Here are some excerpts from a **software security** class that included **software testing tools** as well.

**Memory Based Attacks:**

**Low Level Security:**

Consider low level software security,

which is a concern for systems written

in the C and C++ programming languages.

We will begin by considering

the infamous buffer overflow attack,

which low level software is

vulnerable to in particular.

What is a buffer overflow?

A buffer overflow is a bug that affects

low level code, typically written in C and

C++ with significant

security implications.

Normally a program with

this bug will simply crash.

But an attacker can alter the situation

and cause the program to do much worse.

Allowing the attacker to steal private

information to corrupt important data and

even to run code of his choice.

It is worth studying buffer overflows for

several reasons.

First, they are still relevant today.

C and

C++ are used to write a lot of software.

And that software often has

buffer overflows vulnerabilities.

Lessons we learn here will be relevant

to other software weaknesses.

Considered altogether,

the evidence puts C and

C++ as two of the top three

languages used today.

Therefore, any vulnerabilities

particular to these languages, as buffer

overflows are, are quite relevant to

a good understanding of cyber security.

What software is written in C and C++?

Some examples include operating system

kernels, high performance servers,

such as, web servers and

data base servers.

And embedded systems which appear in cars,

airplanes, industrial control systems and

even the Mars Rover.

These systems are all

of critical importance.

They are the platform for

computing and they drive our economy and

ourselves from here to there.

A successful attack on these systems

has tremendous consequences.

how buffer overflows work and

then learn various ways

to defend against them.

For a complete understanding, we'll

need to learn how a compiler produces

executable code from C source programs.

And how the operating system and

architecture work together

to run these programs.

We'll see that knowing these details

an attacker can exploit bugs and

how the program utilizes memory

in order to attack that program.

In general security often

requires a whole systems view.

And our study here will

be an example of that.

Before we can talk about

how buffer overflows work,

we need to review some details about how

you run a program on a modern computer.

For understanding buffer overflows,

we're particularly interested in how

programs are laid out in memory.

We will consider where the program code

and its data are located in memory.

We will look at the call stack and

how it stores arguments and

local variables of functions

when they are called.

We will look at some of the metadata

that is stored amongst this program data

to make it easier for

the compiler to generate code that can

be used in different circumstances.

For example, no matter which

function calls which other function.

In our discussion we focus on the Linux

operating system process model running on

an Intel X86 32 or 64-bit processor.

While the details differ for

different operating systems and

architectures, the concepts that

we will consider are very similar.

All programs are stored in memory.

A program when it begins

running is called a process.

And that process is given memory by

the operating system in order to run.

Here, we depict the process

as address space.

At the bottom is address 0, the lowest

address, and at the top is the address at

four gigabytes which is the highest

address on a 32 bit system.

The process' view of memory

is that it owns all of it.

As far as it can tell,

it's the only program running on a system.

In reality, these are virtual addresses

that the operating system and

processor map to actual physical

addresses for the memory on the machine.

At the bottom of the address base

is the text segment or code.

Here we see some x86 instructions that

might make up the code of our program.

Just above the text segment is the data

segment, and it has two parts.

The first is the initialize data area.

So here we see a variable y

that's initialized to 10.

And above that is

the uninitialized data area.

Here, the variable x that's

not initialized at all.

All of this data is known at compile time,

so

the compiler can determine where it goes

and specify as much in the executable.

At the top of the address space

comes the command line arguments and

environment variables, and

these are set when the process starts.

Just below them is the stack, the stack

is what holds local variables along with

metadata that the program uses to call and

return from functions.

Above the data segment is the heap.

This is the area that Malik manages.

All of this data is organized and

managed at run time.

That is, how it behaves depends on what

the program does, what it interacts with,

what input files it reads or

writes and so on.

Now we've turned the picture on it's side,

so the lowest address is to the left and

the highest address is to the right,

and we'll use this orientation for

most of the rest of the slides.

Here again we see this stack and

the heap depicted, and

we also show the direction that they grow.

As more memory is needed in the heap, it

grows towards the higher addresses, where

as more memory is needed for the stack, it

grows downward toward the lower addresses.

While the program is running,

it maintains a stack pointer which

indicates the top of the stack.

When the program issues

a push instruction,

it will move the stack pointer

after pushing the value.

Now suppose that after running for

a while, the function that had

pushed these values returns.

In that case, we expect that the function

will pop a large portion of the stack off,

removing all of its local variables and

arguments.

We'll see how this works

exactly in a minute.

The Compiler emits the instructions

that adjusts the stack at run-time.

Likewise code, that is the implementation

of malloc, keeps track of the heap.

The memory that the heap uses

is apportioned by the OS, but

the individual data that's stored

inside the heap is managed by malloc.

For now, we're going to focus on the stack

because that's the target of the first

attack that we'll consider.

The next question is, how does a program

use the Stack while it is running?

