Host Vulnerabilities

UT LAW379M

FALL 2022

LECTURE NOTES

Two Broad Subtopics

1. Software Vulnerabilities
2. Malware

Brief Overview to Execution

Today: ***very brief*** overview of ***<u>Control Flow Hijacking</u>***

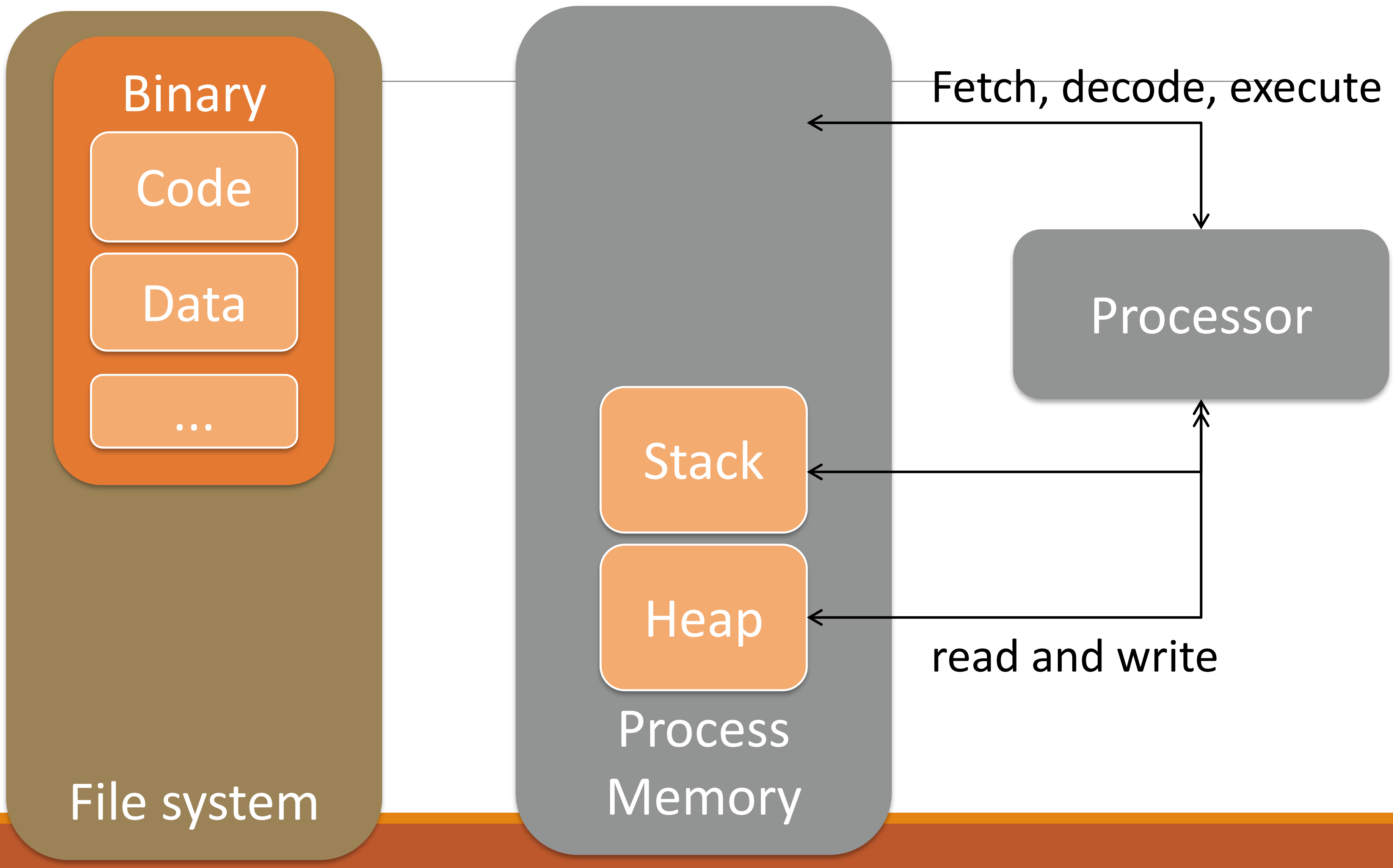
- There are other types of vulnerabilities (e.g misconfigured)
- Control Flow Hijacking is probably the hardest to grasp

Critical Concepts:

- The “normal” flow of control for authorized instructions
- Inputs that change the flow to unauthorized instructions

ATTRIBUTION: Derived from slides by Dave Brumley, CMU

Basic Execution

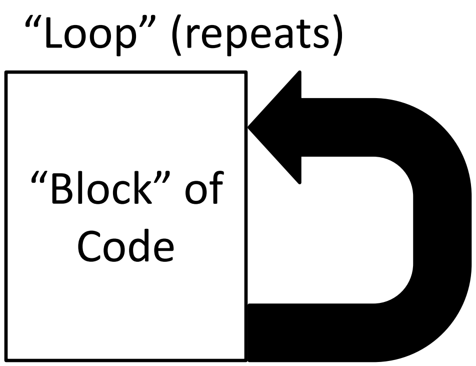


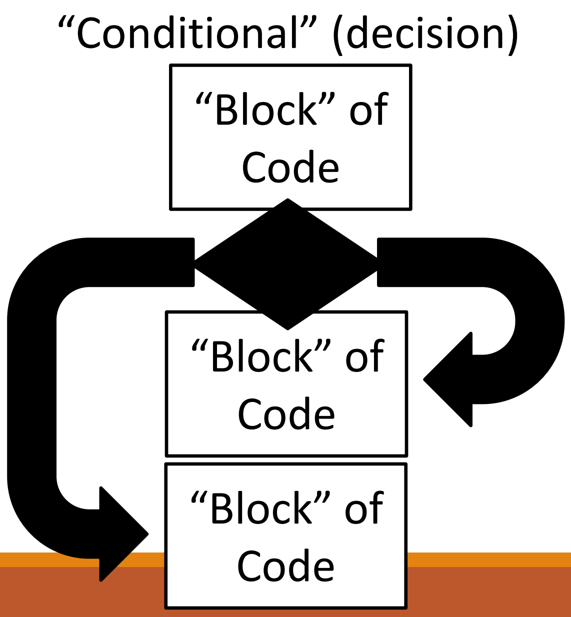
What is “Control Flow”

Computer instructions are generally ***sequential***

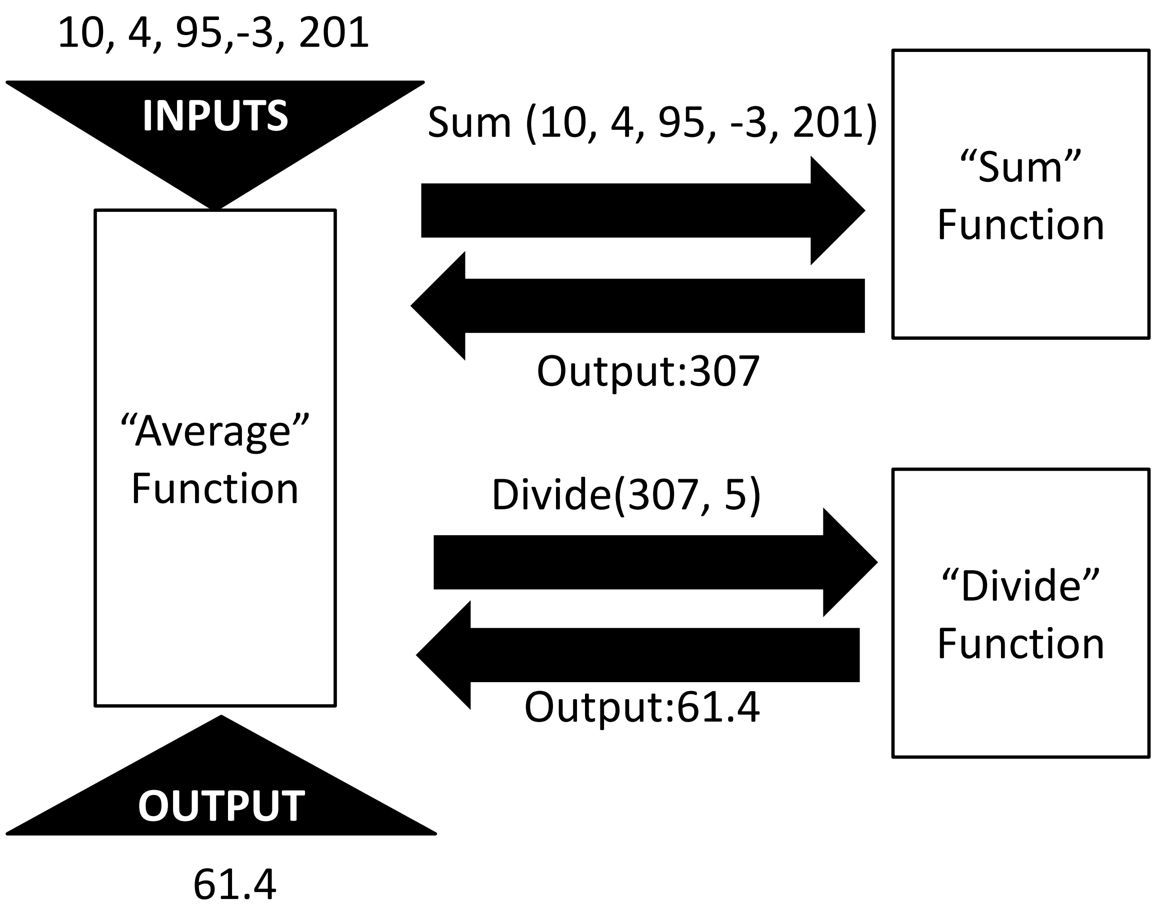
Execute instruction 1, then 2, then 3…

But there are instructions that ***jump around***





Function Call



Function Calls?

Assembly “function calls” don’t really exist

- Rather, jump to new location of instructions in the big long list!!!
- Save context of old location
- Load context for new location
- Include information for “returning” (output)

The “Stack” (Scratch Pad)

