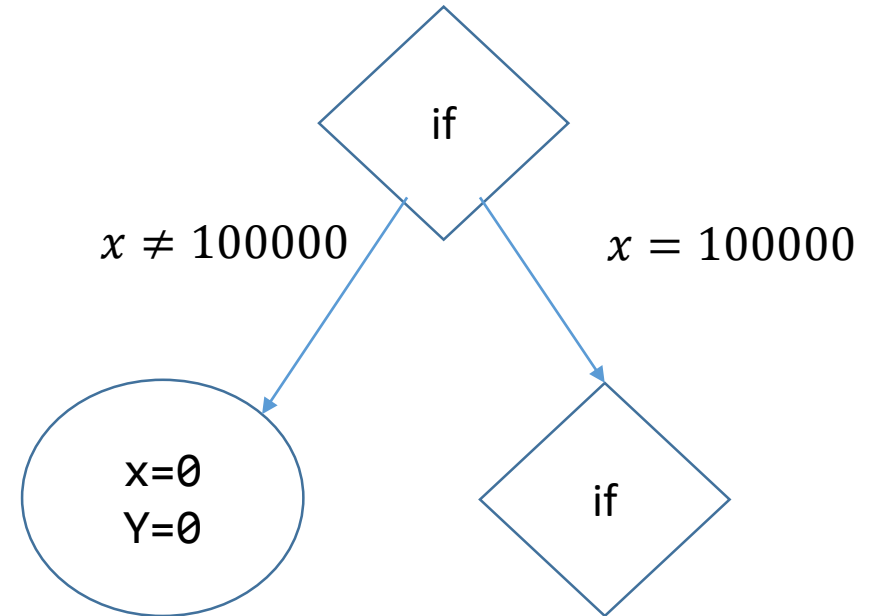
# Concolic Execution



## Concolic execution

- Try a different path, let  $x = 100000$

```
1 void f(int x, int y) {  
2     int z = 2*y;  
3     if (x == 100000) {  
4         if (x < z) {  
5             assert(0); /* error */  
6         }  
7     }  
8 }
```



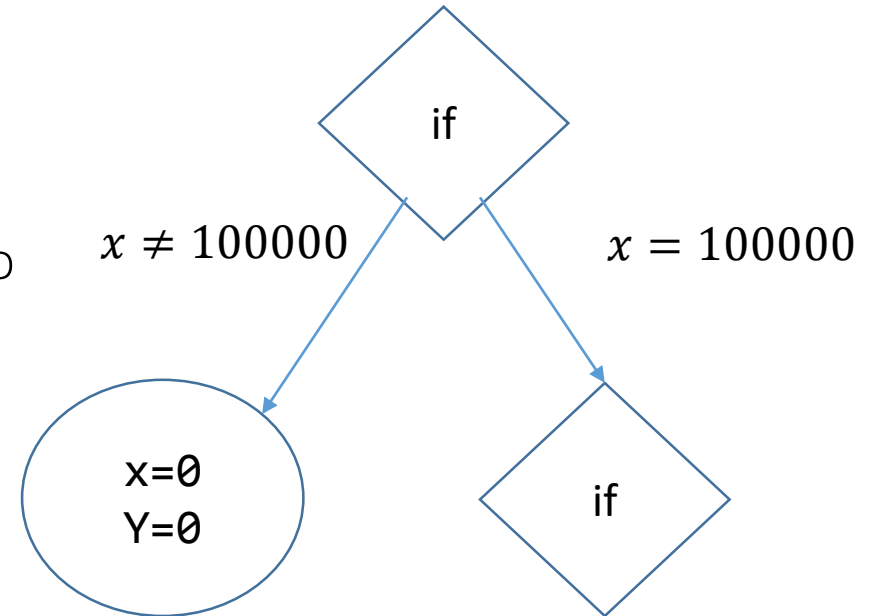
# Concolic Execution



## Concolic execution

- Try a different path, let  $x = 100000$
- Automated theorem prover is then invoked to find values for the input variables  $x$  and  $y$

```
1 void f(int x, int y) {  
2     int z = 2*y;  
3     if (x == 100000) {  
4         if (x < z) {  
5             assert(0); /* error */  
6         }  
7     }  
8 }
```



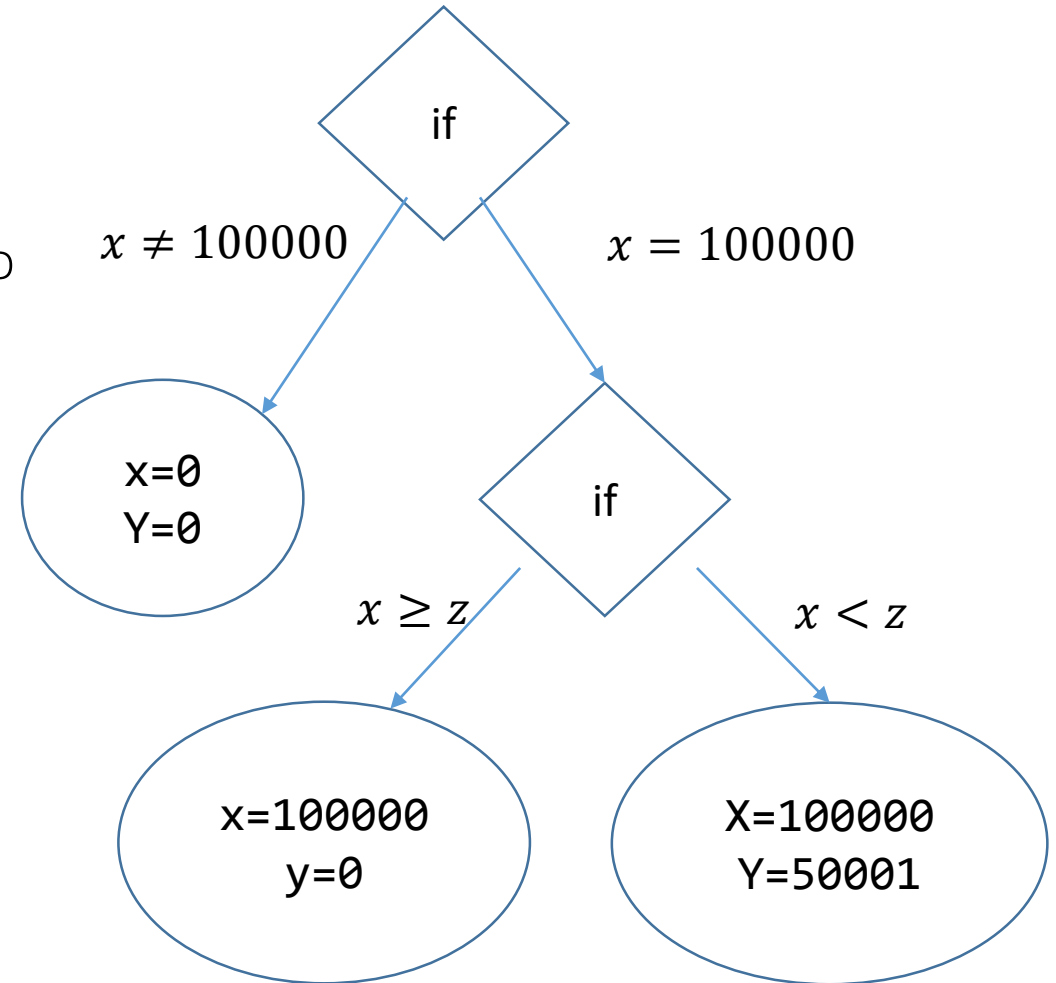
# Concolic Execution



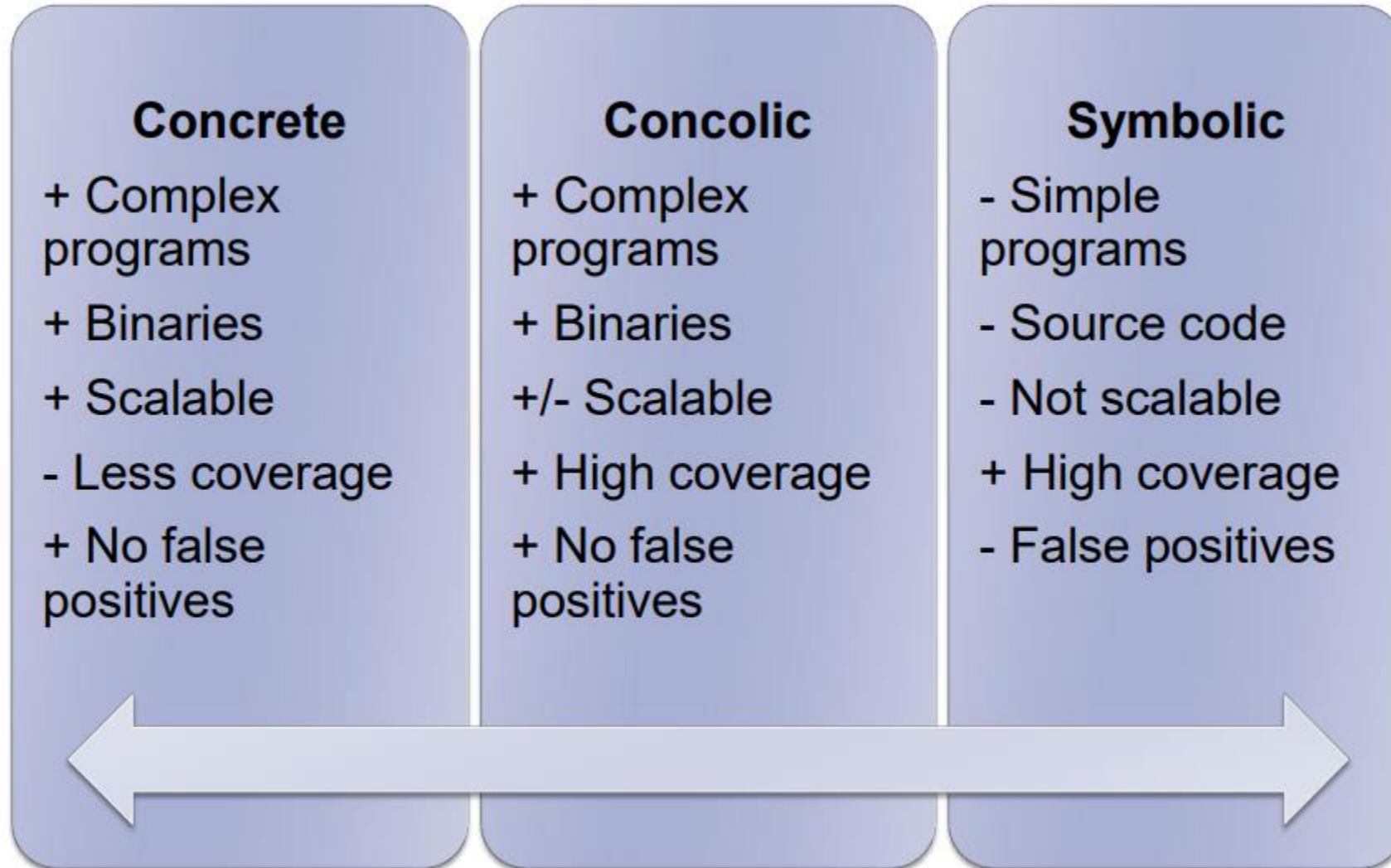
## Concolic execution

- Try a different path, let  $x = 100000$
- Automated theorem prover is then invoked to find values for the input variables  $x$  and  $y$
- A valid theorem might be  $x = 100000, y = 0$

```
1 void f(int x, int y) {  
2     int z = 2*y;  
3     if (x == 100000) {  
4         if (x < z) {  
5             assert(0); /* error */  
6         }  
7     }  
8 }
```