As mentioned earlier,

the Stack is used to support calling and

returning from functions.

We'll now look at the details.

In particular, we'll look at

what data we need to store, and

where we'll put it when

calling a function.

We'll also look at what has to

happen when a function returns.

That is, what data needs to be

restored and where to get it from.

Now let's consider the basic stack layout.

Here we see a simple function, func, that

takes three arguments, arg1, arg2, and

arg3, and has two local variables,

loc1 and loc2.

Below we see the depiction of

the memory of the process.

The highest addresses are to

the right as usual, and

we see a depiction of the caller's data.

That is, the caller of this function.

When the caller goes

to call this function,

it's going to push the arguments

in reverse order of the code.

So remember, the stack grows

from the right to the left.

That is,

the top addresses to the bottom addresses.

So we see then that arg3 comes first,

then arg2, then arg1.

That is,

the opposite order of the program.

Now, the local variables of the functions

are al, are accessed on the stack as well.

And they are stored in the order that

they appear in the program text.

That is, first loc1, and then loc2.

There are a couple bits of information

that are stored in between,

and we'll see what these are in a moment.

Now, suppose the compiler is generating

code to access these variables.

So here we show that within

the function it wants to

increase the value of loc2 by one.

How will it do this?

Well, in order to do it, it needs to

know where loc2 is stored on the stack.

Suppose for argument's sake that it's

stored at this particular address,

how will the program know that?

Well, if we think about it,

if this function could be called from

many different places in the program,

the actual address of loc2 could differ

depending on who called the function.

Therefore, the compiler could not know

this address at compile time, and

it's going to need to do something else.

Fortunately, the compiler always knows

the relative address of this variable.

That is, it's always 8 bytes before

the question marks here on the stack.

Stepping back, we can think of all of

this stuff that's highlighted in blue

as the stack frame for the function.

The arguments and the local variables

plus these extra question marks that

we'll get to in a minute.

Now, because we want to know how to locate

local variables, and for that matter how

to locate arguments, we need a reference

point within the Stack frame.

We'll call that the Frame pointer.

Typically, compilers store the Frame

point in the ebp register.

Therefore, the compiler knows that no

matter where this function is called from

it will always be 8 bytes distant from

the current value of the frame pointer.

Now let's see how we implement

returning from functions.

Here we see main which is called

the function func that we were just

looking at, and we see the stack frame for

func here at the bottom of the slide.

Here's the caller's data

from main that we saved.

Now, when we called func, main was using

the frame pointer, just as func is,

to access its own local variables.

When we return from func,

main is going to want to use the same

frame pointer that it had before so

that when it goes to access its variables

it's going to the right addresses.

So the question is, how do we save and

restore the frame pointer so

that this works properly?

Well, let's think about how main is

going to call func in the first place.

What it will do is it will push

its three arguments, arg3,

arg2, arg1, here hey, 10, minus 3.

It'll push some other data

that we'll see in a minute.

At this point,

the stack frame pointer is right here.

Now what we can do is we can save main's

frame pointer right on the stack.

At this point, we can update the frame

pointer to be the current stack pointer.

And now when the func

function starts to run,

it will push its local variables

after the current stack pointer.

And here we are from where we started.

The next question is,

how do we resume at the same place that

we were in in main when we called func?

Here's what's going on.

As main is running,

the instruction pointer,

eip, is moving through the different

instructions that implement main.

Now it goes to call func.

When it goes to call it, the instruction

pointer is going to move up and

start executing these

different instructions.

So what we want is to resume back to where

we were when we called the function.

Well, we can play the same trick

that we did with the frame pointer,

we can store the instruction pointer just

before calling the function on the stack.

Now, when we go to return, we just have

to set the instruction pointer to 4

off of the current frame

pointer in the call e.

In summary, when calling a function,

we push arguments onto

the stack in reverse order.

Then we push the return address, and

then we jump to the function's address.

Within the called function, we push

the old frame pointer onto the stack.

We set the new frame pointer value to

be where the stack is right now and

then push the local variables in order.

Finally, when returning, we reset the

previous stack frame by updating the frame

pointer, and then we simply jump back

to the instruction pointer that we

saved on the stack which is four

more than the current frame pointer.

**Buffer Overflows:**

Now that we're refreshed on the basics of

how C programs are laid out in memory.

In particular how they use

the stack to support calling and

returning from functions.

We can start looking at

buffer overflow attacks.

Let's look at the components of the name.

A buffer is simply

a contiguous region of memory,

associated with a program variable or

field.

When they use the term buffer,

people are often thinking of strings.

Where a string is simply an array of

characters ending with a null or zero.

For now, we will focus on strings too.

Later, we will consider

a format string of text.

And in the process see how the idea of

a buffer is actually quite general.

An overflow occurs when the program tries

to write more data to a buffer than it

can actually hold.

This term is evocative of data

running off the end of the buffer.

But once again,

the idea is really more general.

