CS107, Lecture 14
Reverse Engineering, Privacy and Trust

CS107 Topic 5: How does a computer interpret and execute C programs?

CS107 Topic 5

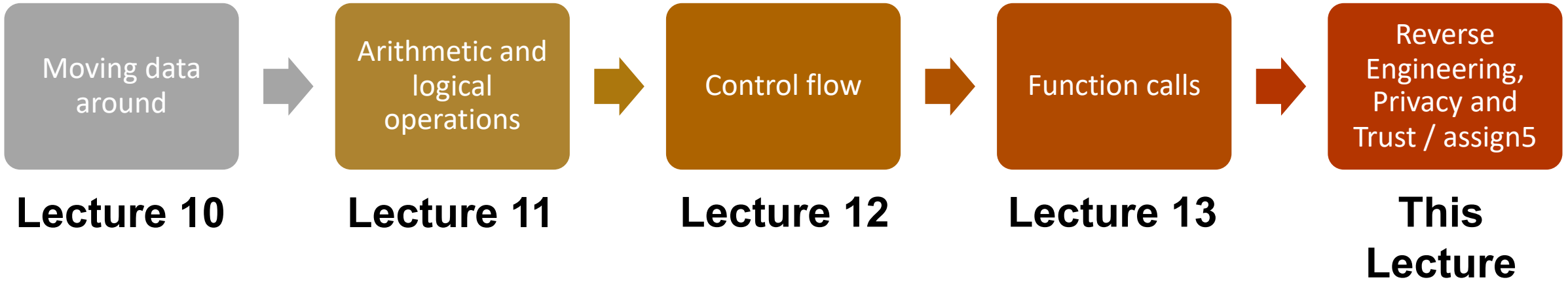
How does a computer interpret and execute C programs?

Why is answering this question important?

- Learning how our code is really translated and executed helps us write better code
- We can learn how to reverse engineer and exploit programs at the assembly level

assign5: find and exploit vulnerabilities in an ATM program, reverse engineer a program without seeing its code, and de-anonymize users given a data leak.

Learning Assembly



Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Learning Goals

- Learn how to approach reverse engineering executables
- Understand the requirements and tasks for assign5
- Learn about the connections between privacy, security and trust

Lecture Plan

- **Recap:** Function Calls in Assembly
- Privacy and Trust
- Assignment 5 Overview
- **Practice**: Minivault

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Lecture Plan

- **Recap: Function Calls in Assembly**
- Privacy and Trust
- Assignment 5 Overview
- **Practice**: Minivault

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Calling Functions In Assembly