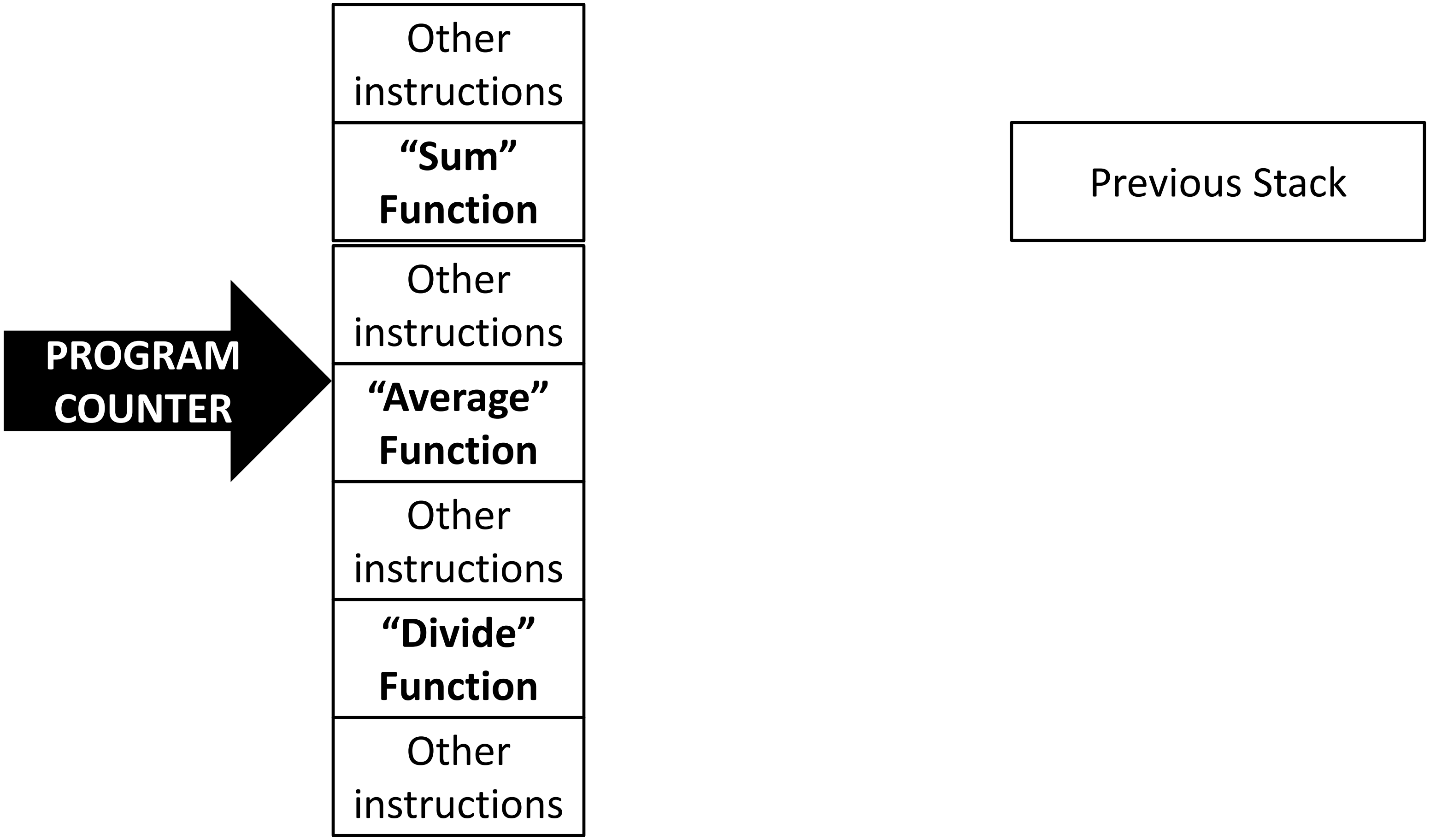
Computers need temporary memory for temporary calculations

In the “Average” example, it temporarily needed the number count

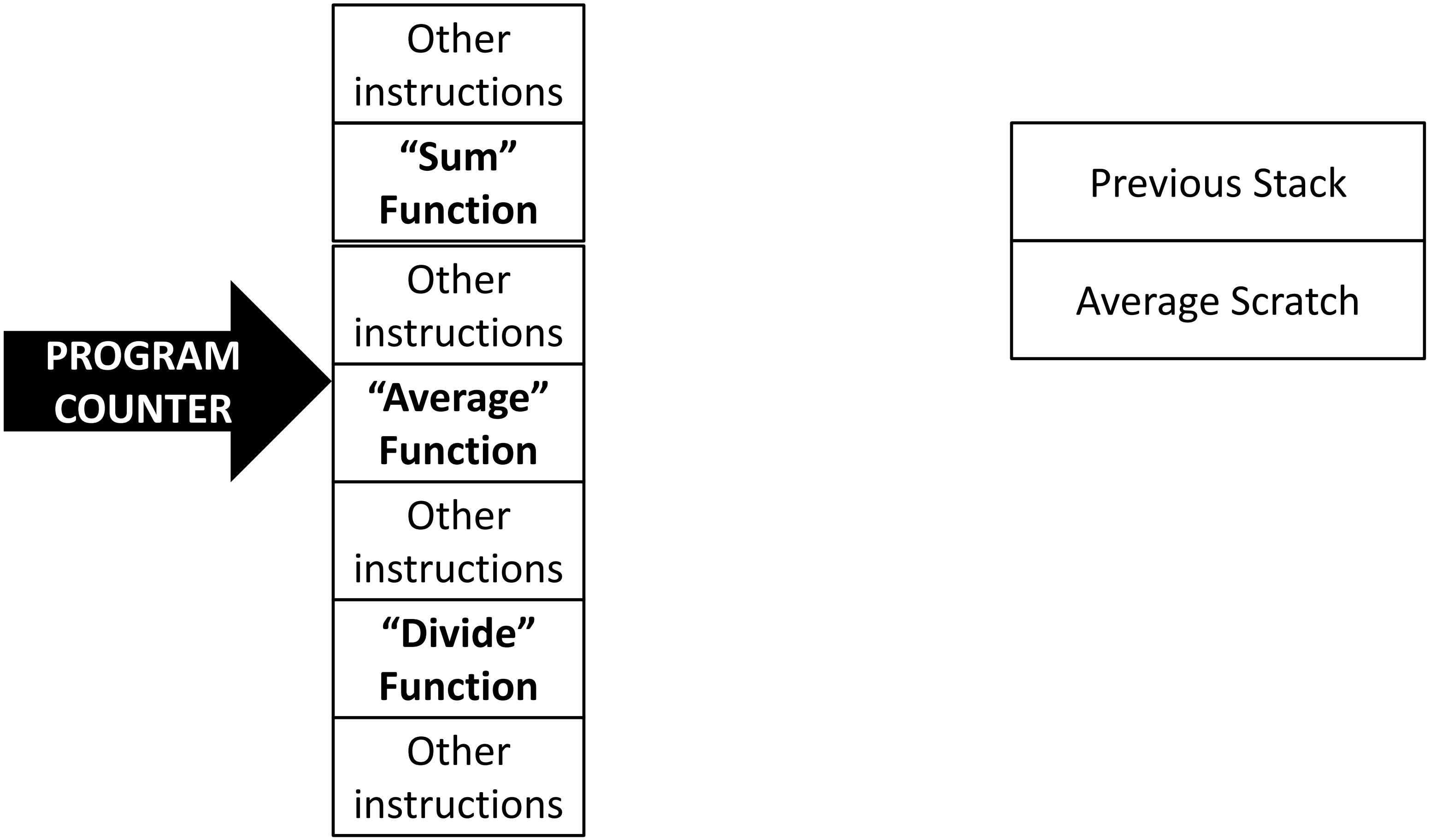
Data like this is stored on “The Stack”

It’s a space in memory that “grows”