Basically, whenever the program

tries to use a variable to

access memory that doesn't belong

to that variable, for example, by

indexing an array out of its bounds, the

program is performing a kind of overflow.

An important question is,

what happens when the program reads or

writes to a buffer outside its bounds?

According to the C programming language

standard, such a program is undefined.

Effectively it is allowed to do anything.

In a move positive for security,

the compiler could choose to insert

code to detect out of bounds accesses.

And terminate the program when they occur.

Instead most compilers simply assume the

program does not have any overflows and so

the program will access whatever

memory that happens to be at

the accessed location.

By knowing how memory is laid out,

an attacker can use out of bounds

accesses to his advantage.

But probably you can see the problem here.

The string has seven characters

plus a null terminator,

whereas the buffer in the local

function only allots four characters.

And so we're going to overflow

that buffer when we call strcopy.

Let's see this depicted on the stack.

First, when calling funct, we see arg1.

We see the frame, the instruction

pointer that we saved from the caller.

And we see the same frame pointer.

Then we see the buffer of the four bytes.

That we allocated inside of Funk.

Now when the stir copy works, it's going

to copy the first four characters.

Then it's going to overwrite

the frame pointer with the remaining

four characters.

When we get to the end of the function,.

We're going to try to follow our similar,

the same process we always do in trying to

return to the function, the caller main.

Of course,

the frame pointer is now corrupted.

So it's going to set it to

whatever this strange value is.

And we're going to segmentation fault.

Now, normally we just think,

oh, that's a crash.

There are bugs in the program.

This is one of them.

Who cares?

Eventually, we'll discover it and

we'll fix it.

Well buffer overflows

are security relevant.

The software design process aims to produce a software architecture according to good principles and rules. This is an iterative process. We first put down our initial design. And then we perform a risk based analysis of that design. As a result, we may determine that the design needs to be improved. And so, we apply our principles and rules and then improve until we are satisfied. A principle is a high-level design goal with many possible manifestations, while a rule is a specific practice that is consistent with sound design principles. Now the difference between these two can be fuzzy, just as the difference between design and implementation is fuzzy. For example, there is often a principle underlying specific practices. Principles may also sometimes overlap. The software design phase tends to focus on principles for avoiding flaws and less on rules which are more part of the implementation phase. We can group our set of software design principles into three categories. *The first category is prevention.* These sorts of principles aim to eliminate software defects entirely. Some bugs would have been prevented by using a type-safe langauge, like Java, which prevents outright buffer overflows, but some low level code that controls hardware often cannot use that language. *The second category of principle is mitigation.* These principles aim to reduce the harm from exploitation of unknown defects. That is, we can assume that some defects will escape our elimination during the implementation phase. But if we design our software in the right way, we may make it harder for an adversary to exploit these defects in a way that's profitable to him. As an example we could run each browser tab in a separate process. So that exploitation of one tab does not yield access to data that is stored in another tab because it is isolated by the process mechanism. *The third category of principle is detection and recovery.* The idea here is to identify and understand and attack, potentially undoing its damage if we're able to recover. As an example, we can perform monitoring of the execution of our program. To try to decide whether an attack has taken place or might be in the process of taking place. And we could perform snapshotting of various stages of our software's execution. In order to revert to a snapshot if we discover later that an attack has corrupted our system.

Our first design principle aims

to help this understanding.

In particular, we want to keep the system

as simple as it possibly can be.

Ideally, it can be so

simple it's obviously correct.

The goal of simplicity applies

to all aspects of the system,

from the external interface to

the internal design and to the code.

Favoring simplicity is consonant with the

classic principal, economy of mechanism.

And its goal is to prevent attacks from

being successful by avoiding defects in

the first place.

The next category of secure design

principles is trust with reluctance.

A system is constructed from many parts.

And the security of the whole system thus

depends on the security of each part.

The parts are trusted to varying degrees.

And as such, we can improve

the security of the overall system by

reducing the amount of trust

placed in its various parts.

And there are several ways to do this.

First, we could use a better design.

Second, we could implement

functions more safely.

For example by using a type safe language.

Third, we could avoid assumptions

on which we base our trust.

For example, we could avoid the assumption

that the third party libraries we

use are actually correct.

And instead we could perform some

of our own testing and validation,

code review, to ensure correctness and

thus establish trust in those libraries.

As another example we

could avoid designing or

implementing security-critical

features that require expertise that

we don't necessarily have.

A key mistake made by many system

builders is to attempt to design or

implement cryptography from scratch.

Something that it turns out

is very hard to get right.

The principle of trusting with reluctance,

when followed, can help prevent

security problems, and can mitigate

problems that might otherwise arise.

One secure design principle that

follows the idea of trusting with

reluctance is to maintain

a small trusted computing base.

The trusted computing base comprises

the system components that

must work in order for

the system to behave securely.