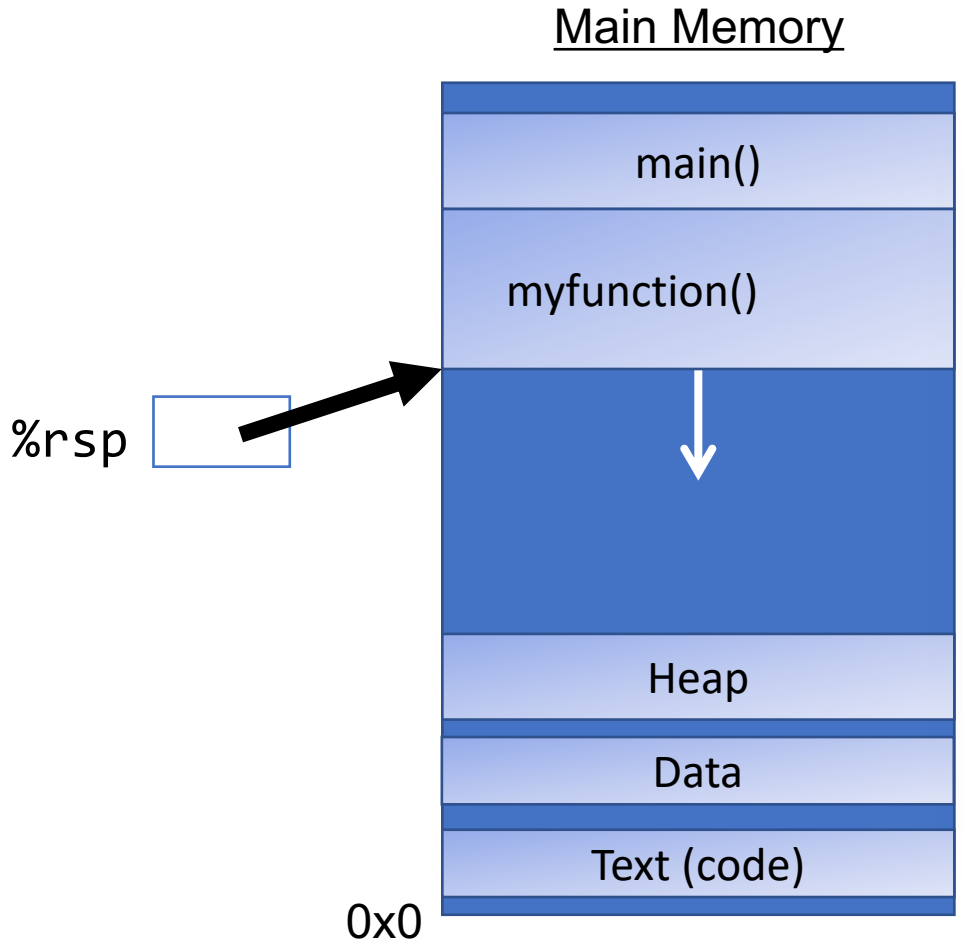
To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



Key idea: %rsp must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.

%rsp

```
func:
  subq $8, %rsp
  movl $10, %edx
  movl $0, %esi
  call strtol
  addq $8, %rsp
  ret
```

Key idea: %rsp must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.

Registers

What does **call** do?

call pushes the next instruction address onto the stack and points %rip to another function's instructions.

Registers

What does **ret** do?

ret pops off the 8 bytes from the top of the stack and puts it into %rip, thus resuming execution in the caller.

ret is separate from the *return value* of the function (put in %rax).

Parameters and Return

- There are special registers that store parameters and the return value.
- To call a function, we must put any parameters we are passing into the correct registers. (%rdi, %rsi, %rdx, %rcx, %r8, %r9, in that order)
- Parameters beyond the first 6 are put on the stack.
- If the caller expects a return value, it looks in %rax after the callee completes.

Local Storage

- So far, we've often seen local variables stored directly in registers, rather than on the stack as we'd expect. This is for optimization reasons.
- There are **three** common reasons that local data must be in memory:
  - We've run out of registers
  - The '&' operator is used on it, so we must generate an address for it
  - They are arrays or structs (need to use address arithmetic)