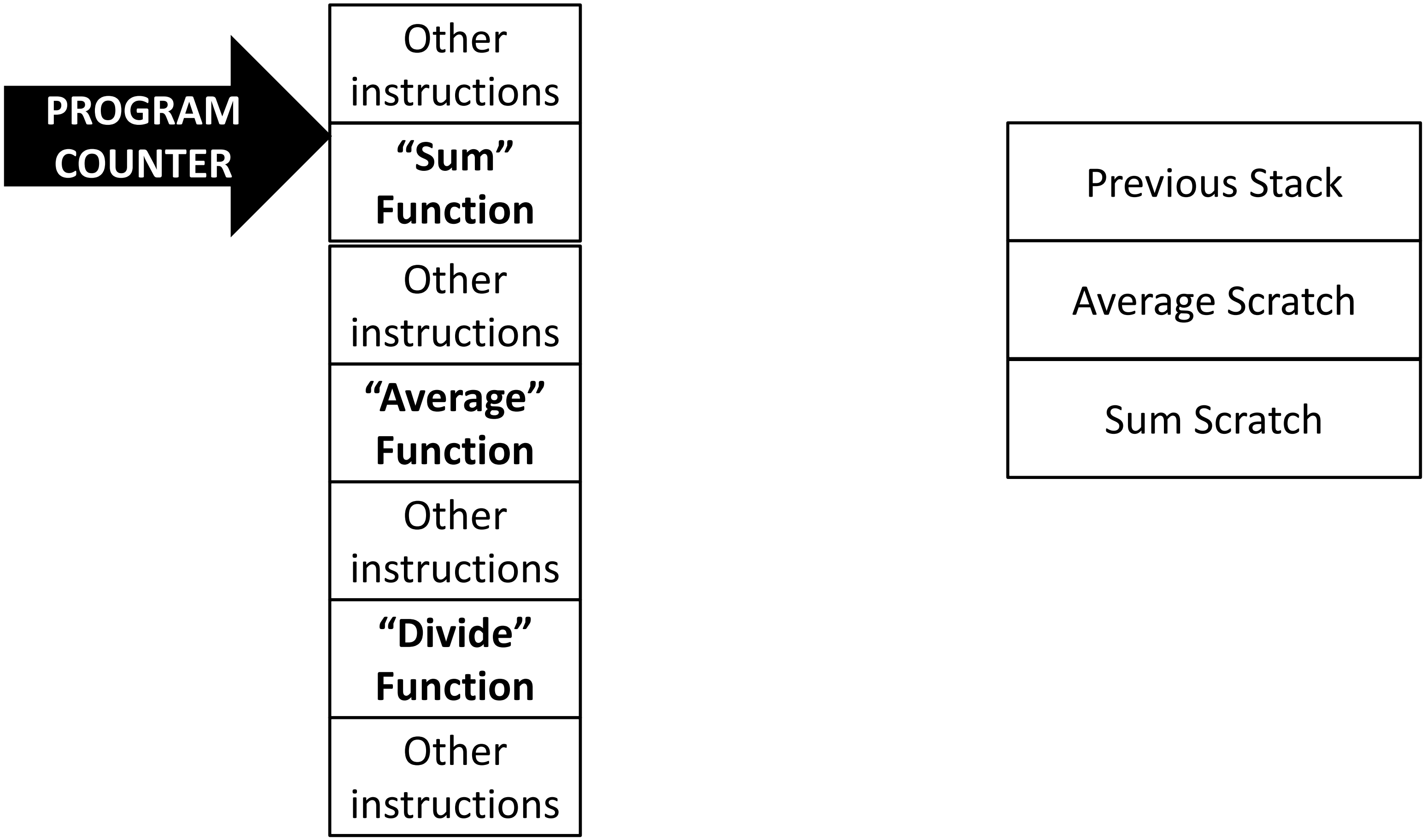
Average Function Start



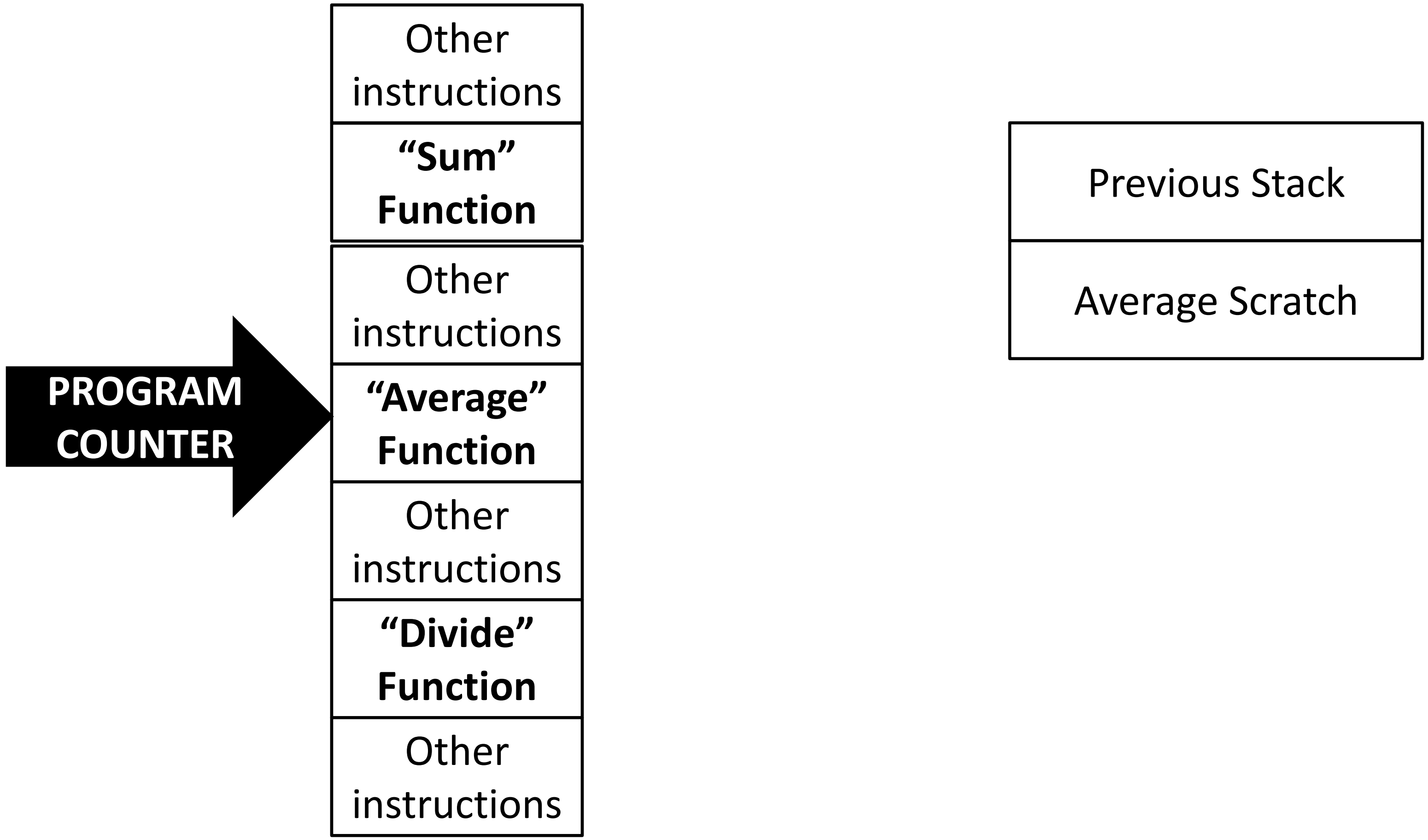
Average Function Create Stack



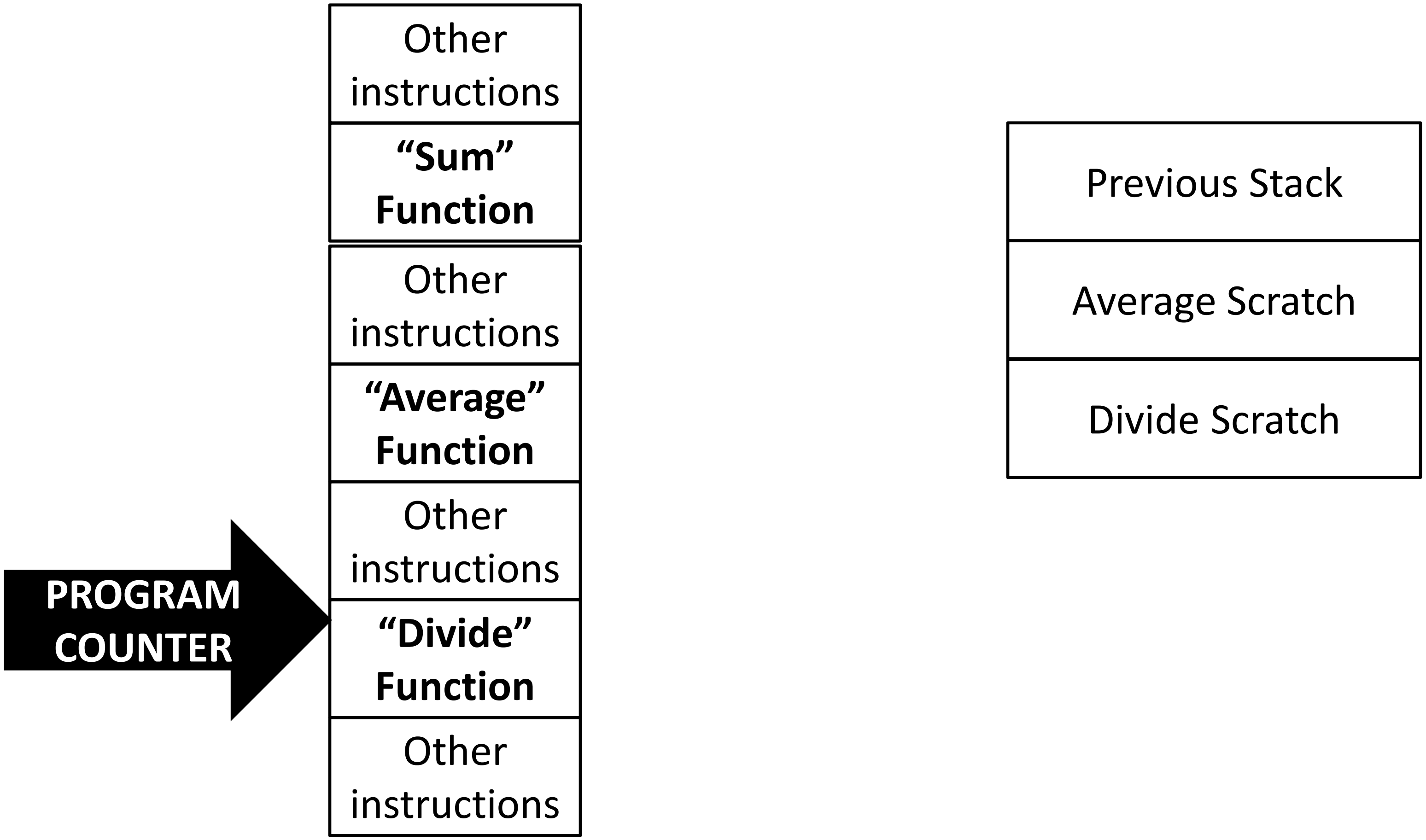
Call Sum from Average



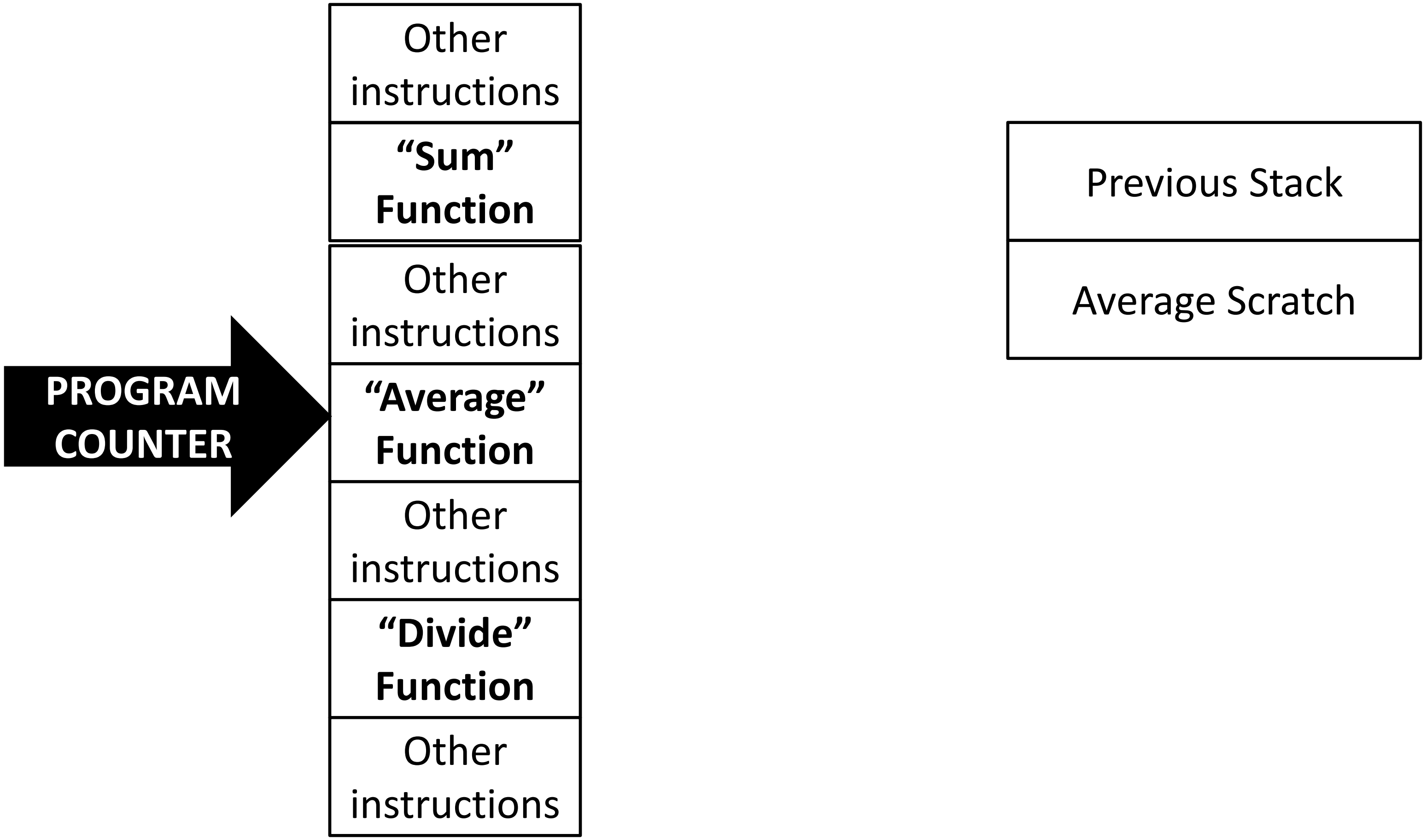
Return from Sum to Average



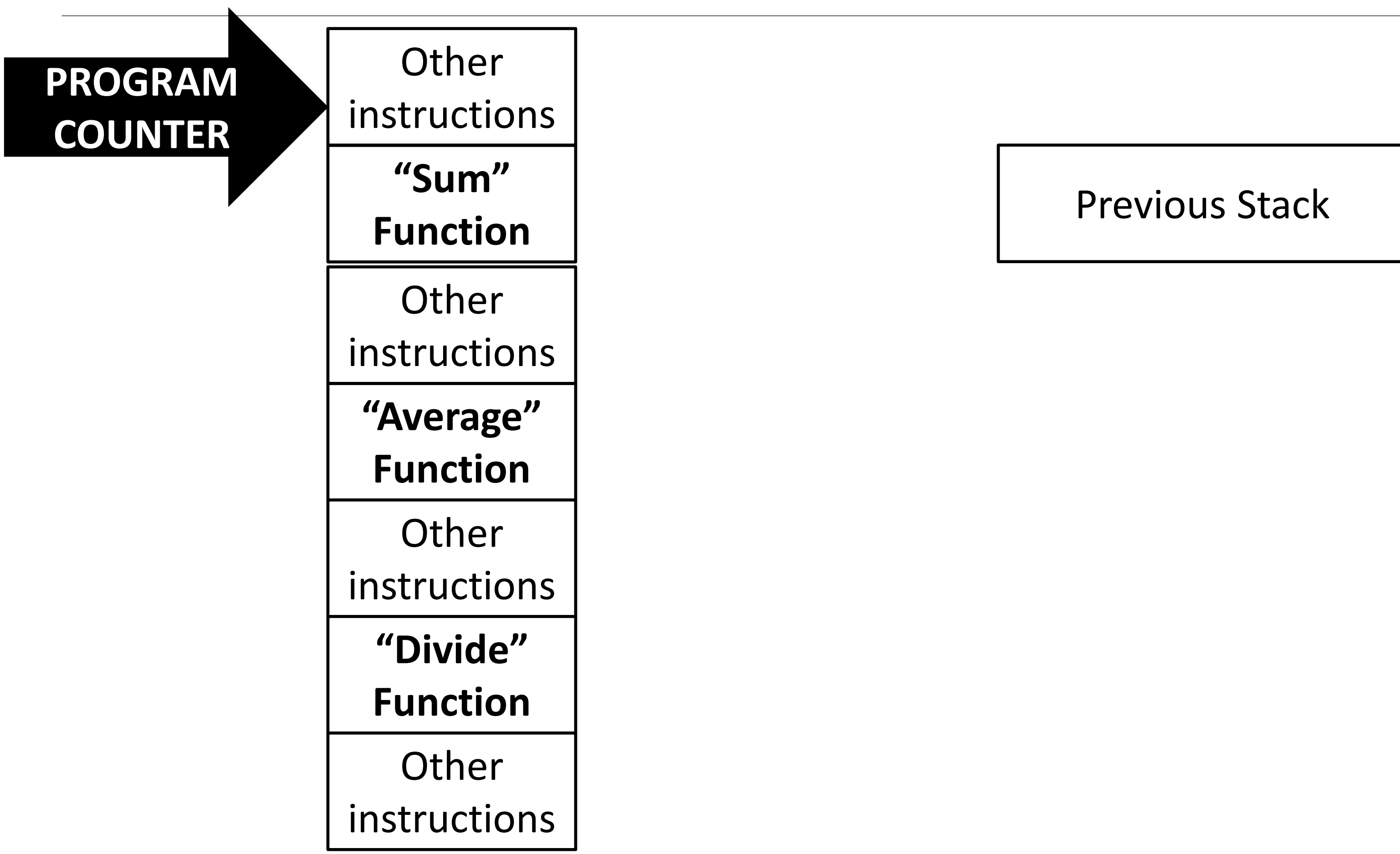
Call Divide from Average



Return from Divide to Average



Return from Average to ???