# Concolic Covering Middle Ground



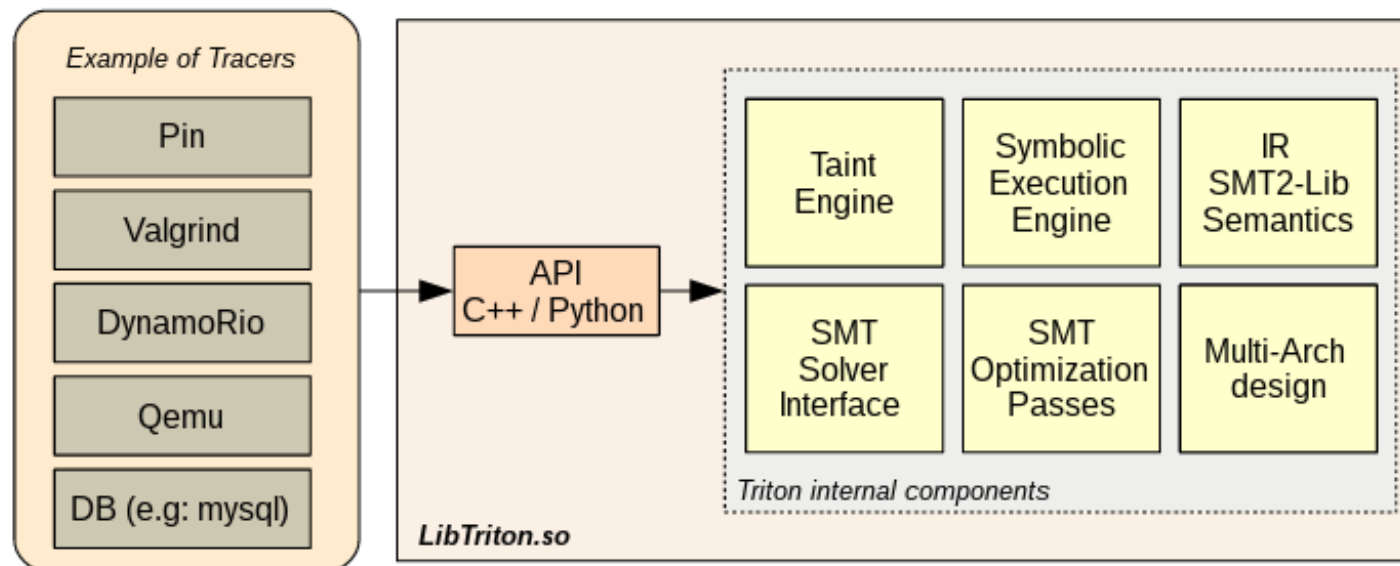
# Symbolic execution engines



## Triton

Triton is a dynamic binary analysis (DBA) framework. It provides internal components like a Dynamic Symbolic Execution (DSE) engine, a Taint Engine, AST representations of the x86 and the x86-64 instructions set semantics

<https://triton.quarkslab.com/>



# Symbolic execution engines



## **Angr**

Python framework for analyzing binaries. It combines both static and dynamic symbolic ("concolic") analysis, making it applicable to a variety of tasks.

<http://angr.io/>



If this is how you feel your in luck!



# #4

Sometimes all you need is a  
little more ponce in your life.



# Ponce



IDA's 2016 plugin contest winner

Built on the triton engine

**TRILON**  
Dynamic Binary Analysis



<https://github.com/illera88/Ponce>

# Ponce



IDA's 2016 plugin contest winner

Built on the triton engine



Supports both x86 and x64 binaries

Cross platform, Windows, Linux  
and OSX natively



<https://github.com/illera88/Ponce>

# Ponce

Just a quick insight into ponce.



## **C2 Channel**

Symbolize the data returned from malware. Find where the data hits the C2 switch and use the SMT solver to determine the control byte

# Ponce

Just a quick insight into ponce.



## **C2 Channel**

Symbolize the data returned from malware. Find where the data hits the C2 switch and use the SMT solver to determine the control byte



## **Finding exploits.**

Run taint analysis on the data returned from a Recv() or a InternetReadfile() to see what blocks of data it touches. Platform to build exploits

# Ponce

Just a quick insight into ponce.



## **C2 Channel**

Symbolize the data returned from malware. Find where the data hits the C2 switch and use the SMT solver to determine the control byte



## **Finding exploits.**

Run taint analysis on the data returned from a Recv() or a InternetReadfile() to see what blocks of data it touches. Platform to build exploits



## **Rapid analysis**

Run and negate conditions on known buffers to find what other commands are supported.

# Ponce



Simple to install (IDA 6.8-6.9):

Download the plugins and copy to your IDAPro plugins folder

```
$ cp Ponce_x64_IDA68_win.p64 <IDA_PRO>/plugins
```

```
$ cp Ponce_x64_IDA68_win.plw <IDA_PRO>/plugins
```





# Ponce Example - Crackme

**OpenRCE Articles**

There are 31,016 total registered users.

Recently Created Topics

- Ultimate Hacking Cha... Jun/21
- CreateMuleX May/31
- let 'IDAPython' impo... Sep/24
- set 'IDAPython' as L... Sep/24
- GuessType return une... Sep/20
- About retrieving the... Sep/07
- How to find specific... Aug/15
- How to get data depe... Jul/07
- Identify RVA data in... May/09
- Immunity Debugger Re... Aug/03

Recent Forum Posts

- How to find specific... hackgrati
- Problem with olydbg sk3dow
- How can I write oly... sk3dow
- New LoadMAP plugin v... mefsto...
- Intel pin in loaded... djmemo
- OOP\_RE tool available? BlfckmIn
- OOP\_RE tool available? van7nu
- Should binaries be n... Kollsar
- Problem with olydbg null42
- lindrapolne immu... skyrock

Recent Blog Entries

- nico Mar/22
- Android Application Reversing
- haisten Mar/14
- Breaking IonCUBE VM
- oleavr Oct/24
- Anatomy of a code tracer
- hasherezade Sep/24
- IAT Patcher - new tool for ...
- oleavr Aug/27
- CryptoShark: code tracer ba... More ...

Recent Blog Comments

- nico on: Mar/22
- IAT Patcher - new tool for ...
- djmemo on: Nov/17
- Kernel debugger vs user mod...
- aceli on: Nov/14
- Kernel debugger vs user mod...
- pedram on: Dec/21
- frida.github.io: scribble...
- capadlemian on: Jun/19

**Memoryze Memory Forensics Tool** Created: Wednesday, November 26 2008 20:06.40 CST # Views: 63154

Author: # Views: 63154

The goal of this article is to demonstrate how simple malware analysis can be using Memoryze and some good old fashion common sense. Readers should have some knowledge of how malware works, and be somewhat familiar with [Memoryze](#). A good place to familiarize yourself with Memoryze is the user guide included in the installer.

Memoryze is designed to aid in memory analysis in incident response scenarios. However, it has many useful features that can be utilized when doing malware analysis. Memoryze is special in that it does not rely on API calls. Instead Memoryze parses the operating systems' internal structures to determine for itself what the operating system and its running processes and drivers are doing.

[Full Article ...](#) [Printer Friendly ...](#) [Write Comment](#) | [View Complete Comments](#)

Username	Comment Excerpt	Date
frabits	good job !tx	Sunday, December 7 2014 07:41 08 CST
crlx	good job !tx	Wednesday, August 8 2012 21:33.24 CDT
zazo010	I'm waita for next lessons. Nice work, thanks!	Sunday, August 14 2011 04:26.00 CDT
easescob	Very nice work, thank!	Monday, June 27 2011 19:28.08 CDT
stolurmer	For folks coming late to the party like me, I w...	Monday, October 25 2010 10:40.36 CDT

**The Molecular Virology of Lexotan32: Metamorphism Illustrated** Created: Thursday, August 16 2007 16:58.00 CDT # Views: 51722

Author: # Views: 51722

This paper is a direct descendent of my previous one regarding the metamorphic engine of the [W32.Evol](#) virus. I advise you to take a look at it before reading this one, or at least be acquainted with the subject of metamorphism. The focus of this paper is the special engine of the [Lexotan32](#) virus.

The virus was released in 29A#6 Virus Magazine in 2002, the Annus Mirabilis of metamorphic viruses. The virus was created by the prolific VX coder, Vecna, and was one of the last complex creations of this kind. I could further elaborate on the genealogy of this virus, but I think it is sufficient to say that this virus is a culmination of many of the techniques developed throughout the author's career.

[Full Article ...](#) [Printer Friendly ...](#) [Write Comment](#) | [View Complete Comments](#)

Username	Comment Excerpt	Date
live2skull	amazing article :)	Sunday, May 8 2016 00:28.54 CDT
redbone	good information ...	Tuesday, July 12 2011 02:56.10 CDT
ignice	cool :)	Saturday, June 18 2011 03:18.50 CDT
lazyworm	very nice!! need it.	Wednesday, June 30 2010 20:25.49 CDT
m4dnut	it's so cool-I thnaks for your efforts. i alway...	Wednesday, July 9 2008 20:32.17 CDT

**Defeating HyperUnpackMe2 With an IDA Processor Module** Created: Thursday, February 22 2007 19:21.58 CST # Views: 69496

Author: # Views: 69496

This article is about breaking modern executable protectors. The target, a crackme known as [HyperUnpackMe2](#), is modern in the sense that it does not follow the standard packer model of yesteryear wherein the contents of the executable in memory, minus the import information, are eventually restored to their original forms.

Modern protectors mutilate the original code section, use virtual machines operating upon polymorphic bytecode languages to slow reverse engineering, and take active measures to frustrate attempts to dump the process. Meanwhile, the complexity of the import protections and the amount of anti-debugging measures has steadily increased.

This article dissects such a protector and offers a static unpacker through the use of an IDA processor module and a custom plugin. The commented IDB files and the processor module source code are included. In addition, an appendix covers IDA processor module construction. In short, this article is an exercise in overkill.