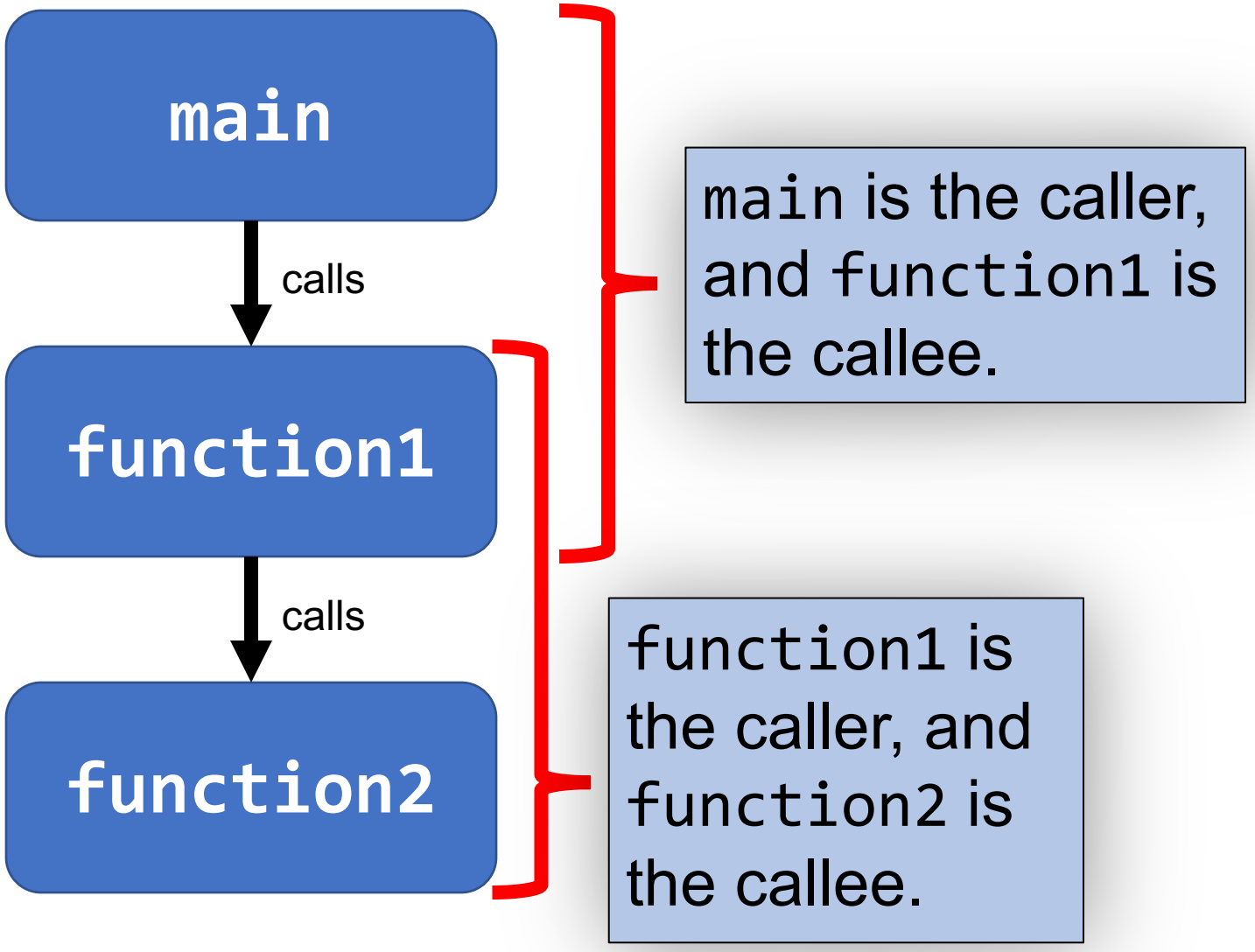
Register Restrictions

There is only one copy of registers for all programs and functions.

- **Problem:** what if ***funcA*** is building up a value in register %r10, and calls ***funcB*** in the middle, which also has instructions that modify %r10? ***funcA***'s value will be overwritten!
- **Solution:** make some "rules of the road" that callers and callees must follow when using registers so they do not interfere with one another.
- These rules define two types of registers: **caller-owned** and **callee-owned**

Caller/Callee

Caller/callee is terminology that refers to a pair of functions. A single function may be both a caller and callee simultaneously (e.g. `function1` at right).