Control Flow Again

In this context, controlling the movement between functions.

The system has to keep track of this flow

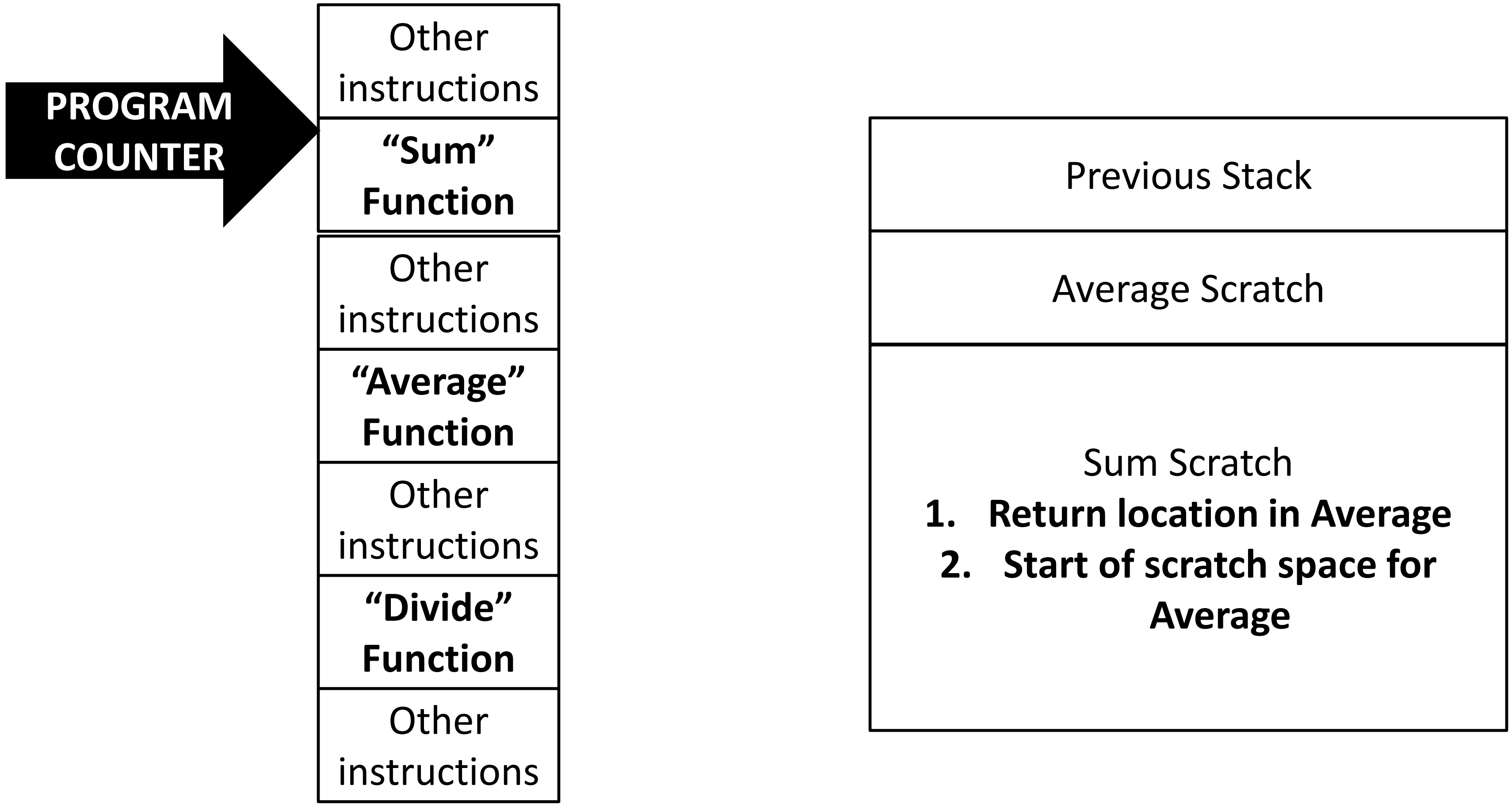
It has to know which part of the stack to erase

It has to know which instruction to jump to

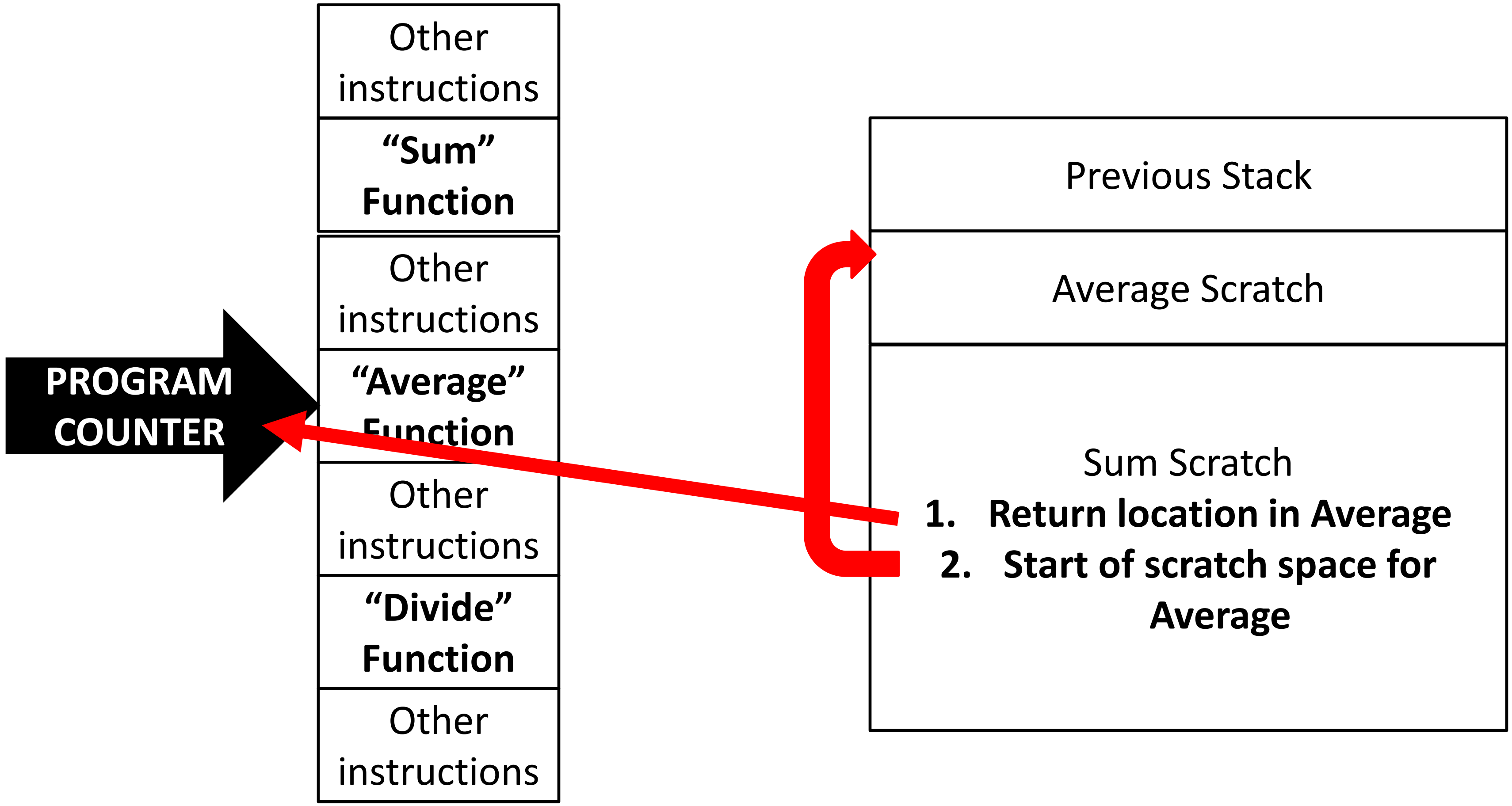
It has to know which instruction to jump ***BACK*** to

Info is stored <u>IN THE STACK</u>

Call Sum from Average



Returning from Sum



What are Buffers?

A buffer is a storage for a sequence of related data

E.g., a program must allocate a buffer for your password

These buffers are often ***stored on the stack***

What are Buffer Overflows?

When a program doesn’t check ***BOUNDS***

That is, the ***INPUT*** is ***TOO BIG*** for the buffer

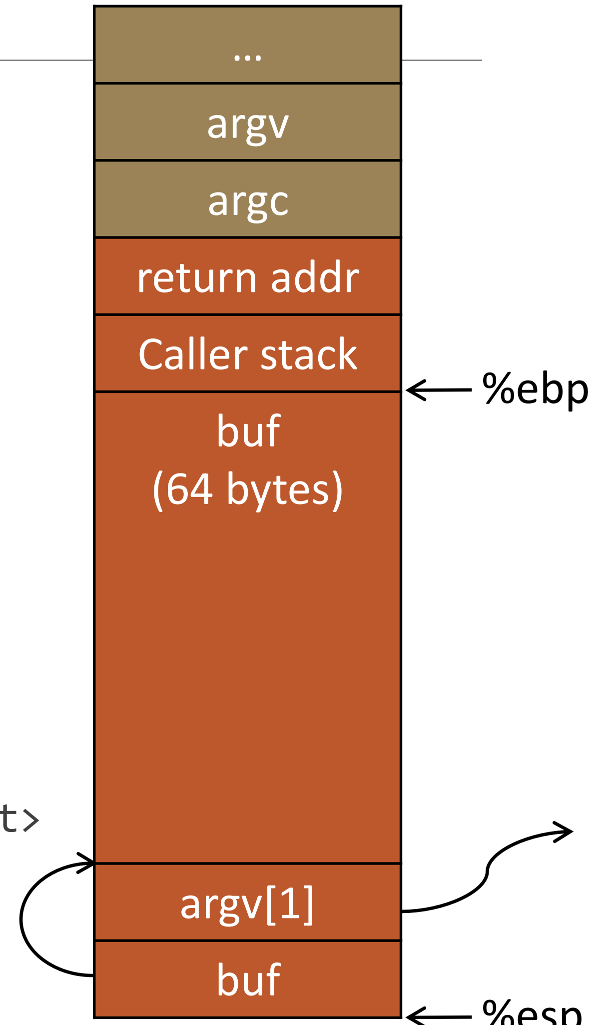
The computer keeps putting more data onto the stack…