[Full Article ...](#) [Printer Friendly ...](#) [Write Comment](#) | [View Complete Comments](#)

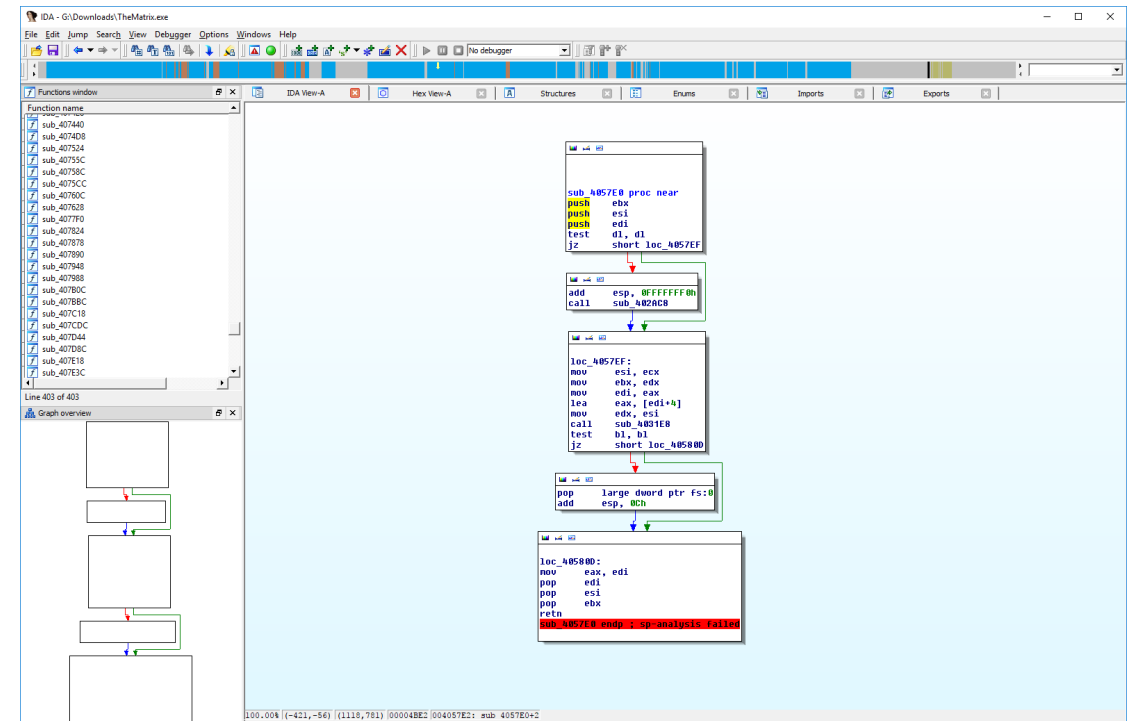
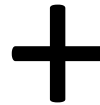
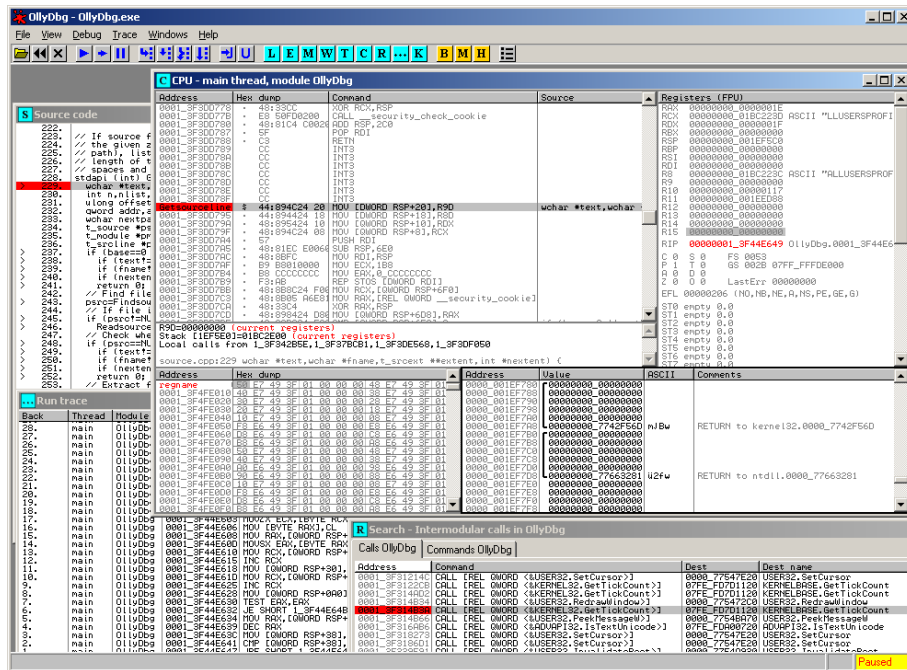
Username	Comment Excerpt	Date
ndaj3	RollfRolls: Thank you for Writing an Great tuto...	Friday, September 4 2009 01:25.09 CDT
eirc	Wow thanks a lot!	Saturday, October 11 2008 05:00 28 CDT
h4x0r	comprehensive analysis, thanks for those no...	Tuesday, May 15 2007 04:15.56 CDT
PoincareLai	good analysis. expecting RollfRolls to wr...	Wednesday, April 4 2007 06:28.45 CDT

# Ponce Example - Crackme



# Ponce Example - Crackme

Normally load your two favorite tools



# Ponce Example - Crackme

OR

Maybe use Ponce?

# Ponce Example

## Standard crackme.exe

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\cyber>crackme.exe P@ssword!
```

```
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
    int i;
    int hash = 0xABCD;

    for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];
    return hash;
}

int main(int ac, char **av)
{
    int ret;
    if (ac != 2)
        return -1;

    ret = check(av[1]);
    if (ret == 0xad6d)
        printf("Win\n");
    else
        printf("fail\n");
    return 0;
}
```

# Ponce Example

## Standard crackme.exe

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\cyber>crackme.exe P@ssword!
```

## Hardcoded serial to check against

```
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
    int i;
    int hash = 0xABCD;

    for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];
    return hash;
}

int main(int ac, char **av)
{
    int ret;
    if (ac != 2)
        return -1;

    ret = check(av[1]);
    if (ret == 0xad6d)
        printf("Win\n");
    else
        printf("fail\n");
    return 0;
}
```

# Ponce Example

## Standard crackme.exe

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\cyber>crackme.exe P@ssword!
```

## Hardcoded serial to check against

Takes the input and XOR's it against the serial.

```
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
    int i;
    int hash = 0xABCD;

    for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];
    return hash;
}

int main(int ac, char **av)
{
    int ret;
    if (ac != 2)
        return -1;

    ret = check(av[1]);
    if (ret == 0xad6d)
        printf("Win\n");
    else
        printf("fail\n");
    return 0;
}
```

# Ponce Example

Standard crackme.exe

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\cyber>crackme.exe P@ssword!
```

Hardcoded serial to check against

Takes the input and XOR's it against the serial.

If  $0xABCD + (\text{serial XOR input}) == 0xAD6D$  Win!

```
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
    int i;
    int hash = 0xABCD;

    for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];
    return hash;
}

int main(int ac, char **av)
{
    int ret;
    if (ac != 2)
        return -1;

    ret = check(av[1]);
    if (ret == 0xad6d)
        printf("Win\n");
    else
        printf("fail\n");
    return 0;
}
```



# Ponce Example

Standard crackme.exe

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\cyber>crackme.exe P@ssword!
```

Hardcoded serial to check against

Takes the input and XOR's it against the serial.

If  $0xABCD + (\text{serial XOR input}) == 0xAD6D$  Win!

Even with the source what is the key?

```
#include <stdio.h>
#include <stdlib.h>

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
    int i;
    int hash = 0xABCD;

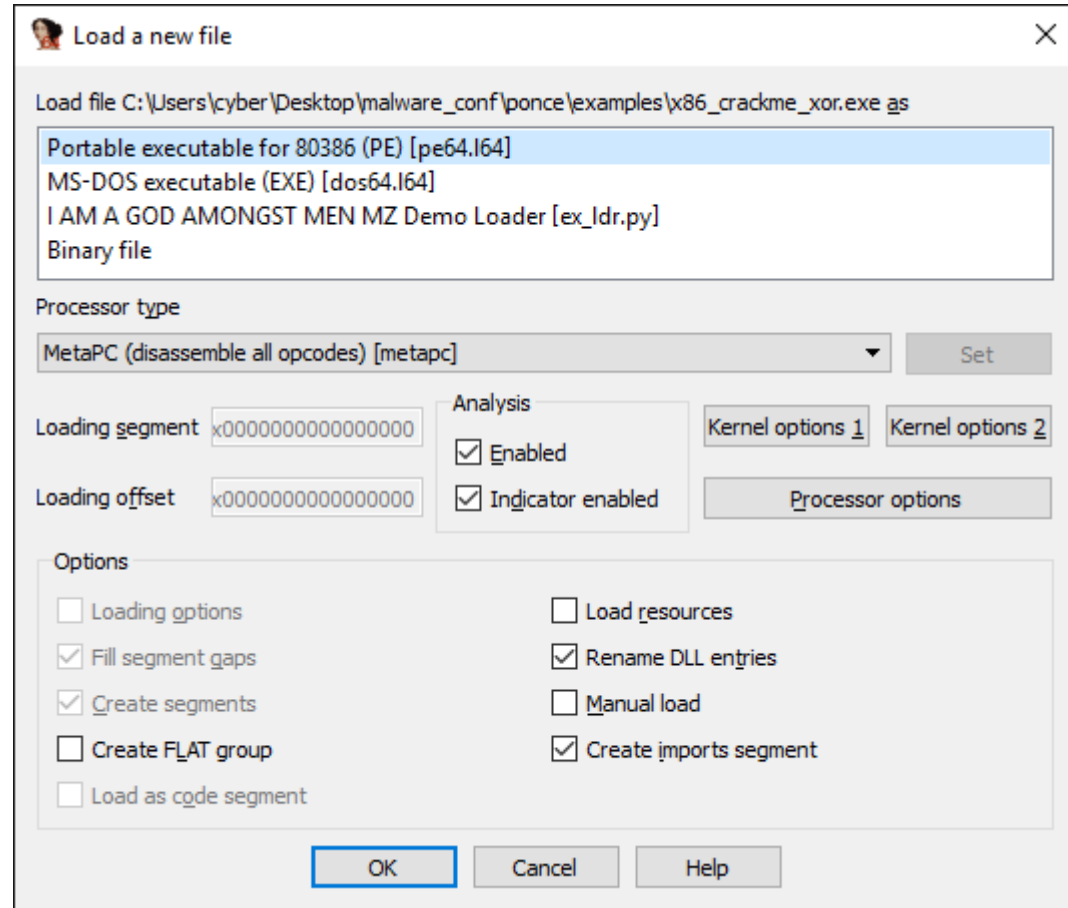
    for (i = 0; ptr[i]; i++)
        hash += ptr[i] ^ serial[i % 5];
    return hash;
}

int main(int ac, char **av)
{
    int ret;
    if (ac != 2)
        return -1;