Register Restrictions

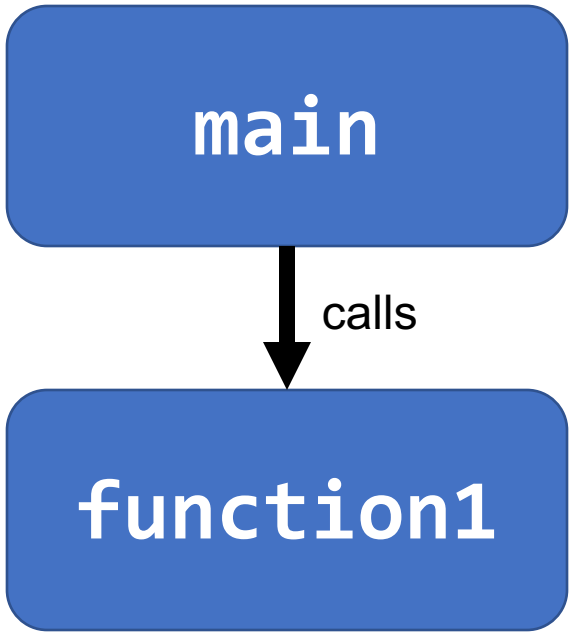
Caller-Owned

- Callee must *save* the existing value and *restore* it when done.
- Caller can store values and assume they will be preserved across function calls.

Callee-Owned

- Callee does not need to save the existing value.
- Caller's values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.

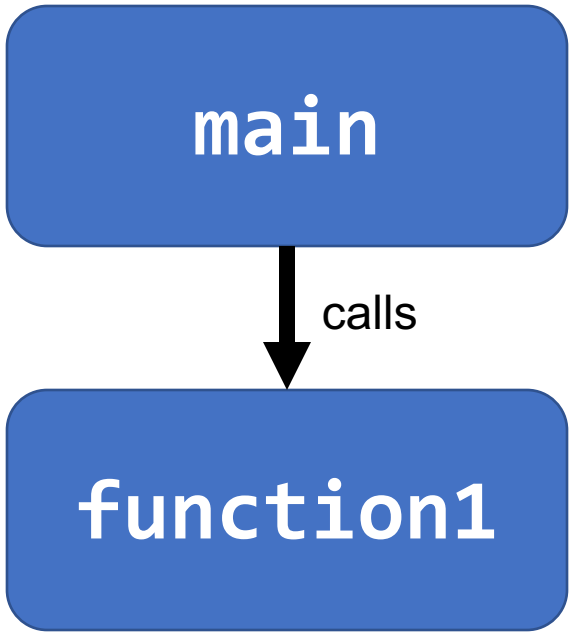
Caller-Owned Registers



`main` can use caller-owned registers and know that `function1` will not permanently modify their values.

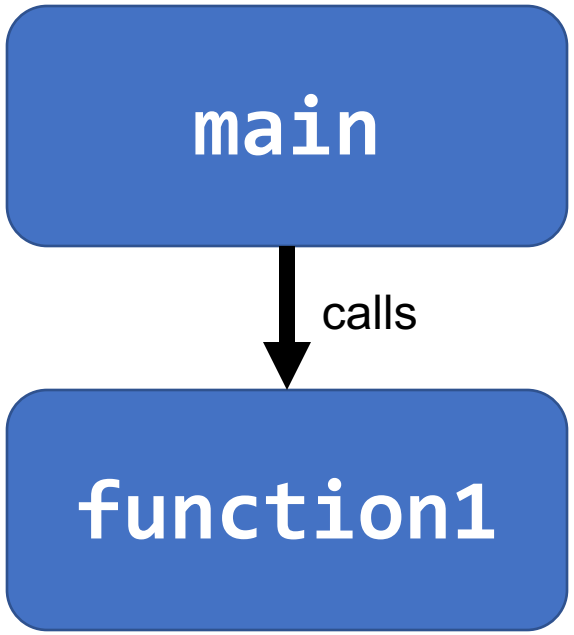
If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.