<u>OVERWRITING WHAT WAS THERE BEFORE</u>

Basic Example

```c
#include <string.h>
int main(int argc, char **argv) {
     char buf[64];
     strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:      push   %ebp
   0x080483e5 <+1>:      mov    %esp,%ebp
   0x080483e7 <+3>:      sub    $72,%esp
   0x080483ea <+6>:      mov    12(%ebp),%eax
   0x080483ed <+9>:      mov    4(%eax),%eax
   0x080483f0 <+12>:     mov    %eax,4(%esp)
   0x080483f4 <+16>:     lea    -64(%ebp),%eax
   0x080483f7 <+19>:     mov    %eax,(%esp)
   0x080483fa <+22>:     call   0x8048300 <strcpy@plt>
   0x080483ff <+27>:     leave
   0x08048400 <+28>:     ret
```

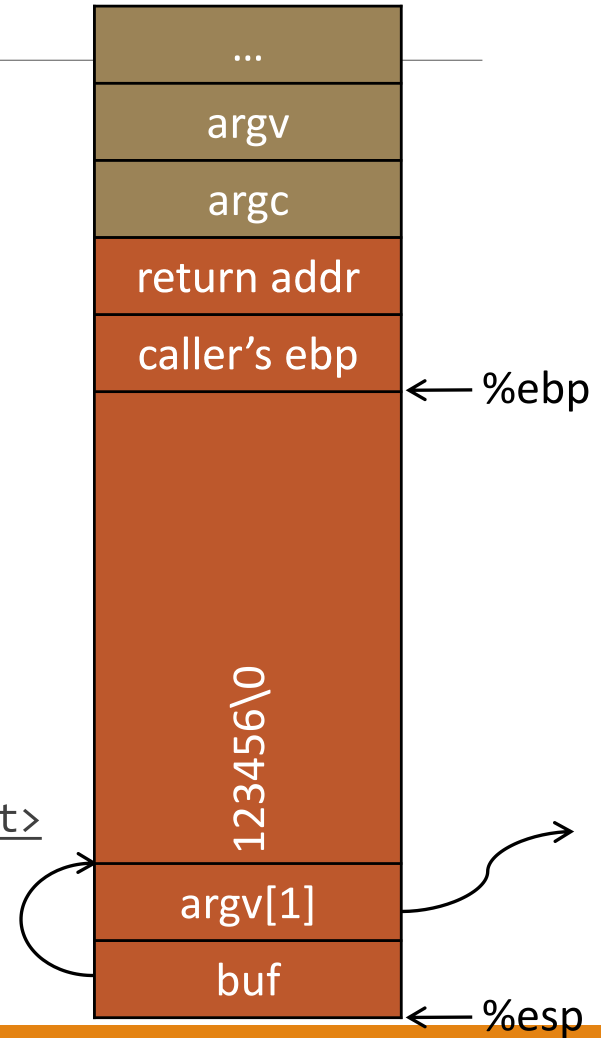


21

“123456”

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $72,%esp
   0x080483ea <+6>:     mov    12(%ebp),%eax
   0x080483ed <+9>:     mov    4(%eax),%eax
   0x080483f0 <+12>:    mov    %eax,4(%esp)
   0x080483f4 <+16>:    lea    -64(%ebp),%eax
   0x080483f7 <+19>:    mov    %eax,(%esp)
   0x080483fa <+22>:    call   0x8048300 <strcpy@plt>
   0x080483ff <+27>:    leave
   0x08048400 <+28>:    ret
```

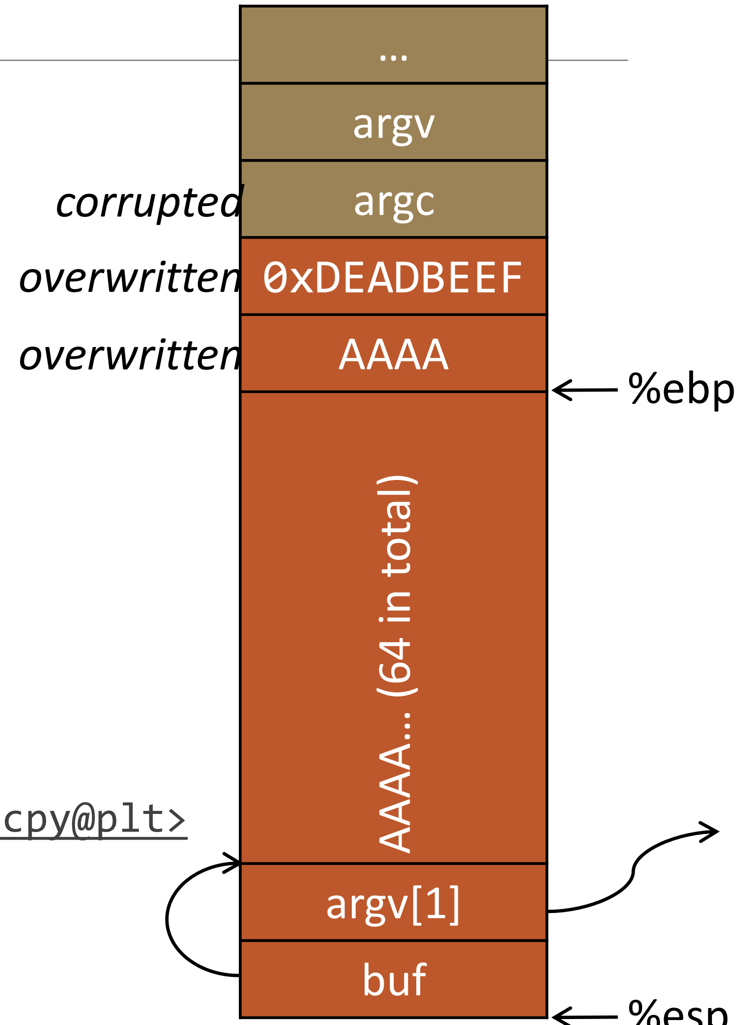


22

“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $72,%esp
   0x080483ea <+6>:     mov    12(%ebp),%eax
   0x080483ed <+9>:     mov    4(%eax),%eax
   0x080483f0 <+12>:    mov    %eax,4(%esp)
   0x080483f4 <+16>:    lea    -64(%ebp),%eax
   0x080483f7 <+19>:    mov    %eax,(%esp)
   0x080483fa <+22>:    call   0x8048300 <strcpy@plt>
   0x080483ff <+27>:    leave
   0x08048400 <+28>:    ret
```



23

Attackers Exploit These

The overflow can overwrite ***THE RETURN ADDRESS***

Attackers can craft inputs that do this

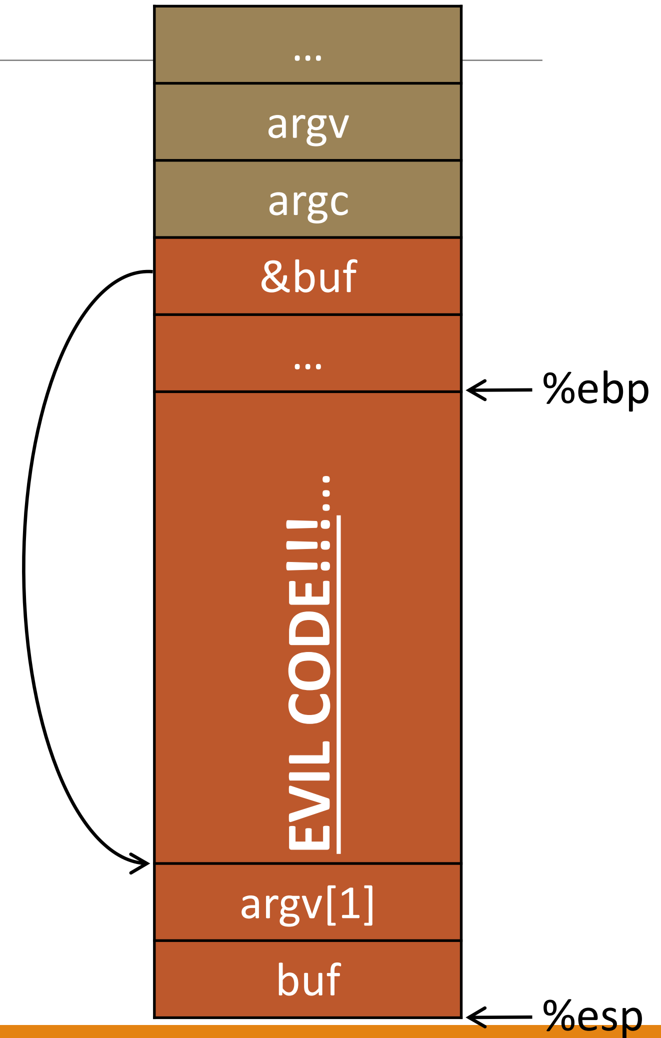
Then they ***HIJACK CONTROL FLOW***

They change where the computer returns to

So, e.g., they jump to a place with evil instructions

Like in the input…

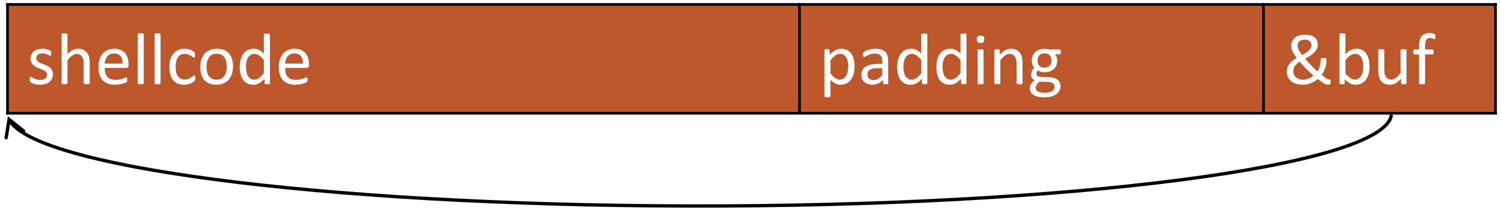
…



Recap

To generate ***exploit*** for a basic buffer overflow:

1. Determine size of stack frame up to head of buffer
2. Overflow buffer with the right size



computation + ***control***

Defenses

Make it harder to control a subverted flow

Make taking control of the flow innocuous

Make it harder to get control of the flow

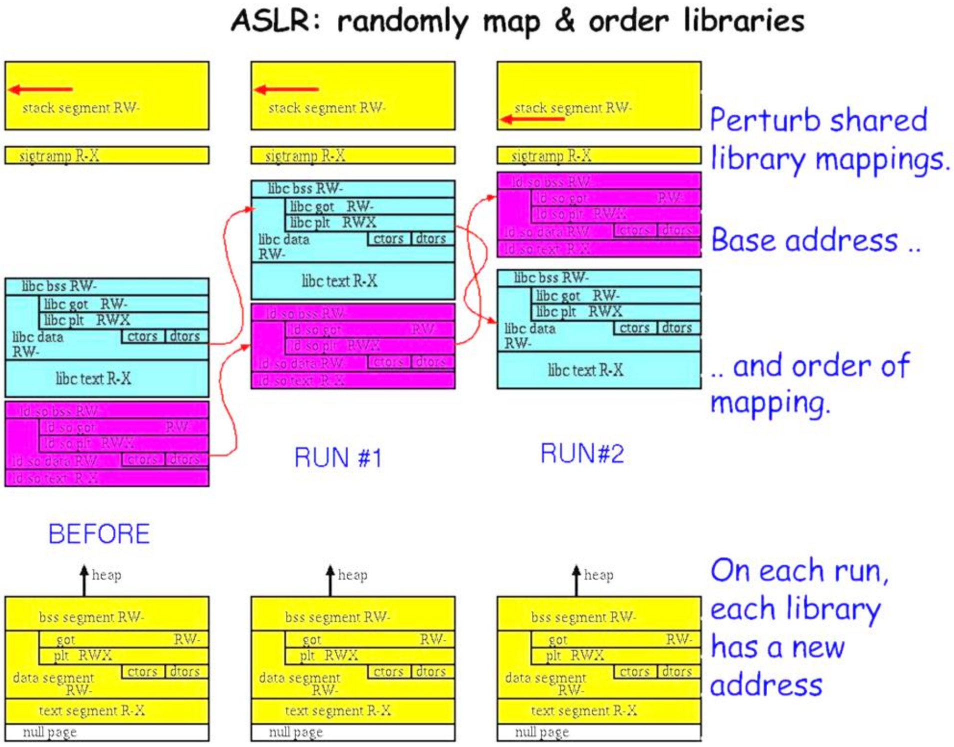
ASLR

Address Space Layout Randomization

Subversion usually needs to know memory layout

General goal: make layout unpredictable

Visualization



Limitations of ASLR

1. **Boot-time based randomization**
2. **Unsupported executables/libraries, low-entropy.**
3. **ASLR does not *trap* the attack**
4. **ASLR does not alert in a case of an attack**
5. **ASLR does not *provide information* about an attack**
6. **ASLR is being bypassed by exploits daily**