    ret = check(av[1]);
    if (ret == 0xad6d)
        printf("Win\n");
    else
        printf("fail\n");
    return 0;
}
```

# Ponce Example

Load the crackme.exe file



# Ponce Example

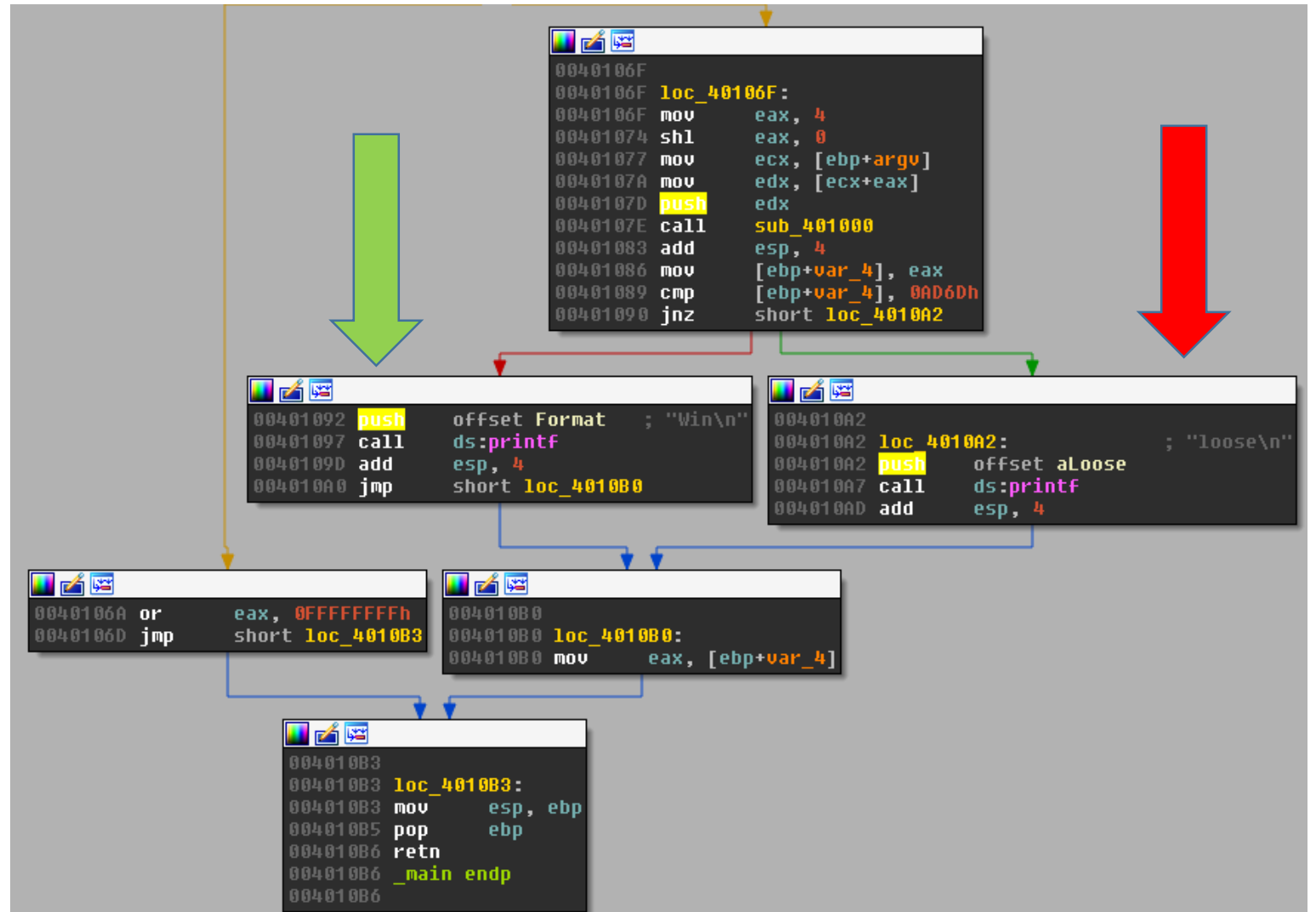
Load the crackme.exe file

The screenshot displays the IDA Pro interface for the `_main` function. The left sidebar shows the function list, with `_main` selected. The main window shows the disassembly of the function, with the control flow graph (CFG) visible. The CFG consists of several basic blocks connected by control flow edges. The initial block (00401060) pushes `ebp`, moves `esp` to `ebp`, pushes `ecx`, and compares `[ebp+argc]` with `2`. If `jz`, it jumps to `loc_40106F`. The `loc_40106F` block moves `eax` to `4`, shifts it left by `0`, moves `ecx` to `[ebp+argu]`, moves `edx` to `[ecx+eax]`, pushes `edx`, calls `sub_401000`, adds `4` to `esp`, moves `eax` to `[ebp+var_4]`, compares `[ebp+var_4]` with `0AD6Dh`, and jumps to `loc_4010A2` if `jnz`. The `loc_4010A2` block pushes `offset aLoose` and calls `ds:printf`, then adds `4` to `esp`. The `loc_401080` block pushes `offset Format` and calls `ds:printf`, then adds `4` to `esp` and jumps to `loc_4010B3`. The `loc_401083` block