Caller-Owned Registers




```
function1:
  push %rbp
  push %rbx
  ...
  pop %rbx
  pop %rbp
  retq
```

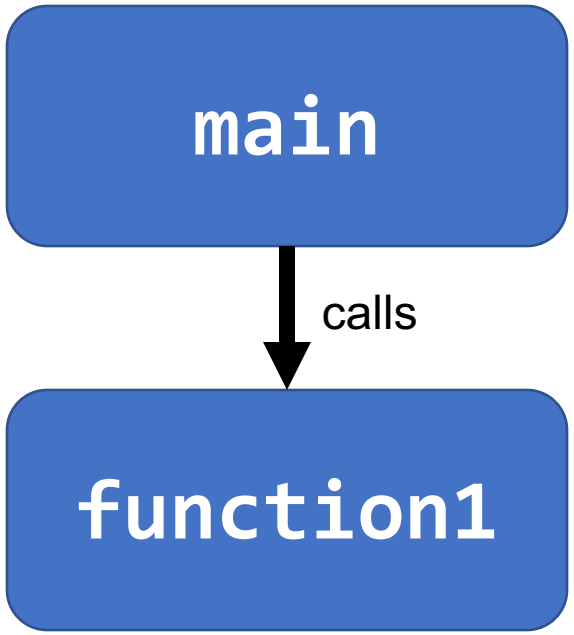
Callee-Owned Registers



`main` can use callee-owned registers but calling `function1` may permanently modify their values.

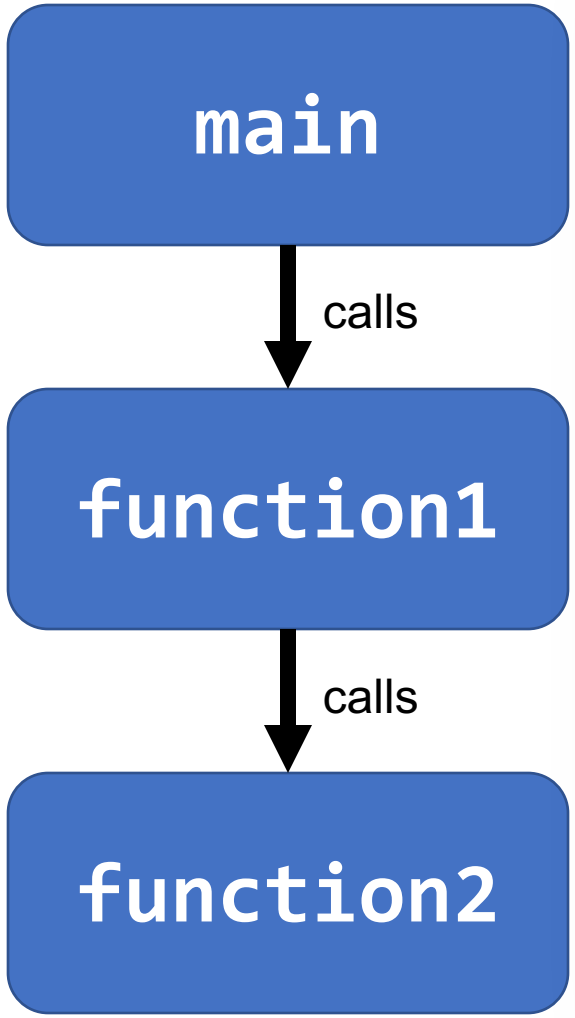
If function1 wants to use any callee-owned registers, it can do so without saving the existing values.

Callee-Owned Registers




```
main:
   ...
   push %r10
   push %r11
   callq function1
   pop %r11
   pop %r10
   ...
```

A Day In the Life of `function1`



Caller-owned registers:

- **`function1`** must save/restore existing values of any it wants to use.
- **`function1`** can assume that calling **`function2`** will not permanently change their values.

Callee-owned registers:

- **`function1`** does not need to save/restore existing values of any it wants to use.
- calling **`function2`** may permanently change their values.

Lecture Plan

- **Recap:** Function Calls in Assembly
- **Privacy and Trust**
- Assignment 5 Overview
- **Practice**: Minivault

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Privacy and Trust

How does a computer interpret and execute C programs?

Why is answering this question important?

- Learning how our code is really translated and executed helps us write better code
- **We can learn how to reverse engineer and exploit programs at the assembly level**

assign5: find and exploit vulnerabilities in an ATM program, reverse engineer a program without seeing its code, and de-anonymize users given a data leak.