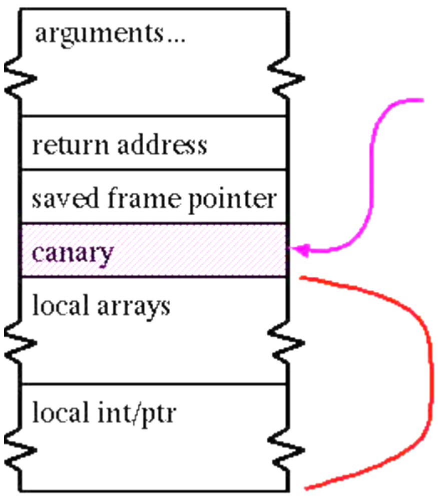
Posted by **MORDECHAI GURI, PH.D.** on December 17, 2015

Making Violations Less Dangerous

W^X Permissions

Finally, Blocking Exploits

Stack Protector



A typical stack frame...

Random value is inserted here by function prologue ...
... and checked by function epilogue

Reordering: Arrays (strings) placed closer to random value -- integers and pointers placed further away

-fstack-protector-all compiled system is 1.3% slower at make build

Image by Dino Dai Zovi

Attacker Oriented Programming?

Behavior isn’t a program

We should be able to perfectly detect bad behavior, right?

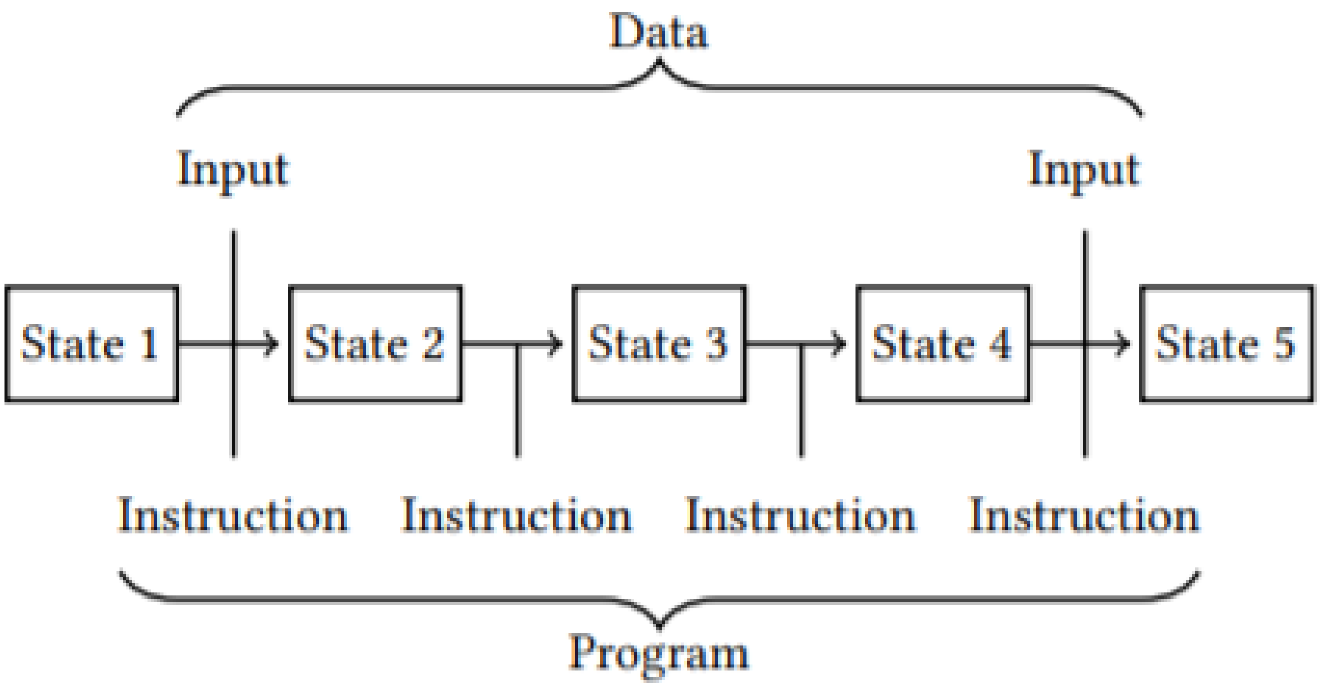
“Weird Machines”

“Weird machines, exploitability, and provable unexploitability”

Written by Thomas Dullien

Explains that users interacting with a program ***is a program***

What is a Program?



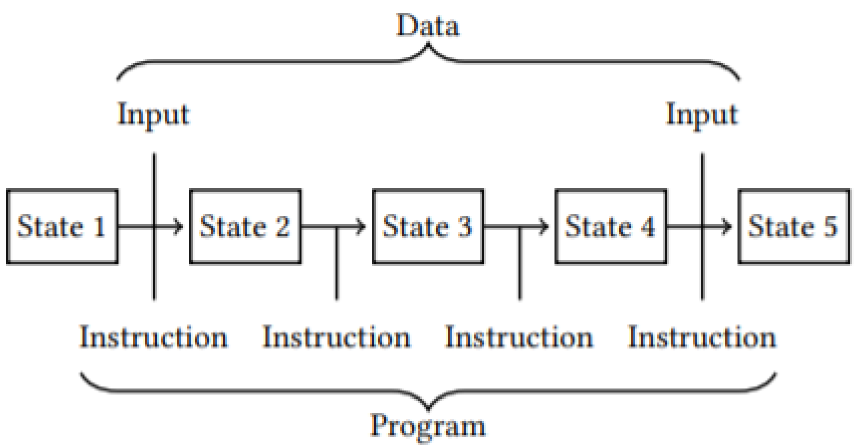
*From Dullien’s Paper

State Machine View

View a “Program” as a state machine

Program starts in state S_0

Based on instruction, advances to state S_i

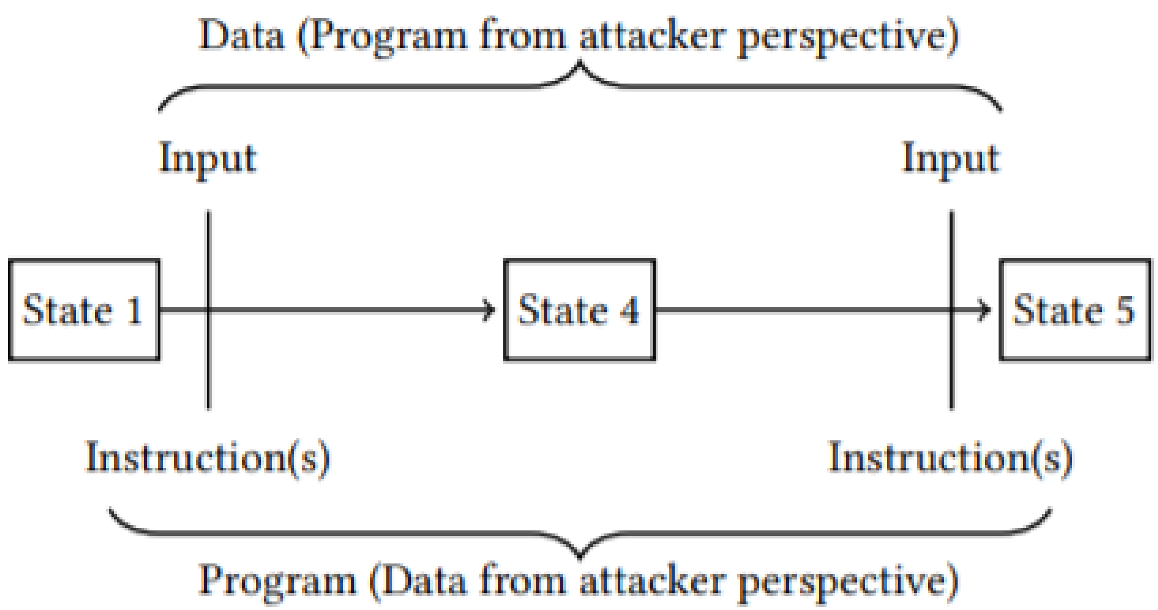


States and User Interactions

Program is in some State. Call it S_0

User interacts with the program

Program advances to state S_1



What is a “User”?

Do we literally mean a flesh-and-blood human?

Really, “user” is just whatever provides the input

This can, of course, just be another process

Thus, two processes interacting ***IS A PROGRAM***

Therefore, determining if “behavior” is good is undecidable

What is “Malicious Code”?

“Software or firmware intended to perform an ***unauthorized*** process that will have ***adverse impacts*** on the confidentiality, integrity, or availability of a system. A virus, worm, Trojan horse, or other code-based entity that infects a host. Spyware and some forms of adware are also examples of malicious code.”

NIST Special Publication 800-53, Revision 5

Common Malware Classes

Malware can be classified by how it spreads or generically behaves

- Virus – typically has to be attached to another program (infection)
- Worm – typically spreads via network vulnerabilities
- Trojan Horse – typically appears benign but contains hostile operations

Malware can also be classified by its behavior

- Spyware – typically designed to steal information, observe behavior, etc.
- Adware – typically designed to "trap" a user into viewing certain ads
- Ransomware – typically locks data unless the user pays a ransom

Two Primary Components

Payload – The code that performs the (harmful) action

Attack Vector/Exploit/Delivery – The code that enables the payload

- May include a transmission component
- May include a stealth component
- May include a mechanism for bypassing security
- May be as simple as an email with an attachment

Malware History

Much of early computer security driven by military concerns

The biggest concern was an unauthorized user or program

Other concerns developed over time

The following slides are a very brief overview/highlight

1972 Government Report:

The technical issue of multilevel computer security is concerned with the concept of malicious threat. By this we recognize that the nature of shared use multilevel computer systems present to a malicious user a unique opportunity for attempting to subvert through programming the mechanism upon which security depends (i. e., the control of the computer vested in the operating system). This threat, coupled with the concentration of the application (data, control system, etc.) in one place (the computer system) makes computers a uniquely attractive target for malicious (hostile) action. Recognition of the implication of malicious threat is important to understanding the security limitations surrounding application of contemporary computer systems. The threat that a single user of a system operating as a hostile agent can simply modify an operating system to by-pass or suspend security controls, and the fact that the operating system controlling the computer application(s) is developed outside of USAF control, contribute strongly to the reluctance to certify (i. e., be convinced) that contemporary systems are secure or even can be secured.