performs `or` on `eax` with `0FFFFFFh` and jumps to `loc_4010B3`. The `loc_4010B3` block moves `esp` to `ebp`, pops `ebp`, and returns. The `loc_401080` block also moves `eax` to `[ebp+var_4]`. The status bar at the bottom indicates the current instruction is `100.00% (-54,162) (539,0) 00000460 00401060: _main (Synchronized with Hex View-1)`. The output window at the bottom shows the message: `THE INITIAL AUTOANALYSIS HAS BEEN FINISHED. 403018: using guessed type char *off_403018;`

# Ponce Example

Win condition

Fail condition



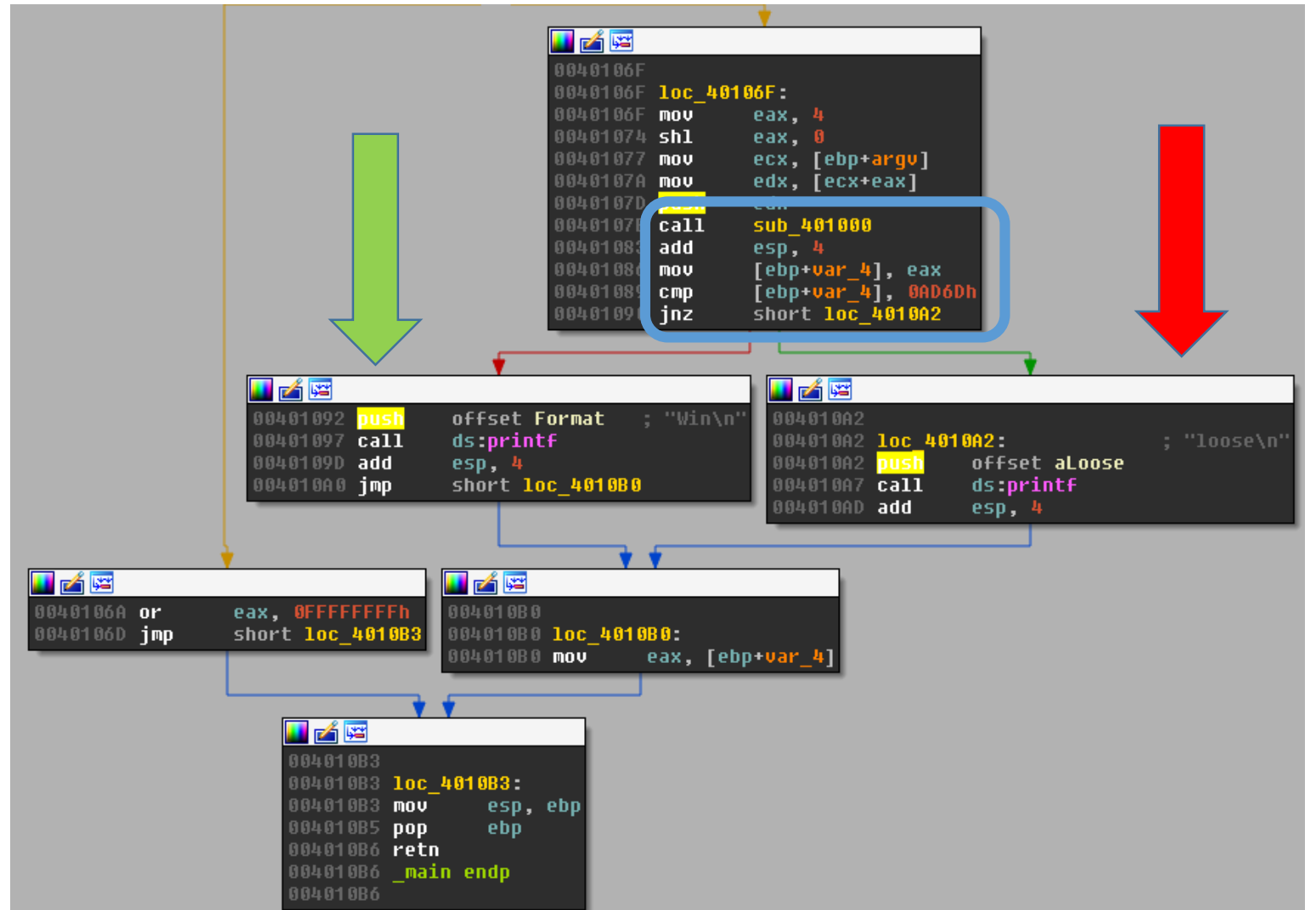
# Ponce Example

Win condition

Fail condition

Call for check\_function

return value into eax  
jnz short loc\_4010A2



# Ponce Example

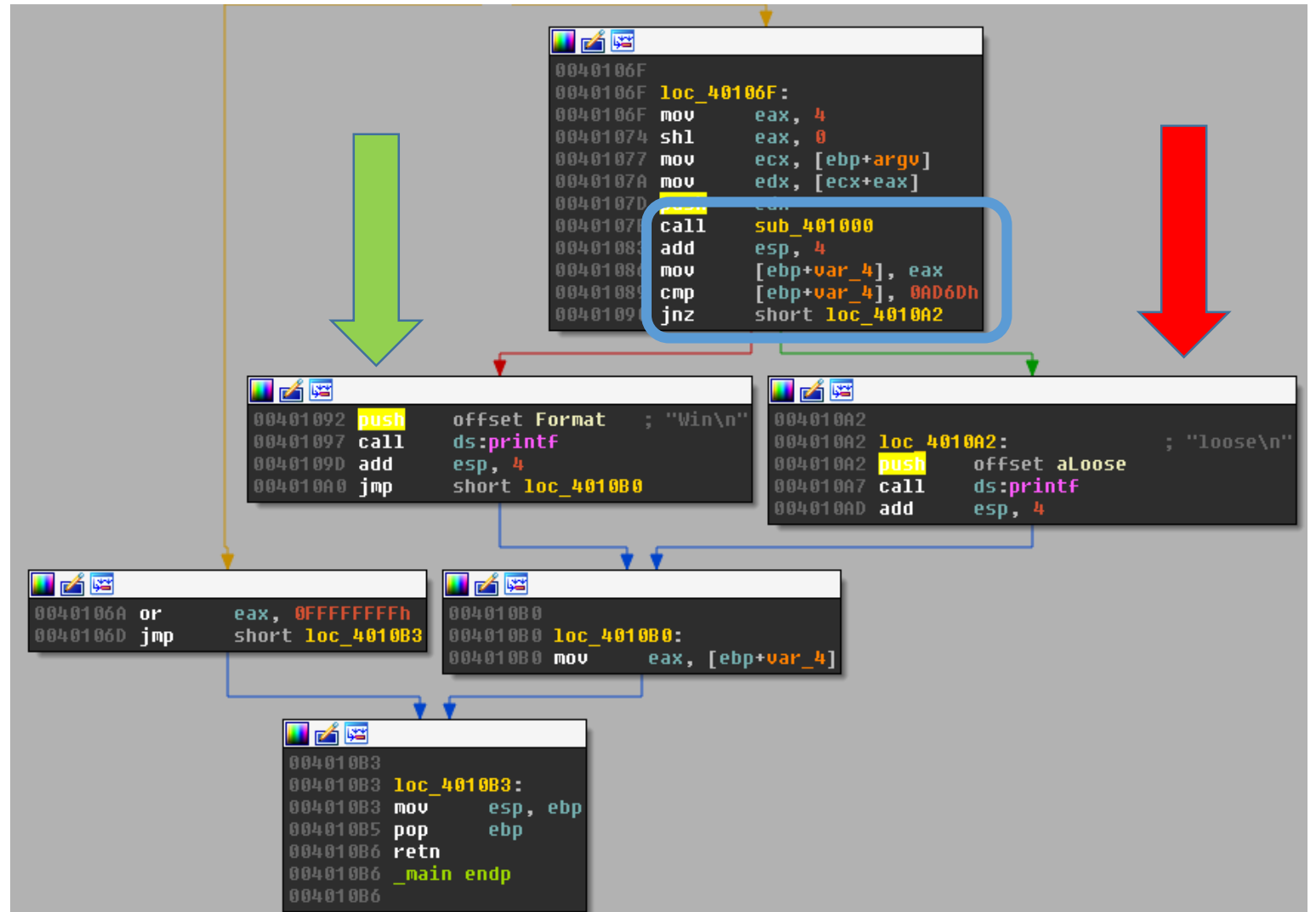
Win condition

Fail condition

Call for check\_function

return value into eax  
jnz short loc\_4010A2

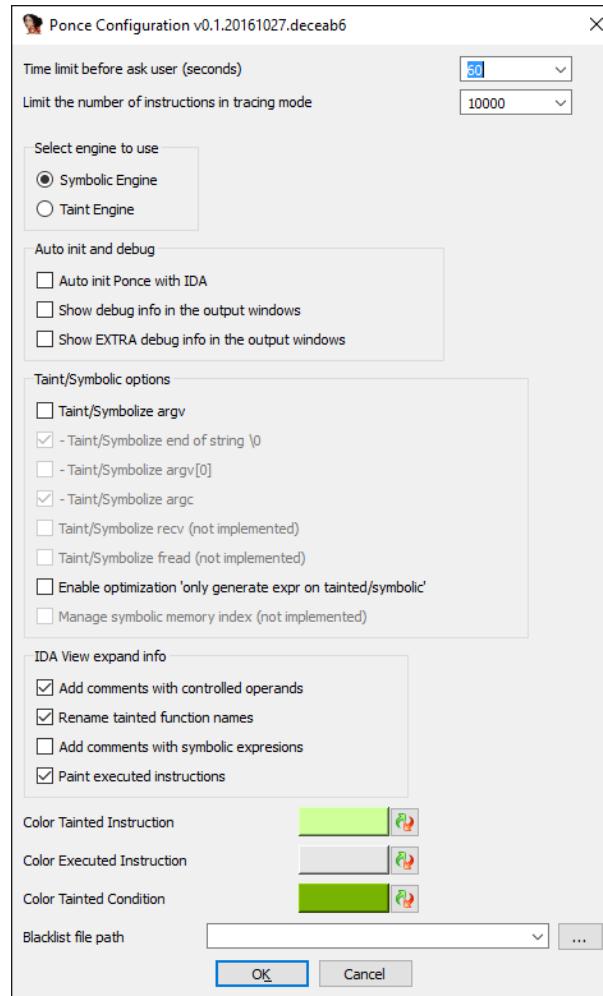
Set a breakpoint at program entry  
Want to symbolize argv[1]



# Ponce Example

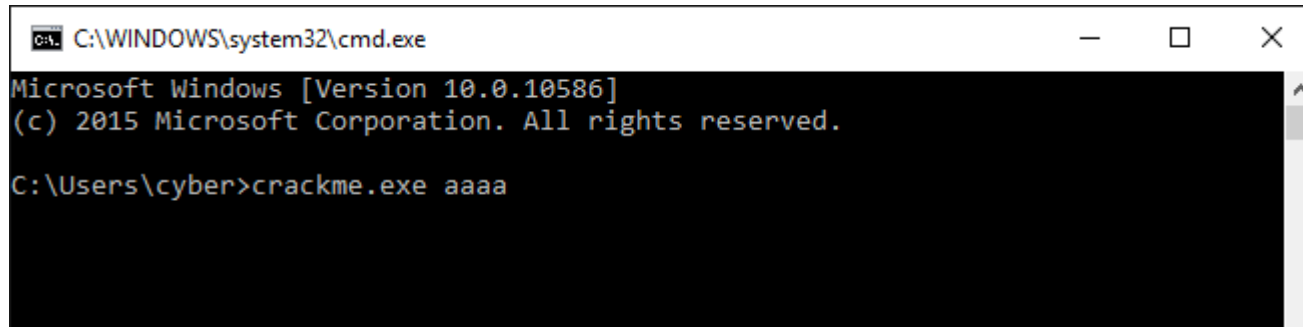
Start ponce plugin

Set to symbolic engine



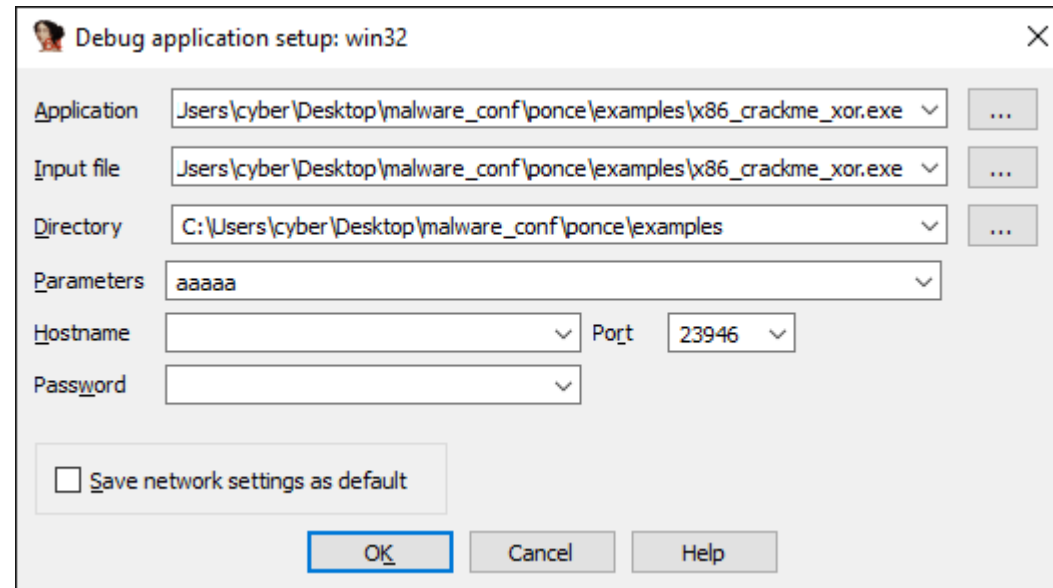
# Ponce Example

Set an input argument



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\cyber>crackme.exe aaaa
```