Privacy and Trust

- Our learning about assembly and program execution helps us better understand computer security.
- Computer security (the protection of data, devices, and networks from disruption, harm, theft, unauthorized access or modification) is important in part because it enables privacy.
- In understanding computer security, it's essential to understand the context in which it comes up (privacy and trust).

Have you been affected by a data breach/hack or other improper access of your data?

How did that make you feel?

Privacy

What is privacy? 4 possible framings:

- Privacy as **control of information** – controlling how our private information is shared with others.
- Privacy as **autonomy** – capacity to choose/decide for ourselves what is valuable.
- Privacy as **social good** – social life would be unlivable without privacy.
- Privacy (protection) as based in **trust** – privacy enables trusting relationships

First two are *individualist* –the value of privacy as an individual right.

Second two are *social* – the value of privacy for a group.

Privacy

Privacy as **control of information** – controlling how our information is communicated to others.

- Consent requires *free* choice with available alternatives and *informed* understanding of what is being offered.
- How many of you just skip past the terms of service for new online services you sign up for?
- Control over personal data being collected (e.g. data exports from services you use, privacy dashboards, device privacy protections)

Privacy

Privacy as **autonomy** – capacity to choose/decide for ourselves what is valuable.

- Links to autonomy over our own lives and our ability to lead them as we choose.
- Do you feel that your autonomy is always respected when using products and services? Why or why not?

"[P]rivacy is valuable because it acknowledges our respect for persons as autonomous beings with the capacity to love, care and like—in other words, persons with the potential to freely develop close relationships" (Innes 1992)

Individualist Models of Privacy

Privacy as **autonomy** and privacy as **control over information** focus the value of privacy at an individual level.

- Individual privacy can conflict with interests of society or the state.
- Many debates over ”privacy vs. security” – whether one should be sacrificed for the other
  - Apple v. FBI case re: unlocking iPhones (link)
  - Debates around encryption (link)

Privacy

Privacy as **social good** – social life would be unlivable without privacy.

- Privacy has a social value in bringing about the kind of society we want to live in.
- What would society look like without privacy?

Privacy

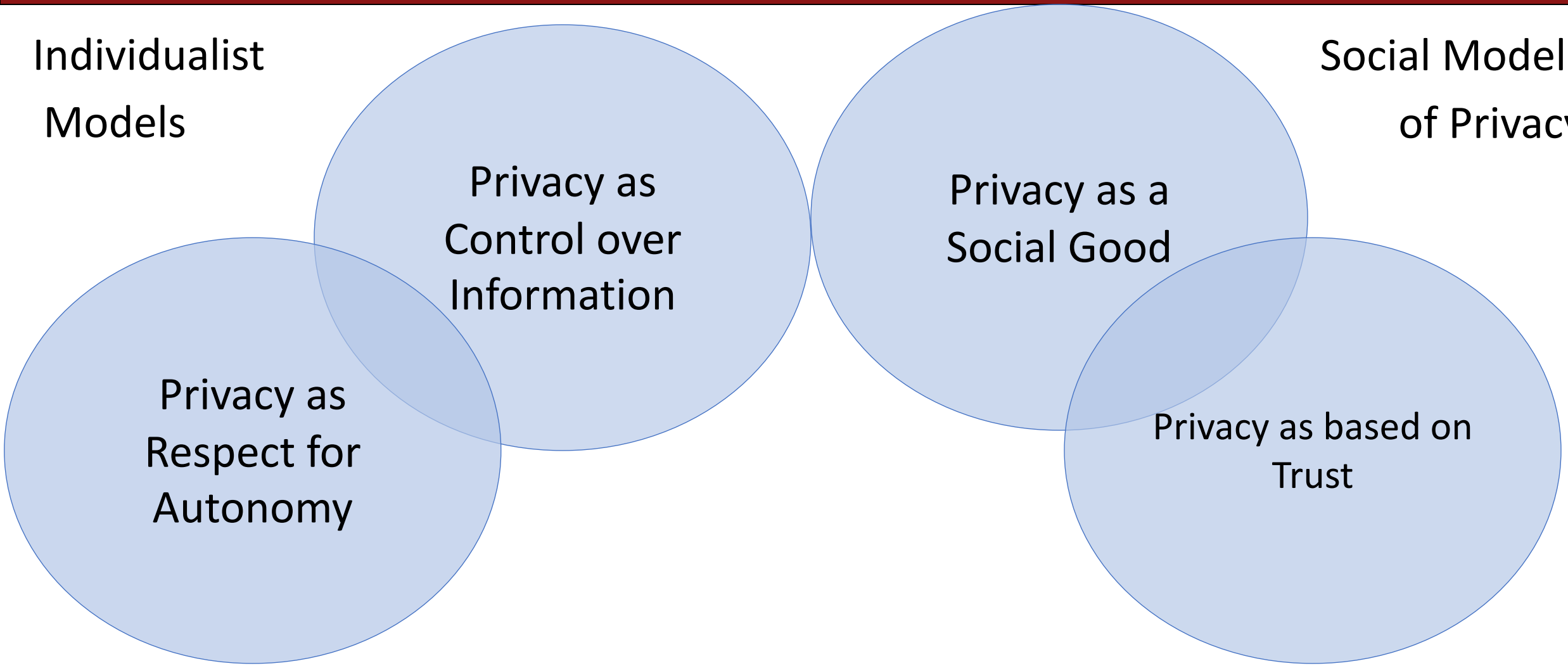
Privacy (protection) as based in **trust** – privacy enables trusting relationships