As an example modern operating systems are

often part of the trusted computing base

because they enforce security policies

like access control policies on files.

When you have a multiuser system and

your files indicate which users are

allowed to read and write the file it is

the responsibility of the operating

system to enforce those policies.

Now we would like to keep

the trusted computing base small and

simple to reduce its overall

susceptibility to compromise.

The idea is that a small simple

trusted computing base is easy for

a human analyst to reason about and

argue that it is correct.

It should also be easier for

an automated tool to establish that

the trusted computing base is correct.

Unfortunately today operating system

kernels, returning to our example,

are not small and simple.

Instead, they are often

millions of lines of code.

Usually the reason is that by keeping

all of these different disparate

components together in the kernel,

we can have higher performance.

But we increase the attack

surface of the kernel and

therefore increase the risk of compromise.

For example, a buggy device

driver could be exploited and

thereby give the attacker

access to the entire kernel.

The kernel's trusted computing base, that

is the parts enforcing security policies,

could therefore be subverted by

the attack, by the device driver.

An alternative design is to

minimize the size of the kernel so

as to reduce the trusted computing base.

For example, device drivers could

be moved outside of the kernel.

And therefore, a compromise of

one of these drivers does not

compromise the portion of functionality

that enforces security policies.

This sort of design is advocated

by microkernel architectures,

which were popular in the 90s, and

are regaining popularity today.

As another example of a failure to

follow the principle of a small,

trusted computing base

consider security software.

This software is obviously part of the

trusted computing base because it used by

the system to enforce security.

In other words, to make sure the system

is free of viruses or worms and the like.

Unfortunately over time this software

has grown in size and complexity and

is now has become vulnerable itself and

is frequently attacked.

Now let's consider another principle

that follows the idea of trusting with

reluctance, and

that is the principle of least privilege.

This principle states that we should

not give a part of the system more

privileges than it absolutely

needs in order to do it's job.

This is similar to the idea of need to

know, not telling someone a secret unless

they really need to know it,

and the motivation is the same.

We want to mitigate any

affects of compromise.

If a component is compromised by

an adversary we want its capabilities to

be limited so that the adversary's

power is also limited.

As an example, consider the idea

of attenuating delegations to

other components in

the design of a system.

For example, a mail program might delegate

to an editor for authoring emails.

This editor could be vi or emacs,

which are both very powerful editors,

and both very desired by users.

Unfortunately, these two editors and

others permit too much power.

For example, they permit escaping to

a command shell to run arbitrary programs.

This is too much privilege for

an email program, because if we

gave an untrusted user access to the email

program they could author an email and

then use the delegated email editor

to execute arbitrary shell commands.

A better design is to use a restricted

editor that only allows the user to

perform the functions that are absolutely

necessary to author emails.

An important lesson here is

that trust is transitive.

If you trust something,

you trust what it trusts.

And, of course,

that trust could be misplaced.

In our previous email client example, the

mailer delegates to an arbitrary editor.

And that editor permits running

arbitrary code by delegating to a shell.

Hence, the mailer permits

running arbitrary code.

This is probably not what we had in mind.

Rule that implements the principle of

least privilege is input validation.

The idea here is to only trust inputs

that we have properly validated.

That is, we don't want to trust any input.

Instead we want to establish

trust in that input by

making sure that it

follows an assumed format.

In short, we are trusting a subsystem

only under certain circumstances.

And we want to validate that

those circumstances hold.

So we have seen several

examples of this so far.

We can trust a given function only if the

range of its parameters is limited, for

example to be within

the length of a buffer.

Or we could trust input taken from a

client form field on a webpage only if it

contains no script tags and therefore

is not subject to cross-site scripting.

We could trust a YAML-encoded string,

but only if it contains no code.

So by validating the input, we are

limiting the influence of that input, and

thereby reducing the privilege

that we're replacing in

the components that process it.

Another secure design principle

that aims to trust with

reluctance is promote privacy.

Broadly speaking, a manifestation of this

principle is to restrict the flow of

sensitive information throughout

the system as much as possible.

Basically the idea is that only

those components that have access to

sensitive information can

possibly disclose it.

And therefore, by limiting access,

we reduce the trust in

the overall system because we don't

have to trust as many components.

As a result, we are mitigating

the potential downside of

an attack because fewer components that

could be compromised will be able to

disclose that sensitive information.

As one example of this principle in

action, consider a student admission

system that receives sensitive

letters of recommendation for

those students as PDF

files from recommenders.

A typical design would allow

reviewers of student applications to

download recommendation files for

viewing onto their local computers.

The problem with this design

is that compromise of

the local computer will then leak private

information contained in the PDF files.

A better design is to then permit

viewing of PDF only in the browser,

as opposed to downloading

it to the local computer.

As a result,

if that local computer is compromised,

no sensitive data is available for

access by the attacker.

The last design principle that we'll

consider that has the flavor of trust with

reluctance is compartmentalization.