Debug application setup: win32

Application: J:\Users\cyber\Desktop\malware\_conf\ponce\examples\x86\_crackme\_xor.exe ...

Input file: J:\Users\cyber\Desktop\malware\_conf\ponce\examples\x86\_crackme\_xor.exe ...

Directory: C:\Users\cyber\Desktop\malware\_conf\ponce\examples ...

Parameters: aaaaa

Hostname: Port: 23946

Password:

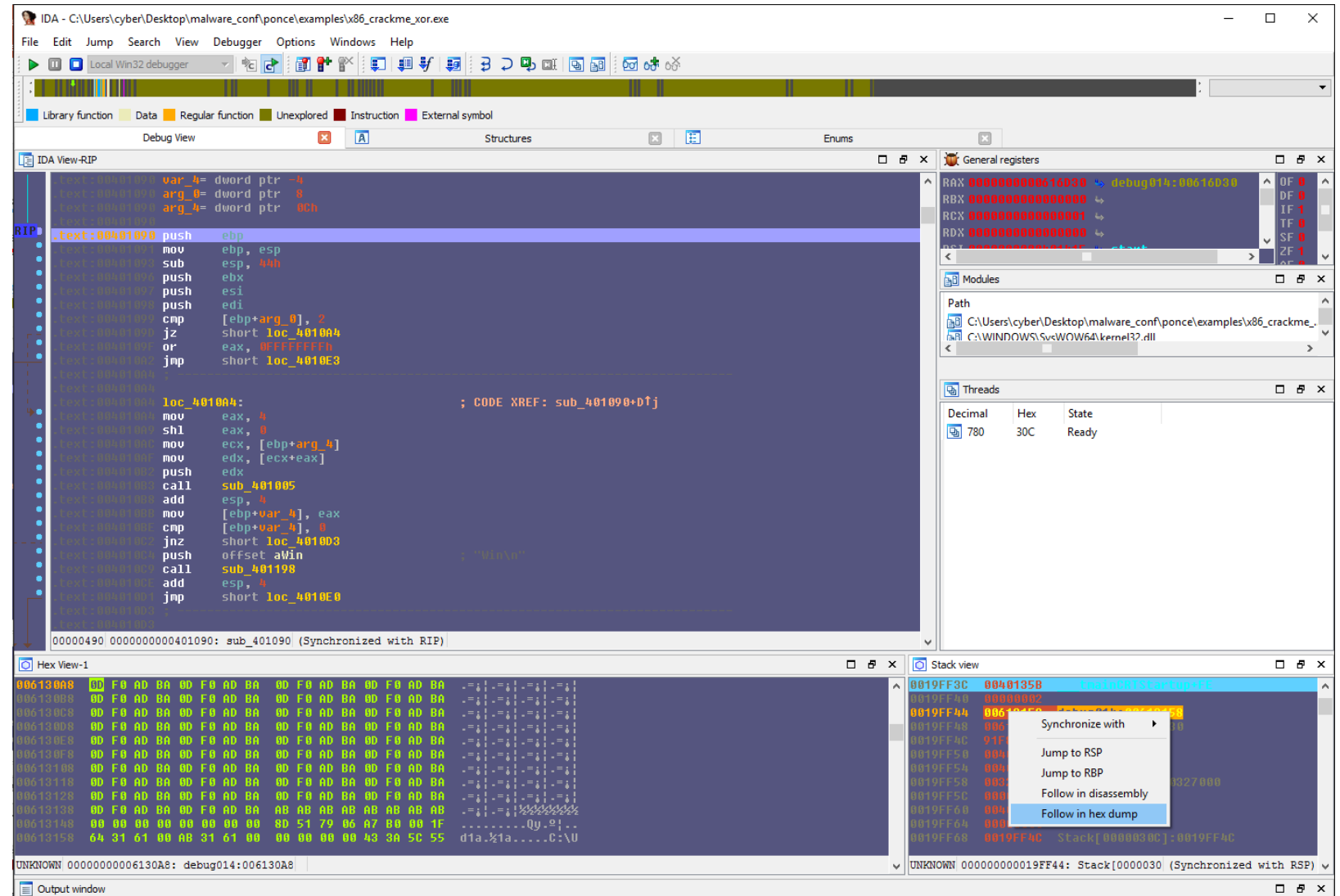
Save network settings as default

OK Cancel Help



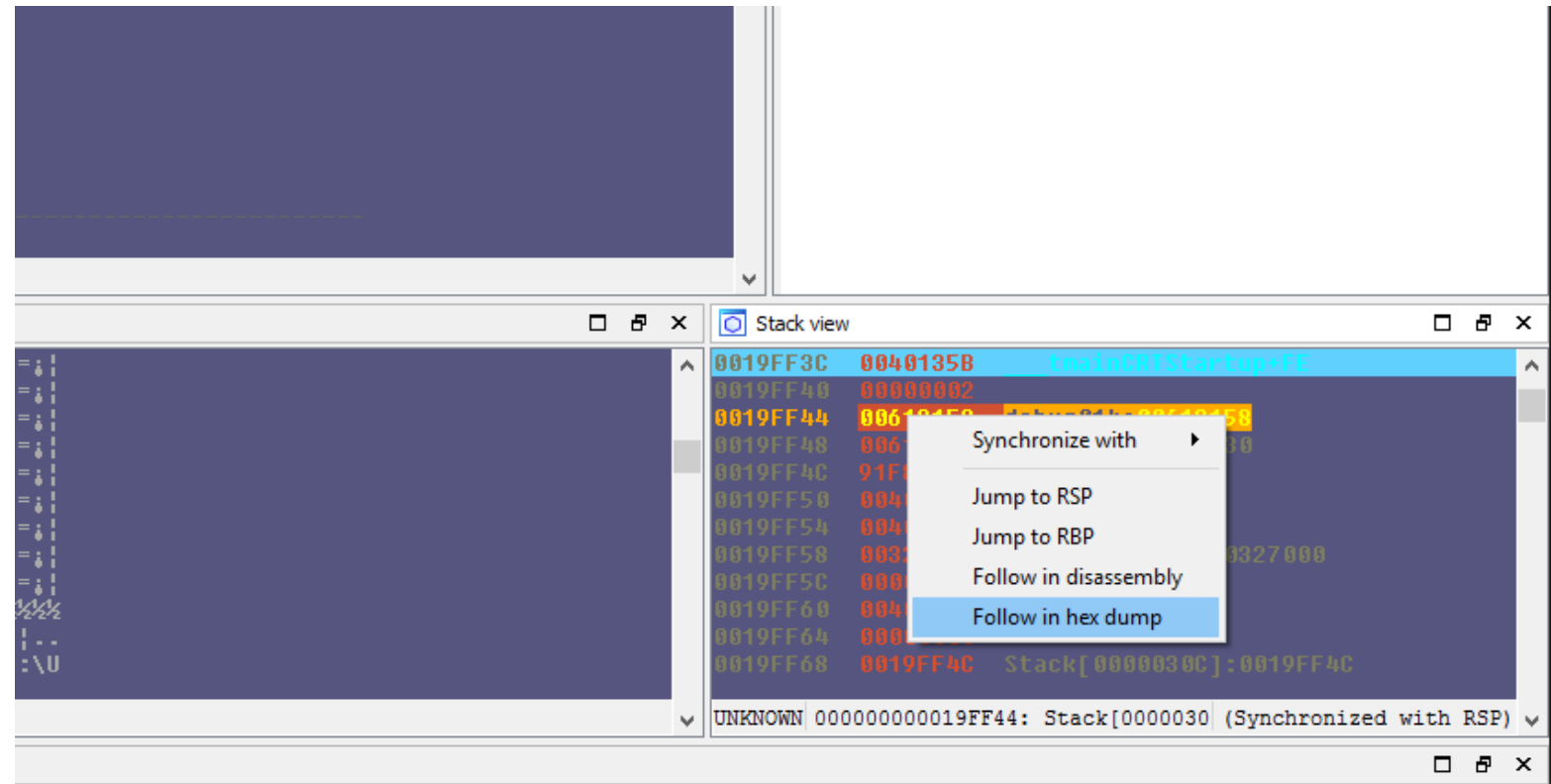
# Ponce Example

Debugger started



# Ponce Example

Jump in hex to the where  
argv[1] is passed in



# Ponce Example

Symbolize the data that bytes  
that we care about

AAAAA

The screenshot shows a debugger window with the following assembly code:

```
text:004010C4 push offset aWin ; "Winrar"  
text:004010C9 call sub_401198  
text:004010CE add esp, 4  
text:004010D1 jmp short loc_4010E0  
text:004010D3 ;  
text:004010D9 ;
```

Below the assembly, the address bar shows: 00000490 0000000000401090: sub\_401090 (Synchronized with RIP)

The Hex View-1 window displays the following hex data:

00613128	0D F0 AD BA 0D F0 AD BA	0D F0 AD BA 0D F0 AD BA	..= !..= !..= !..= !
00613138	0D F0 AD BA 0D F0 AD BA	AB AB AB AB AB AB AB AB	..= !..= ! ????????
00613148	00 00 00 00 00 00 00 00	8D 51 79 06 A7 B0 00 1F	.....Qy.º ..
00613158	64 31 61 00 AB 31 61 00	00 00 00 00 43 3A 5C 55	d1a.¼1a.....C:\U
00613168	73 65 72 73 5C 63 79 62	65 72 5C 44 65 73 6B 74	sers\cyber\Deskt
00613178	6F 70 5C 6D 61 6C 77 61	72 65 5F 63 6F 6E 66 5C	op\malware_conf\
00613188	70 6F 6E 63 65 5C 65 78	61 6D 70 6C 65 73 5C 78	ponce\examples\x
00613198	38 36 5F 63 72 61 63 6B	6D 65 5F 78 6F 72 2E 65	86_crackme_xor.e
006131A8	78 65 00 61 61 61 61	00 00 00 00 00 00 AB AB	xe.aaaaa.????????
006131B8	AB FE EE FE EE FE EE	00 00	¼ e e e .....
006131C8	81 51 7A 09 EF B0 00	61 00	0z.pl ia Pla
006131D8	EE FE EE FE EE FE EE		

A context menu is open over the hex view, with the following options:

- Synchronize with
- Symbolic (selected) - Symbolize Memory (Ctrl+Shift+M)
- Data format
- Columns
- Text
- Edit... (F2)
- Save to file...

The bottom status bar shows: AU: idle Down Disk: 8GB



# Ponce Example

Implement SMT solver to deduce what bytes we needed to obtain the endpoint

The screenshot displays the Immunity Debugger interface. The assembly window shows the following code:

```
00401080 mov     [ebp+var_4], eax ; Symbolized regs: eax
00401089 cmp     [ebp+var_4], 0AD6Dh ; Symbolized memory: 0x10FF30
00401090 jnz     short loc_4010A2 ; Symbolized regs: zf
```

The symbol table shows:

```
004010A2
004010A2
004010A7
004010AD
```

The disassembly window highlights the instruction at address 00401080:

```
FFh 00401083
00401080 loc_401080: Symbol
00401080 mov     eax, [ebp+var_4]
```

The context menu is open, showing the following options:

- Group nodes
- List cross references to... Ctrl+X
- List cross references from... Ctrl+J
- Enter comment... :
- Enter repeatable comment... ;
- Edit function... Alt+P
- Hide Ctrl+Numpad+-
- Text view
- Proximity browser Numpad+-
- Undefine U
- Synchronize with ▶
- Run to cursor F4
- Add write trace
- Add read/write trace
- Add execution trace
- Add breakpoint F2
- Xrefs graph to...
- Xrefs graph from...
- Enable ponce tracing Ctrl+Shift+E
- Symbolic ▶
- SMT ▶**
- Snapshot ▶
- Execute native Ctrl+Shift+F9

The SMT submenu is open, showing the following options:

- Negate & Inject Ctrl+Shift+N
- Negate, Inject & Restore snapshot Ctrl+Shift+I
- Solve formula ▶**

The Solve formula option is selected, and the formula `5. 0x401090 -> 0x4010a2` is displayed in the bottom right corner.

# Ponce Example

We have the key!

SX7@Y

```
PDB: DIA interface version 9.0
Debugger: thread 9652 has exited (code 44399)
Debugger: thread 4736 has exited (code 44399)
Debugger: process has exited (exit code 44399)
[+] Solution found! Values:
- SymVar_0 (argv[1][0]):0x53 (S)
- SymVar_1 (argv[1][1]):0x58 (X)
- SymVar_2 (argv[1][2]):0x37 (7)
- SymVar_3 (argv[1][3]):0x40 (@)
- SymVar_4 (argv[1][4]):0x59 (Y)
```