https://apps.dtic.mil/sti/pdfs/AD0758206.pdf

1987: Fred Cohen’s Viruses

“Computer Viruses: Theory and Experiments” Fred Cohen, 1987

Introduced the concept of a self-replicating, evil program

The program attaches to a “good” program infecting it

When the infected program is run, the virus runs

The virus does it’s evil AND spreads itself to other programs

Concepts first proposed by Cohen in 1984

1988: Morris Worm

Robert Morris wrote a self-spreading piece of code (worm)

Spread using exploits in:

- send mail,
- Finger
- rsh/rexec

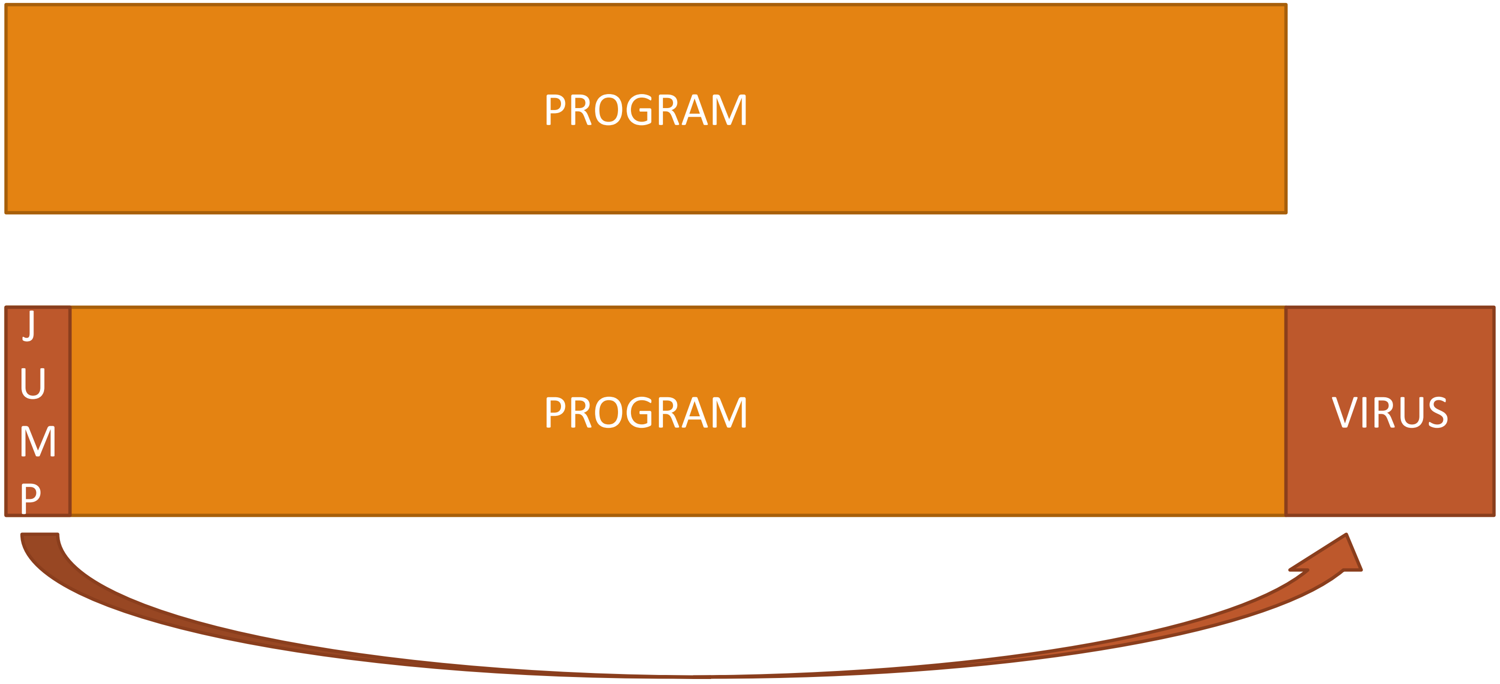
Also guessed weak passwords

Copied code to new machine, compiled, and executed

Accidentally re-infected machines until machines became unusable

DOS attack brought down the Internet

Early DOS Viruses



Impact of Early Viruses

Amazingly, most early malware ***DID MINIMAL DAMAGE***

Often just a delivery system with a weak payload

Many viruses spread for the sake of spreading

Even the Morris worm was disruptive by accident

There were exceptions, but also plenty of hype

1995: Concept Macro Virus

Microsoft Word and Excel have limited scripting ("Macros")

Concept was the first virus written completely as a macro

It was a delivery system only with no payload

But was an important proof-of-concept

Users often open documents directly from email

2000: I Love You

Visual Basic Script virus

Appears as email attachment:

- LOVE-LETTER-FOR-YOU.txt.vbs
- The .vbs often hidden on Windows

When executed:

- Damaged many office files
- Sent email out to email address book ***automatically***

Spread worldwide in hours

2004: MyDoom

Fastest spreading mass mailer virus at the time
- Slows overall internet performance by about 10%
- Slows average web page load times by about 50% percent
- Responsible for approximately one in ten e-mail messages.

Appears as a delivery error, mail error, etc

Includes an attachment that, if clicked on, mails out copies

Also attempted to spread via P2P vile sharing Kazaa

Opened a back door for remote control

Attempted to launch a DDOS against the SCO Group’s website

2005: Sony Rootkit

Sony CD’s from the 2004-2005 era installed a “Rootkit”

- Rootkit, as name implies, usually installs with elevated access
- Using this elevated access, it can change the OS
- This bypasses usual security such as antivirus, etc
- Also usually very good at being undetectable

Installed at root with an EULA ***that did not mention the software***

In 2005, ***US-CERT ISSUED AN ADVISORY!!!***

Texas, ***under Greg Abbot***, was the first state to sue

Why is the Sony Rootkit So Bad?

In addition to violations of privacy, etc, caused:

- Slowing the system, consuming resources
- False alarms from antivirus

OPENED HOLES FOR ADDITIONAL MALWARE

- "Stinx-E trojan"

2013: Cryptolocker

Modern Ransomware

(1980’s had a ransomware called CyberAIDS)

Locks up system and uses public key crypto

In addition to fiat currency, accepted BitCoin

2016: Mirai

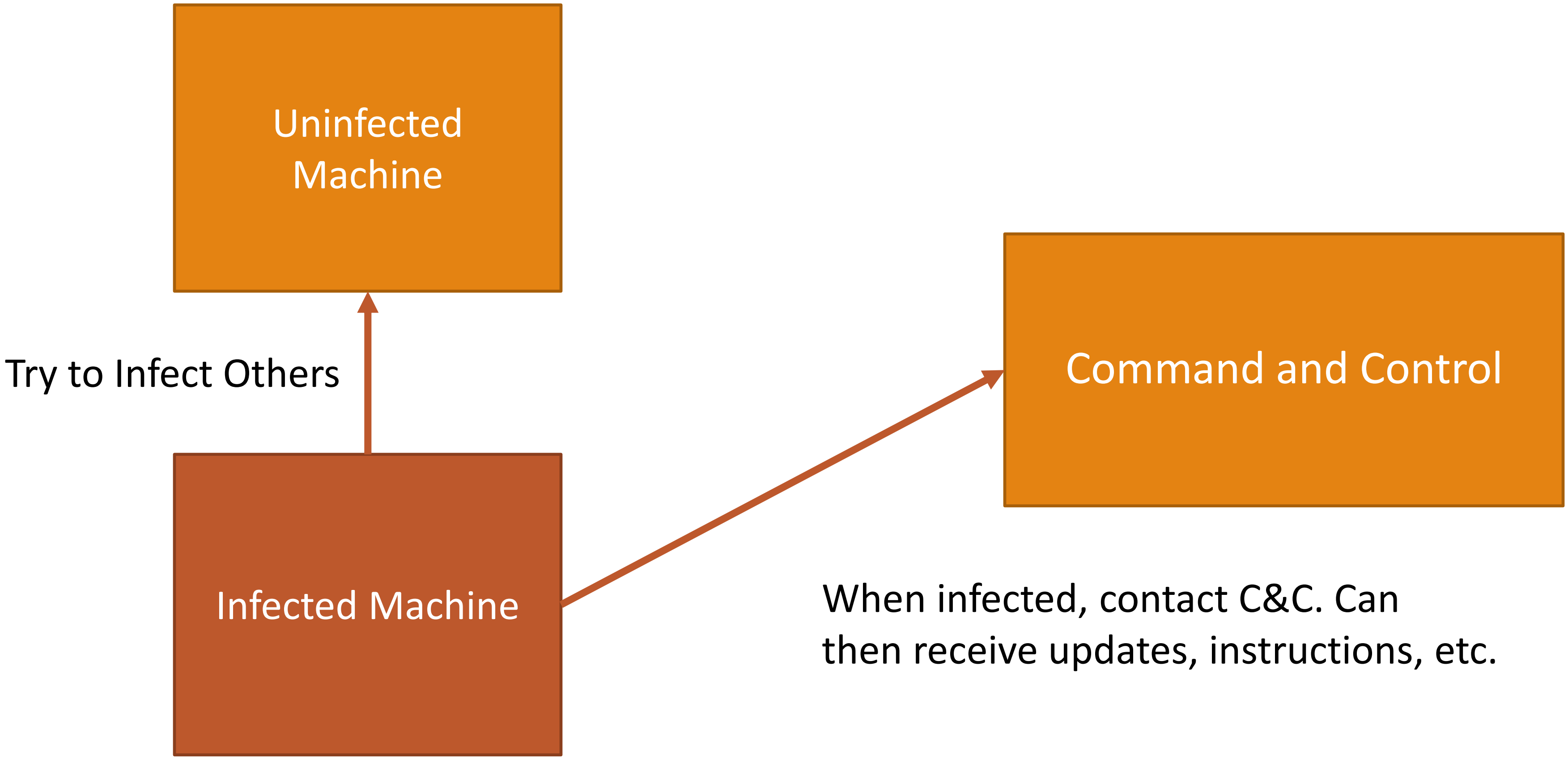
Worm that finds vulnerabilities in IoT devices

Takes over the device (“Zombie”)

Corrdinates all devices with a Command and Control

Launched a powerful DDOS against “krebs on security”

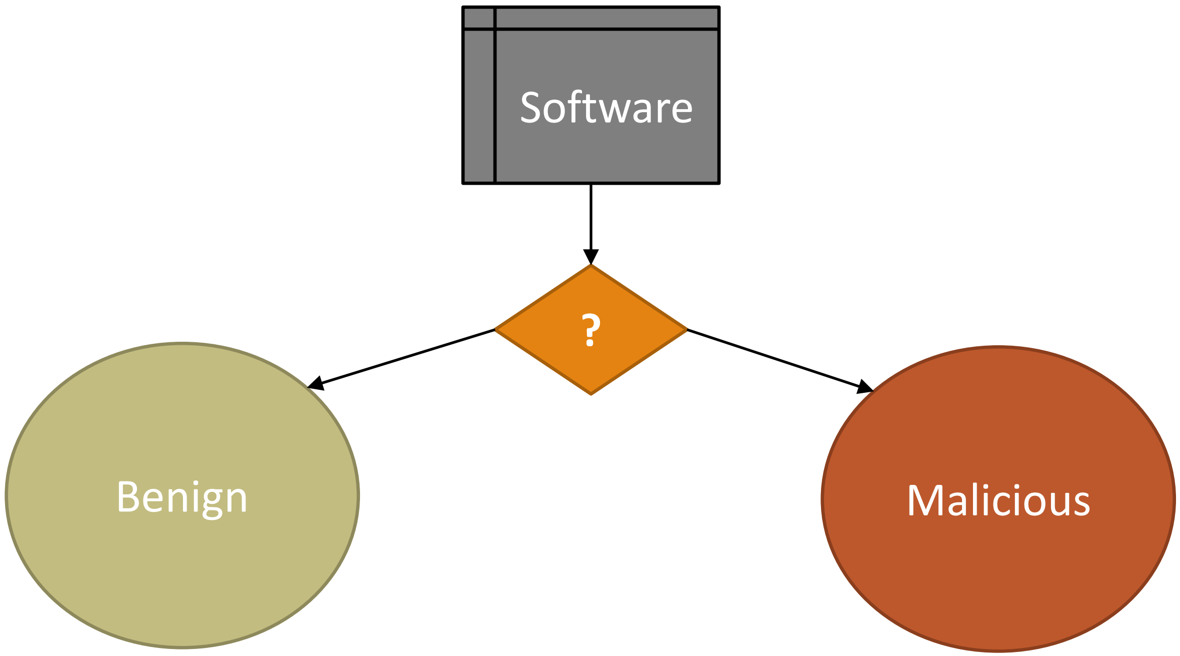
Command and Control Concept



Identify and neutralize	Identify and neutralize malware before the attack
Mitigate	Mitigate malicious activity during the attack
Recover or restore	Recover or restore damaged systems after the attack