This design principal suggests that we

should isolate system components in

a compartment or a sandbox.

And thereby reduce the privilege

of those components by

making certain interactions or

operations impossible.

In this way we can prevent bad things,

that is,

those only enabled by the operations

that we're not allowing.

And we can mitigate the results of an

attack by preventing the compromised code

from doing anything beyond

what the sandbox would allow.

As an example, we might disconnect

a student records database from

the internet and

grant access only to local operators.

This compartmentalizes

the student records database,

therefore preventing

attacks from the outside.

Just as importantly if a local operator

attempted to compromise the database and

exfiltrate the data it would not be

able to do that via the Internet.

Another example of a mechanism for

compartmentalization is

the Seccomp system call in Linux.

This system call enables us to build

compartments for untrusted code.

SecComp was introduced in Linux in 2005.

The way that it works is that

the affected process can

subsequently only perform

a limited set of system calls.

Read, write, exit, and sigreturn.

Notice that there is no support for

the open system call.

Instead, the compartmentalized process can

only use already open file descriptors.

SecComp therefore isolates a process by

limiting its possible interactions to

only these system calls.

Follow-on work produced seccomp-bpf,

which is more general.

In particular, it allows us to limit

a process to a policy-specific set of

system calls,

rather than just the four I just listed.

This policy is enforced by the kernel.

It's specified by something like

Berkeley Packet Filters, or BPF.

SecComp BPF is used by Chrome,

OpenSSH, vsftpd,

which is the very secure ftp daemon,

and others.

So let's look at how we might

isolate Flash player that aims to

execute a file that we

browse to on the internet.

First suppose we browse to a website

that has a shockwave file.

Which is a program that's going

to be run by Flash player.

As usual, we'll save this

file into the local machine.

Next, we'll call fork to

create a new process.

And that process will open the file.

Now this process is still

running as Chrome, and so

we must exec the process to now

run as Flash player instead.

Now, before the process

can get much further,

we call seccomp-bpf to

compartmentalize it.

Limiting the system calls

that it can perform.

In particular, we do not allow

it to open network connections.

And we don't allow it

to write files to disc.

That way, we can ensure that Flash,

if compromised, will be limited

in the damage that it can do.

The secure design principle

of Defence in Depth aims to

increase security through diversity.

The idea is not to rely on a single

defense or even a single kind of defense.

That way if one defense is overcome there

is still another one to get passed.

This principle can mitigate damage from

an attack or can prevent it altogether.

An example application of Defense in Depth

would be to do all of the following.

Use a firewall to prevent access to

all network ports other than that of

the outward facing web server.

Encrypt all sensitive data like employee

records when stored at rest, for

example on the hard drive.

And use a type safe language to write the

web server to prevent various classes of

attack like buffer overruns.

A less diverse defense might

rely on one of these, but

not all of them and then be vulnerable

if the defense was bypassed.

One common application of Defense in

Depth is to the authentication process.

Now, we know that passwords

can be chosen poorly and

possibly guessed by adversaries, thereby

bypassing the intent of authentication.

Passwords can also be stolen.

Now, one way to defend against

a stolen password is to

encrypt the password database

that keeps those passwords.

But to do so

would assume that compromise is possible.

A company may think that all of

their outer defenses, firewalls, and

secure software, and fully patched

systems should keep adversaries out.

But this is probably a bad assumption.

And therefore an extra defense of

encrypting the password database is

a good idea.

Now another sort of Defense in Depth

is the use of community resources.

The idea here is to depend

not only on yourself but

to depend on a diversity of people

toward solving the security problem.

One way to do that is to use hardened

code, perhaps from other projects.

For example, rather than rolling

your own crypto libraries you could

use libraries implemented by others,

and tested and assured over time.

On the other hand, you shouldn't

just trust them out of the box.

You probably want to test that

they will meet your needs and

ensure that you're using them properly.

Another way to use community

resources is to vet designs publicly.

This is typically what happens

with modern cryptosystems.

Instead of trying to keep

the algorithm that is used hidden,

the designers of the algorithm make it

public so others can comment on it and

potentially find mistakes.

Finally, stay up on recent threats and

research.

There's a broad security community

that is tracking the state of

affairs in security today.

So NIST is a good place for standards.

OWASP, CERT, and

Bugtraq have vulnerability reports.

SANS Newsbites has a great collection

of the latest top threats.

And of course, the latest and

greatest research in long term trends and

technology can be found in academic and

industry conferences and journals.

The final design principle in our

list is monitoring and traceability.

This principle is an acknowledgement

that there is no perfect security and

eventually all systems

will succumb to an attack.

As such, we need to design systems

that allow us firstly to discover when

an attack has taken place and secondly to

figure out how the attack succeeded so

that we can try to stop it the next time.

A way to satisfy both aims is to have

the system log relevant information during

it's operation.