Python

AU: idle | Down | Disk: 3GB

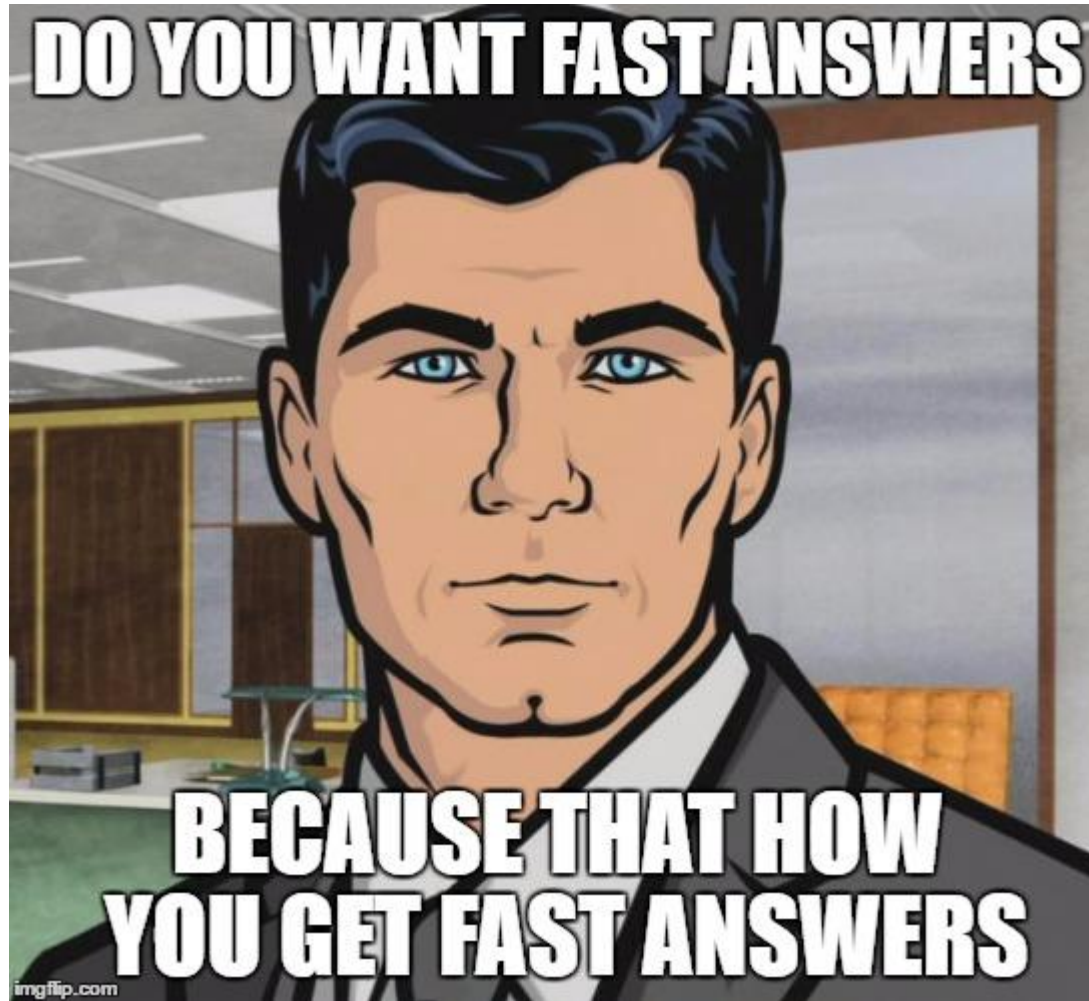
# Ponce Example

Success!  
We have the key

SX7@Y

```
E ~  
cyber@DESKTOP-CS7U7T8 ~  
$ ./x86_crackme_xor.exe "'python -c 'print 'SX7@Y'''"  
win!  
cyber@DESKTOP-CS7U7T8 ~  
$
```

# Ponce





#5



Manticore is your friend...

# Manticore



A Plugin to IDA is great, but what if I wanted to do something more automated?



# Manticore



A Plugin to IDA is great, but what if I wanted to do something more automated?

Maybe a code oriented approach?



# Manticore



A Plugin to IDA is great, but what if I wanted to do something more automated?

Maybe a code oriented approach?

Not an auto-solve but a much better approach.



# Manticore

Open source tool

Command line interface

Quickly generates use cases



```
ubuntu@ubuntu: ~  
ubuntu@ubuntu:~$ manticore  
usage: manticore [-h] [--workspace WORKSPACE] [-v] [--profile]  
                [--buffer BUFFER] [--size SIZE] [--offset OFFSET]  
                [--maxsyb MAXSYMB] [--data DATA] [--env ENV]  
                [--policy POLICY] [--dumpafter DUMPAFTER]  
                [--maxstorage MAXSTORAGE] [--maxstates MAXSTATES]  
                [--procs PROCS] [--timeout TIMEOUT] [--replay REPLAY]  
                [--coverage COVERAGE] [--errorfile ERRORFILE]  
                [--context CONTEXT] [--assertions ASSERTIONS] [--names NAMES]  
                PROGRAM [PROGRAM ...]  
manticore: error: too few arguments
```

# Manticore

Open source tool

Command line interface

Quickly generates use cases

Python API

To answer more in depth questions

- How many times does a program execute this function?



```
ubuntu@ubuntu: ~  
ubuntu@ubuntu:~$ manticore  
usage: manticore [-h] [--workspace WORKSPACE] [-v] [--profile]  
                [--buffer BUFFER] [--size SIZE] [--offset OFFSET]  
                [--maxsyb MAXSYMB] [--data DATA] [--env ENV]  
                [--policy POLICY] [--dumpafter DUMPAFTER]  
                [--maxstorage MAXSTORAGE] [--maxstates MAXSTATES]  
                [--procs PROCS] [--timeout TIMEOUT] [--replay REPLAY]  
                [--coverage COVERAGE] [--errorfile ERRORFILE]  
                [--context CONTEXT] [--assertions ASSERTIONS] [--names NAMES]  
                PROGRAM [PROGRAM ...]  
manticore: error: too few arguments
```



# Manticore

Open source tool

Command line interface

Quickly generates use cases

Python API

To answer more in depth questions

- How many times does a program execute this function?
- What input causes execution of this block of code?

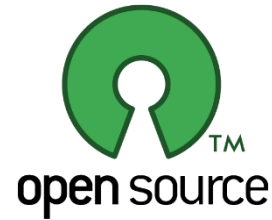


```
ubuntu@ubuntu: ~  
ubuntu@ubuntu:~$ manticore  
usage: manticore [-h] [--workspace WORKSPACE] [-v] [--profile]  
                [--buffer BUFFER] [--size SIZE] [--offset OFFSET]  
                [--maxsyb MAXSYMB] [--data DATA] [--env ENV]  
                [--policy POLICY] [--dumpafter DUMPAFTER]  
                [--maxstorage MAXSTORAGE] [--maxstates MAXSTATES]  
                [--procs PROCS] [--timeout TIMEOUT] [--replay REPLAY]  
                [--coverage COVERAGE] [--errorfile ERRORFILE]  
                [--context CONTEXT] [--assertions ASSERTIONS] [--names NAMES]  
                PROGRAM [PROGRAM ...]  
manticore: error: too few arguments
```



# Manticore

Open source tool



Command line interface

Quickly generates use cases

```
ubuntu@ubuntu:~$ manticore
usage: manticore [-h] [--workspace WORKSPACE] [-v] [--profile]
                [--buffer BUFFER] [--size SIZE] [--offset OFFSET]
                [--maxsyb MAXSYMB] [--data DATA] [--env ENV]
                [--policy POLICY] [--dumpafter DUMPAFTER]
                [--maxstorage MAXSTORAGE] [--maxstates MAXSTATES]
                [--procs PROCS] [--timeout TIMEOUT] [--replay REPLAY]
                [--coverage COVERAGE] [--errorfile ERRORFILE]
                [--context CONTEXT] [--assertions ASSERTIONS] [--names NAMES]
                PROGRAM [PROGRAM ...]
manticore: error: too few arguments
```

Python API

To answer more in depth questions

- How many times does a program execute this function?
- What input causes execution of this block of code?
- At point X in execution, is it possible for variable Y to be a specified value?





# Manticore



## Pythonic symbolic programming

Hook addresses and add constraints

```
from manticore import Manticore

hook_pc = 0x400ca0

m = Manticore('./path/to/binary')

@m.hook(hook_pc)
def hook(state):
    cpu = state.cpu
    print 'eax', cpu.EAX
    print cpu.read_int(cpu.SP)

    m.terminate() # tell Manticore to stop

m.run()
```

# Manticore



## Pythonic symbolic programming

Hook addresses and add constraints

Read/write into registers/memory at point of execution

```
from manticore import Manticore

hook_pc = 0x400ca0

m = Manticore('./path/to/binary')

@m.hook(hook_pc)
def hook(state):
    cpu = state.cpu
    print 'eax', cpu.EAX
    print cpu.read_int(cpu.SP)

    m.terminate() # tell Manticore to stop

m.run()
```

# Manticore



## Pythonic symbolic programming

Hook addresses and add constraints

Read/write into registers/memory at point of execution

Use SMT solver in Z3 to do symbolically solve deep assembly structures.

```
from manticore import Manticore

hook_pc = 0x400ca0

m = Manticore('./path/to/binary')

@m.hook(hook_pc)
def hook(state):
    cpu = state.cpu
    print 'eax', cpu.EAX
    print cpu.read_int(cpu.SP)

    m.terminate() # tell Manticore to stop

m.run()
```

# Manticore



Simple to install (Ubuntu 16.04):

Of course working in a virtual environment is recommended

```
$ pip install manticore
```

```
$ apt install z3
```



# Manticore

Google CTF 2016

- Cory Duplantis

Reverse engineering challenge

Bust out the rush tape!



# Manticore – Dive in

```
0000000000400590
0000000000400590 ; Segment type: Pure code
0000000000400590 ; Segment permissions: Read/Execute
0000000000400590 _text segment para public 'CODE' use64
0000000000400590 assume cs:_text
0000000000400590 ;org 400590h
0000000000400590 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
0000000000400590
0000000000400590
0000000000400590 ; Attributes: noreturn
0000000000400590
0000000000400590 ; int __cdecl main(int, char **, char **)
0000000000400590 main proc near
0000000000400590 sub     rsp, 8
0000000000400594 cmp     edi, 2
0000000000400597 jz      short loc_4005AA
```

```
0000000000400599 mov     esi, offset format ; "./unbreakable_enterprise_product_activa"...
000000000040059E mov     edi, 1             ; status
00000000004005A3 xor     eax, eax
00000000004005A5 call    _errx
```

```
00000000004005AA
00000000004005AA loc_4005AA:                ; src
00000000004005AA mov     rsi, [rsi+8]
00000000004005AE mov     edx, 43h          ; n
00000000004005B2 mov     edi, offset dest ; dest
00000000004005B8 call    _strncpy
00000000004005BB mov     esi, esi
00000000004005BF call    sub_4027F0
00000000004005C4 xor     eax, eax
00000000004005C6 call    sub_402830
```

# Manticore – Dive in

```
iva''...  
000000000004005AA  
000000000004005AA loc_4005AA: ; src  
000000000004005AA mov rsi, [rsi+8]  
000000000004005AE mov edx, 43h ; n  
000000000004005B3 mov edi, offset dest ; dest  
000000000004005B8 call _strncpy  
000000000004005BD xor eax, eax  
000000000004005BF call sub_4027F0  
000000000004005C4 xor eax, eax  
000000000004005C6 call sub_402830
```