- Privacy may help enable trusting relationships essential for cooperation. For instance, a *fiduciary*: someone who stands in a legal or ethical relationship of trust with another person (or group). The fiduciary must act for the benefit of and in the best interest of the other person.
  - E.g. tax filer with access to your bank account
  - Should anyone who has access to personal info have a *fiduciary* responsibility? (Richards & Hartzog 2020).
- This model of privacy stresses the essential relationship of trust placed in any holder of personal data and the responsibilities that result from this trust.

Models of Privacy

Individualist Models

Social Models of Privacy



Loss of Privacy

Loss of privacy can cause us various harms, including:

- ***Aggregation*:** combining personal information from various sources to build a profile of someone
- ***Exclusion:*** not knowing how our information is being used, or being unable to access or modify it (Google removing personal info from search – link)
- ***Secondary Use***: using your information for purposes other than what was intended without permission.

Who Should We Trust?

Both security and privacy rely on trusted people (who administer security, perform penetration tests, submit vulnerabilities to databases, or keep private information secret). The final piece of the security puzzle is understanding trust.

Trust = Reliance + Risk of Betrayal

What makes trust unique to relationships between people is that trust exposes one to being *betrayed or being let down* (Baier 1986).

Penetration Testing & Trust

Penetration testing is the practice of encouraging or hiring security researchers to find vulnerabilities in one's own code or system.

The tester is placed in a position of trust: they are given access to the system itself and encouraged to find exploitable vulnerabilities, with the expectation that the tester will share what they have found with you.

Hiring a penetration tester means *relying on* their skill at finding vulnerabilities and also *trusting* that their ethical compass will lead them to tell you and to act as a trustworthy *fiduciary* (guardian of your interests). In Assignment5, you will have the opportunity to test your own ethical compass!

Example: Differential Privacy

Imagine a large database, perhaps a medical database, with personal information and records of past activity tied to a name.

The records might be useful for research purposes, or to train a machine learning model to predict future health outcomes, but what if giving access to the records exposed the privacy of individual person's health records?

Differential privacy is a formal measure of privacy that attempts to address these concerns. By adding inconsequential noise (changing a birthday from 2001 to 2002, for example) or removing records, differential privacy protects individuals from *aggregation* by making them harder to identify (Dwork 2008).