Such logs will contain things

like transactions processed,

failed log in attempts, and

other unexpected events.

The logs produced by a system can

then be aggregated with other logs of

systems on an enterprise's network.

And together when considered, all of

these can aid the diagnosis of attacks.

Software like splunk can be used for

aggregation.

A key element of a security conscious

software development process is

performing code reviews with

the assistance of tools.

Increasingly such tools employ

a technology called static analysis.

To understand the basics we will

develop a flow analysis that tries to

understand how tainted values

flow around a program.

Our goal is to find bugs.

Then, we will look at how to increase

the precision of the basic analysis by

making it context,

flow and path sensitive.

Current practice for software

assurance tends to involve testing.

The idea here is to make sure that

the program runs correctly on a set of

inputs that seem relevant to

the operation of the program.

So, we provide those

inputs to the program.

The program produces some outputs.

And an oracle that the tester

constructs confirms that

the outputs are the ones

that are expected.

So testing is great because,

if a test fails,

you have a concrete failure that you

can use to help find what the issue is.

And, really also to establish whether or

not the failure is a true one.

On the other hand, coming up with

test cases actually running them.

And doing them so that they cover all

of the relevant code paths is very

difficult and provides no guarantees.

The no guarantee part is especially

troubling for security tests.

Because it only takes one failure,

one failed code path to lead to

a vulnerability that could

take down an entire system.

Another approach that's complimentary

is to use manual code auditing and

the idea here is to convince someone else

on your software development team that

the source code is correct.

Now, the benefit is that humans,

when they review code,

can generalize beyond single runs.

So they can do better than

just single test cases.

They can imagine executions in their

head that maybe they haven't written

test cases for.

On the other hand,

human time is very expensive

especially compared to computer time.

And once again, this is a difficult task

that provides now guarantees of coverage.

And of course,

we are worried about security, and so

the malicious adversary is going to

exploit those things that we miss.

So, static analysis to the rescue.

The idea here is to use a computer program

to analyze a program's source code

without running it.

So in a sense we're asking a computer

to do what a human might have

done during a code review.

And the benefit here is

much higher code coverage.

The computer program, the static analyzer,

can reason about many possible runs of

the program, and in some cases even

reason about all possible runs, and

thereby provide a guarantee

that the property that

we've attempted to establish,

hopefully a reasonable.

static analysis,

surprise, surprise, has drawbacks too.

For example, it can only tend to

analyze limited properties rather than

complete functional correctness.

It may miss some errors or

have false alarms for engineering reasons.

And it may also be time consuming to run.

Static analysis can have significant

impact on a security oriented

development process.

Because static analysis can throughly

check limited but useful properties and

there by eliminate entire

categories of errors,

it frees up developers to

concentrate on deeper reasoning.

Static analysis tool usage can also

encourage better development practices.

Programmers who use tools start to

develop programming models that

avoid mistakes in the first place.

Programmers also end up thinking about and

making manifest their assumptions.

For example, they will use annotations

that tools understand in order to

improve tool precision by specifying

what they intend for a program to do.

Static analysis tools are becoming

a commercial reality.

And so companies are starting

to feel this impact today.

In particular, there are many

different elements of an analysis that

trade off with one another.

A practical tool must decide which

elements are most important.

Some of these elements are the following.

A precise analysis aims to model

program behavior very closely.

To make the most informed decision

about whether it has found a bug, so

as to avoid false alarms.

A scalable analysis will

successfully analyze large programs,

without unreasonable resource

requirements, so that is space and time.

Often times scalability is

a direct tradeoff with precision.

An understandable analysis takes

its human user into account.

For example, the analysis is designed so

that alarms are easy to understand and

are actionable.

An analysis that enjoys all

three features is easier to

reduce if its focus is clean code.

The intuition is that

code that is hard for

a human to understand, is also hard for

a tool to understand and vice versa.

As such, it may be reasonable to issue

false alarms or run more slowly.

When doing so is because of confusing or

convoluted code patterns.

In this way we can view

such poor performance of

the static analyzer positively.

If programmers clean up their code

they will reduce the total number of

false alarms and

perhaps improve the running time.

While also making their code easier for

humans to understand.

We are now going to look in

depth at one particular kind of

static analysis called flow analysis.

Roughly speaking,

a flow analysis tracks how values might

flow between the different

memory locations in a program.

This kind of analysis is

interesting because it can help us

find bugs whose root cause is

improperly trusting user input.

And in particular,

input that has not been validated.

In a tainted flow analysis,

untrusted input is considered tainted.

Various operations in the program,

on the other hand,

expect to operate only on untainted data.

If the flow analysis finds that tainted

data could be used where untainted data is

expected, there could be

a potential security vulnerability.

You can improve your

analysis by adding sensitivity,

which increases the analyses precision.

these constraints will complain

that there is an error, when in fact,

there is none.

And therefore,

the analysis has produced a false alarm.