# Manticore – Dive in

```
input_addr = 0x6042c0
num_bytes = 0x43

# Entry point
@m.hook(0x400729)
def hook(state):
    """ CAUTION: HACK """
    """ From entry point, jump directly to code performing check """

    # Create a symbolic buffer for our input
    buffer = state.new_symbolic_buffer(0x43)

    # We are given in the challenge that the flag begins with CTF{
    # So we can apply constraints to ensure that is true
    state.constrain(buffer[0] == ord('C'))
    state.constrain(buffer[1] == ord('T'))
    state.constrain(buffer[2] == ord('F'))
    state.constrain(buffer[3] == ord('{'))

    # Store our symbolic input in the global buffer read by the check
    state.cpu.write_bytes(input_addr, buffer)

    # By default, `hook` doesn't patch the instruction, so we hop over
    # the hooked strncpy and execute the next instruction
    state.cpu.EIP = 0x4005bd
```



# Manticore – Dive in

Set up a symbolic buffer 0x43 in size

```
input_addr = 0x6042c0
num_bytes = 0x43

# Entry point
@m.hook(0x400729)
def hook(state):
    """ CAUTION: HACK """
    """ From entry point, jump directly to code performing check """

    # Create a symbolic buffer for our input
    buffer = state.new_symbolic_buffer(0x43)

    # We are given in the challenge that the flag begins with CTF{
    # So we can apply constraints to ensure that is true
    state.constrain(buffer[0] == ord('C'))
    state.constrain(buffer[1] == ord('T'))
    state.constrain(buffer[2] == ord('F'))
    state.constrain(buffer[3] == ord('{'))

    # Store our symbolic input in the global buffer read by the check
    state.cpu.write_bytes(input_addr, buffer)

    # By default, `hook` doesn't patch the instruction, so we hop over
    # the hooked strncpy and execute the next instruction
    state.cpu.EIP = 0x4005bd
```

# Manticore – Dive in

Set up a symbolic buffer 0x43 in size

In this challenge the string starts with: **CTF{**

```
input_addr = 0x6042c0
num_bytes = 0x43

# Entry point
@m.hook(0x400729)
def hook(state):
    """ CAUTION: HACK """
    """ From entry point, jump directly to code performing check """

    # Create a symbolic buffer for our input
    buffer = state.new_symbolic_buffer(0x43)

    # We are given in the challenge that the flag begins with CTF{
    # So we can apply constraints to ensure that is true
    state.constrain(buffer[0] == ord('C'))
    state.constrain(buffer[1] == ord('T'))
    state.constrain(buffer[2] == ord('F'))
    state.constrain(buffer[3] == ord('{'))

    # Store our symbolic input in the global buffer read by the check
    state.cpu.write_bytes(input_addr, buffer)

    # By default, `hook` doesn't patch the instruction, so we hop over
    # the hooked strncpy and execute the next instruction
    state.cpu.EIP = 0x4005bd
```

# Manticore – Dive in

Set up a symbolic buffer 0x43 in size

In this challenge the string starts with: **CTF{**

Write data to the buffer

```
input_addr = 0x6042c0
num_bytes = 0x43

# Entry point
@m.hook(0x400729)
def hook(state):
    """ CAUTION: HACK """
    """ From entry point, jump directly to code performing check """

    # Create a symbolic buffer for our input
    buffer = state.new_symbolic_buffer(0x43)

    # We are given in the challenge that the flag begins with CTF{
    # So we can apply constraints to ensure that is true
    state.constrain(buffer[0] == ord('C'))
    state.constrain(buffer[1] == ord('T'))
    state.constrain(buffer[2] == ord('F'))
    state.constrain(buffer[3] == ord('{'))

    # Store our symbolic input in the global buffer read by the check
    state.cpu.write_bytes(input_addr, buffer)

    # By default, `hook` doesn't patch the instruction, so we hop over
    # the hooked strncpy and execute the next instruction
    state.cpu.EIP = 0x4005bd
```

# Manticore – Dive in

Set up a symbolic buffer 0x43 in size

In this challenge the string starts with: **CTF{**

Write data to the buffer

Hook does not patch instruction

Set EIP to after the call

```
input_addr = 0x6042c0
num_bytes = 0x43

# Entry point
@m.hook(0x400729)
def hook(state):
    """ CAUTION: HACK """
    """ From entry point, jump directly to code performing check """

    # Create a symbolic buffer for our input
    buffer = state.new_symbolic_buffer(0x43)

    # We are given in the challenge that the flag begins with CTF{
    # So we can apply constraints to ensure that is true
    state.constrain(buffer[0] == ord('C'))
    state.constrain(buffer[1] == ord('T'))
    state.constrain(buffer[2] == ord('F'))
    state.constrain(buffer[3] == ord('{'))

    # Store our symbolic input in the global buffer read by the check
    state.cpu.write_bytes(input_addr, buffer)

    # By default, `hook` doesn't patch the instruction, so we hop over
    # the hooked strncpy and execute the next instruction
    state.cpu.EIP = 0x4005bd
```

# Manticore – Dive in

Function to remove failed paths –  
auto negate and inject

---

```
# Failure case
@m.hook(0x400850)
def hook(state):
    """ Stop executing paths that reach the `failure` function"""
    print("Invalid path.. abandoning")
    state.abandon()

# Success case
@m.hook(0x400724)
def hook(state):
    print("Hit the final state.. solving")

    state._solver._command = 'z3 -t:240000 -smt2 -in' # Hack around to
    buffer = state.cpu.read_bytes(input_addr, num_bytes)
    res = ''.join(chr(state.solve_one(x)) for x in buffer)
    print(res) # CTF{0The1Quick2Brown3Fox4Jumped5Over6The7Lazy8Fox9}

    # We found the flag, no need to continue execution
    m.terminate()

m.should_profile = True
m.run(procs=10)
```

# Manticore – Dive in

Function to remove failed paths –  
auto negate and inject

---

Define end point in execution

- Use z3 to solve against the input address we defined
- 

```
# Failure case
@m.hook(0x400850)
def hook(state):
    """ Stop executing paths that reach the `failure` function"""
    print("Invalid path.. abandoning")
    state.abandon()

# Success case
@m.hook(0x400724)
def hook(state):
    print("Hit the final state.. solving")

    state._solver._command = 'z3 -t:240000 -smt2 -in' # Hack around tr
    buffer = state.cpu.read_bytes(input_addr, num_bytes)
    res = ''.join(chr(state.solve_one(x)) for x in buffer)
    print(res) # CTF{0The1Quick2Brown3Fox4Jumped5Over6The7Lazy8Fox9

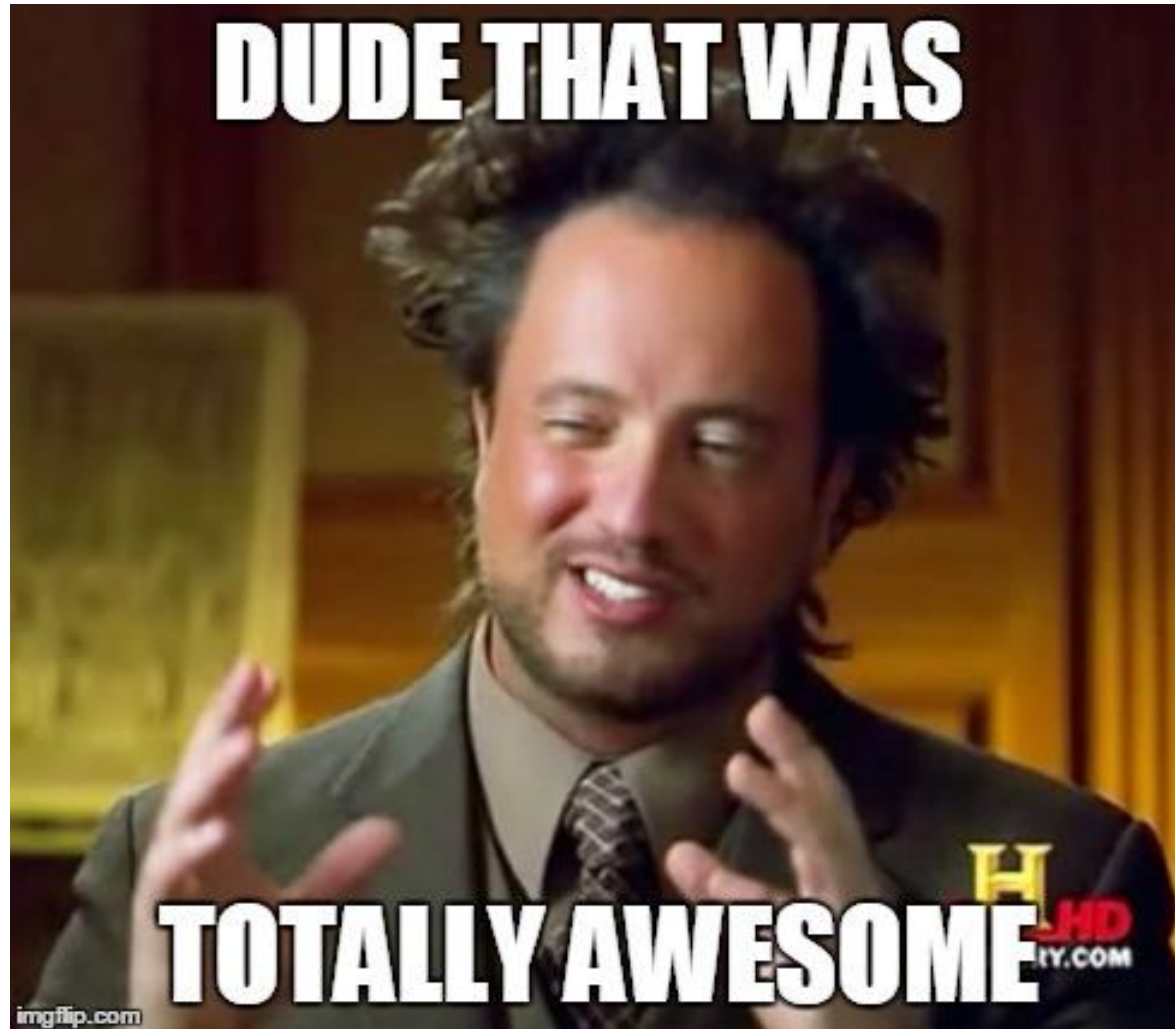
    # We found the flag, no need to continue execution
    m.terminate()

m.should_profile = True
m.run(procs=10)
```

# Manticore – Dive in

```
Invalid path.. abandoning  
Invalid path.. abandoning  
Invalid path.. abandoning  
Invalid path.. abandoning  
Invalid path.. abandoning  
Invalid path.. abandoning  
Hit the final state.. solving  
CTF{0The1Quick2Brown3Fox4Jumped5Over6The7Lazy8Fox9}eeeeeeeeeeeeeeeeee  
ubuntu@ubuntu:~/Downloads$
```

# Manticore – Dive in





# #6

Okay, so maybe it won't solve everything but it will help.

# Where are we going?



## **Symbolic of the future?**

Traditional tools have served us really well and will still be the go to solve all of the in depth questions we have



## **The future of reverse engineering?**

Increasing pace of malware, new outbreaks and the need to get answers fast. We can't afford the time that attackers provide.

# SYMBOLIC OF THE FUTURE

Thanks for your time!

Questions?

(Also it's lunch time and I'm hungry)