Differential Privacy's Trust Model

Differential privacy assumes that the only threat to privacy is an *external user querying the database* who must be prevented from aggregating data that could identify a user.

In other words, the *trust model* of differential privacy is that the database owners and maintainers are to be fully trusted, and no one else.

Differential Privacy: The Other Threats

But is that the only threat? Differential privacy does not protect against improper use by people with full access to data or against leaks of the whole database, which may be the primary data exposure risks.

Differential privacy also does not question the assumption that amassing & storing large amounts of personal data is worth the risk of inevitable leaks (Rogaway 2015).

In every evaluation of privacy, we can ask: who is trusted? Who is distrusted? Does this model concentrate trust (and therefore power) in a single individual or small group, or does it distribute trust?

Lecture Plan

- **Recap:** Function Calls in Assembly
- Privacy and Trust
- **Assignment 5 Overview**
- **Practice**: Minivault

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

assign5

You are a security researcher hired to explore potential vulnerabilities and issues at Stanford Bank. **3 core parts:**

1. **Uncovering ATM software vulnerabilities**
2. **Demonstrating how a data leak can lead to data aggregation and uncovering of personal information**
3. **Reverse engineering a secure program**

Lecture Plan

- **Recap:** Function Calls in Assembly
- Privacy and Trust
- Assignment 5 Overview
- **Practice: Minivault**

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Optimizations you'll see

`nop`

- **`nop/nopl`** are "no-op" instructions – they do nothing!
- Intent: Make functions align on address boundaries that are nice multiples of 8.
- "Sometimes, doing nothing is how to be most productive" – Philosopher Nick

`mov %ebx,%ebx`

- Zeros out the top 32 register bits (because a `mov` on an e-register zeros out rest of 64 bits).

`xor %ebx,%ebx`

- Optimizes for performance as well as code size (read more here):

```
b8 00 00 00 00          mov $0x0,%eax
31 c0                   xor %eax,%eax
```

Funky Assembly you'll see

Some functions like **printf** take *variable numbers of arguments.*

- It turns out that in assembly when we call these functions, we must indicate the presence of any float/double arguments by setting %rax to the count of vector registers used. If none are used (i.e., no parameters of float/double type), it sets %rax to zero.

Recap

- **Recap:** Function Calls in Assembly
- Privacy and Trust
- Assignment 5 Overview
- **Practice**: Minivault

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Extra Practice

Extra Practice – Escape Room 2

https://godbolt.org/z/8e31fG4r5

`escape_room`

Escape room assembly code

```
000000000000115b <escape_room>:
    115b:  48 83 ec 08           sub    $0x8,%rsp
    115f:  ba 0a 00 00 00        mov    $0xa,%edx
    1164:  be 00 00 00 00        mov    $0x0,%esi
    1169:  e8 d2 fe ff ff        callq  1040 <strtol@plt>
    116e:  48 89 c7              mov    %rax,%rdi
    1171:  e8 d3 ff ff ff        callq  1149 <transform>
    1176:  a8 01                 test   $0x1,%al
    1178:  74 0a                 je     1184 <escape_room+0x29>
    117a:  b8 00 00 00 00        mov    $0x0,%eax
    117f:  48 83 c4 08           add    $0x8,%rsp
    1183:  c3                    retq
    1184:  b8 01 00 00 00        mov    $0x1,%eax
    1189:  eb f4                 jmp    117f <escape_room+0x24>
```

Escape room assembly code

```
0000000000001149 <transform>:
    1149:  8d 04 bd 00 00 00 00  lea    0x0(,%rdi,4),%eax
    1150:  8d 50 01              lea    0x1(%rax),%edx
    1153:  83 fa 32              cmp    $0x32,%edx
    1156:  7f 02                 jg     115a <transform+0x11>
    1158:  89 d0                 mov    %edx,%eax
    115a:  c3                    retq
```