So the problem here is that our

analysis is flow insensitive.

Each variable has one qualifier

which abstracts the taintedness of

all values that it will ever contain.

On the other hand, a flow sensitive

analysis would account for

variables whose contents change.

Each assigned use of a variable would

effectively have a different qualifier.

We could even implement such

a flow sensitive analysis by

transforming the program to assign

to a variable at most once.

And this is converting

the program to what is

called static single assignment form or

SSA.A path sensitive analysis

will rule out such a path

by qualifying each constraint

with a path condition.

So instead of just generating

the constraint, untainted is less than or

equal to alpha, it will instead include

the condition that says this constraint is

only valid when x is not equal to 0.

On the other hand,

the constraint tainted is less than or

equal to alpha is only

valid when x equals 0.

When determining whether

constraints have a solution,

the path conditions will be considered.

If flow and

path sensitivity are so useful,

is there any reason not to use them?

It would seem that the added precision

is important for avoiding false alarms,

which would otherwise arise

by conflating flows or paths.

But both sensitivities make the flow

analysis problem harder to solve.

For example,

flow sensitivity increases the number of

variables the analysis considers, and

thus increases the number of nodes in

the constraint graph that we must solve.

Path sensitivity requires the use of

predicates to distinguish different paths,

which means we must

employ more general and

expensive solving procedures

to make use of them.

In short, the added precision decreases

performance, which ultimately hurts

scalability, limiting the size

of programs we can analyze.

This trade-off is sometimes managed by

employing a demand driven analysis,

which adds precision only as needed and

where it's needed.

We focused on tracking tainted flows

through blocks of code, normal statements.

Now we'll consider how we handle

tainted flows to function calls.

we need to give flow

variable names to the argument and

return value of the function.

So this is a problem of

context insensitivity.

These call sites, as I said,

are conflated in the graph.

So context sensitive analysis solves this

problem by distinguishing

call sites in some way, so

that we don't allow a call at one place

to return it's value to another call.

We can do this by giving call

sites a label, call it i and

perhaps correlate it with the line number

in the program at which the call occurs.

And then in the graph, matching up

calls with corresponding returns,

only when the labels on flow edges match.

We'll also add what we call

polarities to these edges,

to distinguish calls from returns.

So, minus i for argument passing,

and plus i for return values.

So let's see the example again, but

this time in a context sensitive setting.

Just like flow and path sensitivity,

context sensitivity is a tradeoff,

favoring precision over scalability.

In particular, the context insensitive

algorithm takes roughly time O of n,

where n is the size of the program,

while the context sensitive algorithm

will take time O of n cubed.

On the other hand, sometimes the added

precision actually helps performance.

By eliminating infeasible paths

it can reduce the size of

the constraint graph by a constant factor.

And this is true for flow and

path sensitivity too.

But the general trend is that greater

precision means lower scalability.

One way to push the trade-off

more towards performance.

Is to be selective about where

context sensitivity is used.

Rather than using it at all call sites,

we could use it only at some of them.

The remaining call sites are conflated,

as in an insensitive analysis.

This means that the analysis

coarsely models calls from one site,

as if they could return to another.

Rather than having one

blob of insensitive sites,

we could also imagine groups of sites,

that are sensitive with respect to

the other groups, but insensitive

with respect to their own members.

Doing this amounts to giving all sites,

in the same group, the same index.

Finally, it's also possible to

limit the depth of sensitivity.

For example, to only match nested

calls up to a certain limit.

let's finish up this discussion of our

tainted flow analysis by seeing how to

scale it up to a more complete

language and problem set.

So far we've considered scalar values or

pointers treated as blocks.

But we haven't considered

dereferences through pointers and

how those affect the tainting.

Well, here's a problem.

A solution exists,

even though the assignment to q of p

is going to affect the contents of p.

And therefore, this is actually an error

that our analysis will fail to uncover.

So what do we need to do to fix this.

Well, we need a way of

identifying when flow can

happen through assignments via aliases.

And we do this by, when we assign

pointers to pointers, we create edges

that go both ways between the flow

labels of the things that they point to.

Okay, with this extra constraint,

a solution no longer exists.

We can see now that tainted

can flow to untainted, and so

therefore, we've caught the error.

So again, the idea is that an assignment

via a pointer flows both ways.

That is, it anticipates that this,

through an alias,

we could assign a value that

would create a flow later on.

And the flow both ways allows

that assignment to be captured.

Unfortunately, this can

lead to false alarms.

So we have a couple of choices for

reducing them.

One is, if we know that an assignment

through an alias can never happen, for

example because the alias

is labeled as const,

then the backward flow is not needed and

we do not need to include the edge.

Or we could just drop

the backward flow edge anyway,

anticipating that

an assignment will not happen.

That's problematic, whether or

not that's guaranteed by the type system.

So effectively, this is trading

a false alarm for a missed error.

And different analyses make different

choices about this question.