Identifying Malware

Primarily a ***classification*** problem



Classification Approaches

Static Analysis

- Analyze the software to categorize it
- Compare against known patterns (signatures)
- Or ~~determine~~ ***guess*** how it will behave (heuristics)

Dynamic Analysis

- Analyze the software's ***execution***
- Identify behavior that violates a security policy
- Or, ~~determine~~ ***guess*** if behavior is dangerous
- Typically in a "safe" container (emulation or sandboxing)

Early “Anti Virus”

“Virus Bulletin” started in 1989

Still available at www.virusbulletin.com

Used to print ***BYTE SEQUENCES*** of known viruses

8 Tunes - CER: The virus probably originates in Germany and infects COM and EXE files. The length of the virus code is 1971 bytes. When triggered, it will play one out of eight different tunes. The virus attempts to deactivate two anti-virus programs: Bombsquad and Flushot+.

```
8 Tunes            33F6 B9DA 03F3 A550 BB23 0353 CB8E D0BC ; Offset variable
```

Virus Bulletin, January 1991

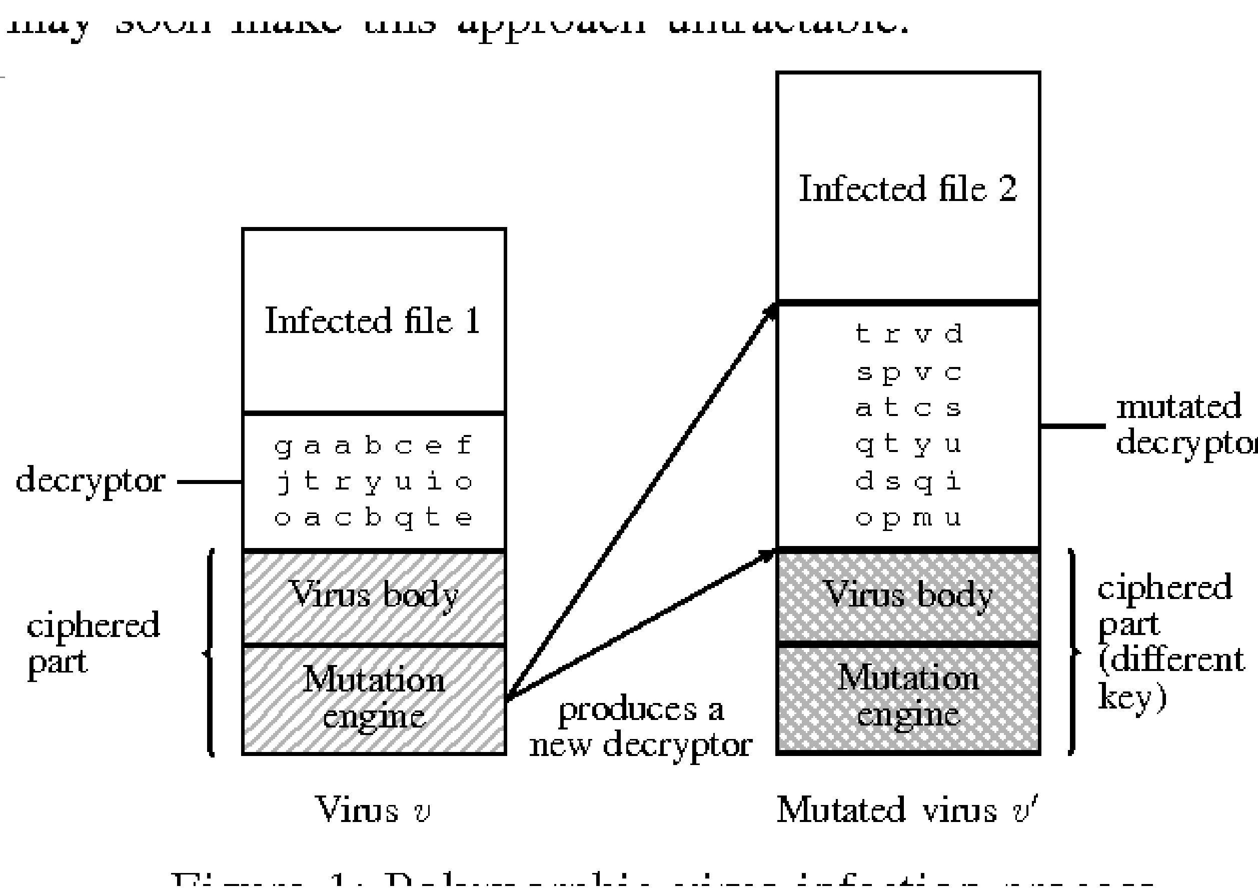
Virus Advancements

Antivirus scanners emerged with “libraries” of virus signatures

In response, viruses became “polymorphic”’

- Each infection encrypts virus under a different key
- Decryption engine decrypts virus for operations
- Encryption means that each infection has unique bytes

Polymorphic Virus Diagram



“Automated extraction of polymorphic virus signatures using abstract interpretation” by Chaumette and Tabary

Anti-virus Arms Race

Advanced Signatures

- Signature is not just a byte sequence
- Each “signature” is a mini-program of detection instructions

Partial Interpreter

- Virus usually takes control early
- Interpret the first bytes to see if its decrypting
- Decrypt and then scan

(Developed in the 1990’s)

Beyond Scanning

Behavior Blockers

- Tries to block bad behavior
- But what counts as “bad”?

Integrity Checkers

- Checksum files
- Detect unauthorized changes
- But what is authorized?

Heuristics

- Look for “telltale” signs
- Minimally effective; too many false positives

Emulation and Sandboxing

Emulation ***simulates*** execution

- Default arguments
- Stubbed I/O
- Simulate the “beginning” when viruses activate

Sandboxing runs the software in a virtual environment

- Default arguments
- Usually for a limited amount of time (e.g., 1 minute)
- Observe changes to the filesystem

Problems

- Non-contextual execution including arguments
- Malware that detects sandboxes

Malware Scanning

Antivirus scanning is representative of all malware scanning

Always “behind” the enemy

Signatures can only catch “known” malware

Guesses always have FP and FN

Dynamic execution can be detected/evaded

Enemy: Halting Problem

Decidability is a classic computer science problem

Halting Problem:

- Given: a program P and input I
- Can you write a program D that determines if P ***halts*** on input I
- (Halts, meaning e.g., not stuck in an infinite loop)
- Over the ***set of all possible programs***, the answer is <u>***NO***</u>
- (Maybe able to determine for some, but not for all)

Alan Turing proved this in 1936!

Halting problem proven to extend to any non-trivial characteristic

In Other Words

There is no program that can detect all malware

Is this just theoretical?

- What if we can detect 99.9999999%?
- What if we can detect all the “important” threats?

Thoughts from 1995

In 1995, Gryaznov wrote, “Scanners of the year 2000”

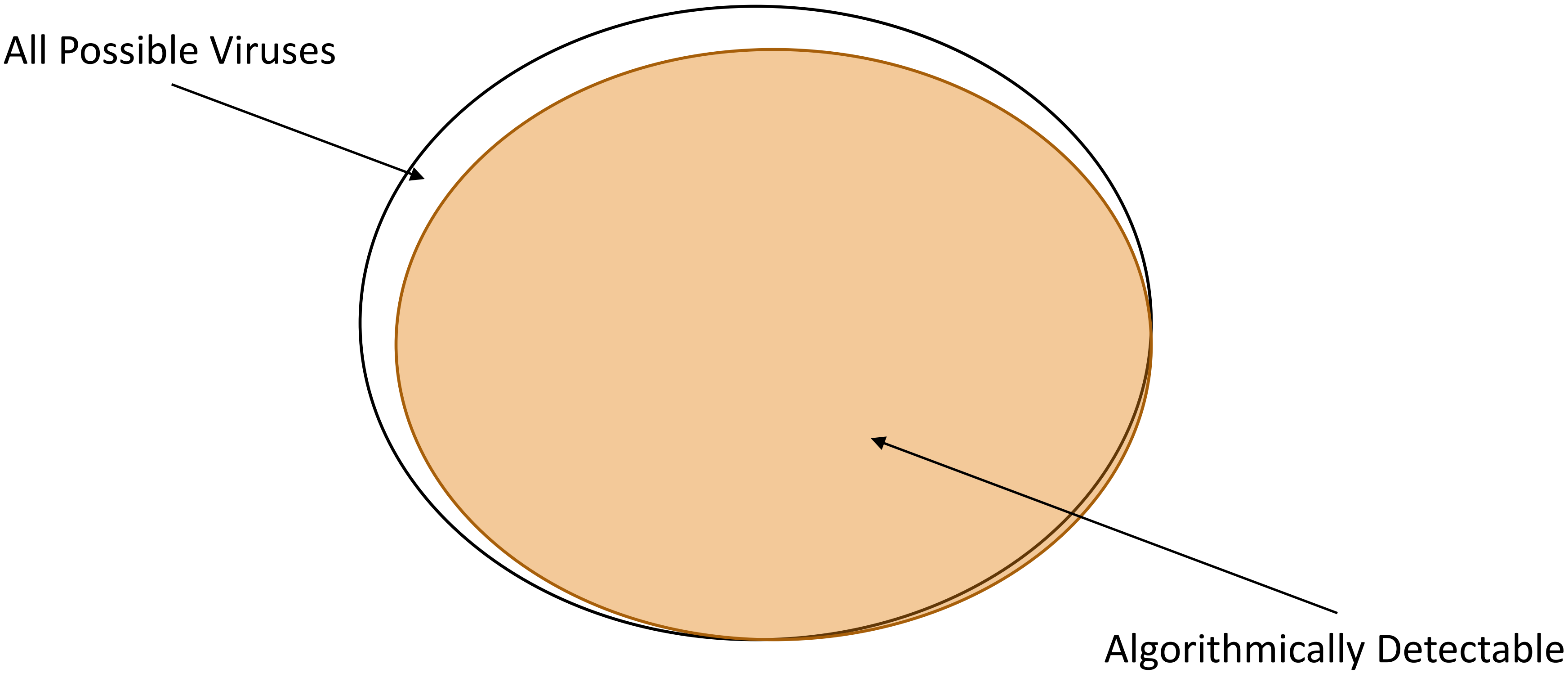
He discussed Heuristics

Specifically mentioned the halting problem, but said:

Fortunately, this does not rule out a possibility of 90 or even 99 per cent reliability. And with the remaining one per cent cases we hopefully shall be able to deal with using *our traditional virus signatures scanning technique.*

<u>GRYAZNOV WAS WRONG.</u>

Gryaznov's View



Why It Doesn’t Work

Gryaznov treats viruses as if they are created at random

Viruses are created by ***human beings***

If an antivirus writer creates an algorithm, the adversary adjusts

The adversary moves into the space not detected by the algorithm

Correct View