Another sort of hidden flow that might

happen is what's called an implicit flow.

So, we can perform what we

call information flow analysis

to discover when such

implicit flows happen.

Data did not flow, but

the information did.

We can discover this by

maintaining a second label that we

call the program counter label.

And it represents the maximum,

in the lattice,

taint, affecting the current position

of the program counter in the program.

Assignments will generate

constraints involving the pc.

So x equals y will

produce two constraints,

one between the labels of y and

x as usual, and another saying that

the pc label must be able to be

upper bounded by the label of x, so

that a flow does not leak

through the assignment of x.

And this will allow us to

track information flows.

So this is discovering the implicit flow,

that is that the tainted source

is tainting the destination.

So, should we always track

information flow constraints?

Well, tracking them with a pc label

can sometimes lead to false alarms.

If in a program,

the dst value is always going to be 0,

no matter what the value of src is,

and so there is no information leak.

And yet, the assignments to dst will

be tainted according to the use of

the pc label, and

therefore it will produce a false alarm.

Extra constraints due to the pc

label can also hurt performance.

Developers typically don't write

programs like this, and generally,

implicit flows have little overall

influence in well written programs.

So as an engineering question,

oftentimes implicit flows are not

tracked in industrial analyses.

One point though, is that if you're

analyzing untrusted programs, for example,

mobile code that's being downloaded by

your browser or some other system, that

you run locally, there is a greater chance

that an adversary will maliciously exploit

this weakness in your analysis to try to

taint information, or perhaps leak it.

However, this situation aside, tainting

analyses tend to ignore implicit flows.

We have considered how to analyze most

of the key elements of a language.

But not all of them.

A robust tool obviously

has to handle them all.

First, while assignments

obviously transfer the taint from

the source to the target,

what should happen if the source is

an expression rather than a variable?

We must define the taint of operators.

Usually, if an argument to

an operator is tainted,

then the expression containing

that operator is, too.

So, far we've assumed we know

which function a call will invoke.

This is important for

linking up the flow constraints at

the caller with those in the callee.

But what if we make a call

through a function pointer?

The pointer is not assigned until runtime.

So, that static compiled time analysis.

Canno, cannot always

be sure of its target.

As it turns out we can use

a kind of flow analysis to

determine the set of possible targets

a function pointer could contain.

The analysis follows the same

principles of the flow analysis that

we've already seen.

This way, when analyzing a call site using

a function pointer, we add constraints as

if all possible targets were called

rather than a single target.

And even more course analysis

of possible targets, for

example, based on a function's type

could also be used rather than one

that more precisely determines the

possible values of the function pointer.

C structs are records have several fields.

The most precise analysis can track

the taintedness of each field of

a struct separately as if

they were separate variables.

But such precision can be expensive.

Alternatively, an analysis can track

the taintedness of the same field of

all instances of a struct as if that

field was a kind of global variable.

Now this hurts precision but

is often a big boost to performance.

An in between position is to analyze only

the taintedness of the struct as a whole.

Or to do so for only some of its fields.

Note that these decisions are relevant

to object oriented languages.

You can view objects as a struct

containing function pointers and so

the trade offs we've just

considered apply in the analysis of

object oriented languages.

Another way to aggregate data in

a language is to store it in an array, and

there are several,

similar trade-offs here.

You can track taintedness

of each element or

you could track it on

the array as a whole.

The challenge here is that while

struct fields are constants,

array indexes are computed by arithmetic.

As such, it is sometimes hard to

track them precisely at compile time

because we won't know what arithmetic

produces until we run the program.

And so

a hybrid approach is inevitably employed.

Up until now we have focused

on how the analysis works.

A key part of using it to

perform security reviews is to

properly label the tainted sources and

untainted sinks.

And there are many options here.

We could label array indexes and

format strings and SQL strings.

We can also label functions that

act as sanitizers or validators.

These are library routines that take in

tainted data and return untainted data.

For example a function that escapes

an HTML string would be a sanitizer.

Labeling sanitizers is needed

to avoid false alarms.

Finally flow analysis need not only apply

to finding improper use of tainted data.

It can also be used to discover

the improper disclosure of

sensitive information.

In particular we can imagine labeling

sensitive data as secret, and

untrusted output channels as public.

And our goal is to prevent secret

sources from reaching public sinks.

In essence this is the dual of

the tainting problem, where the only

difference is that the lattice and

flow relationships are inverted.

Finally, we should point out that

flow analysis is not the only kind of

static analysis, in,

in fact there are many other kinds.

One common analysis, that is often used

to assist other analysis problems,

is called pointer analysis.

With a common flavor of it

called points to analysis.

This analysis determines whether two

pointers are possibly aliases, that is,

whether they could both

point to the same memory.

Earlier we saw that knowing this

is important for not missing bugs.

In fact, our tainted flow analysis

involved a very coarse grain

form of point to pointer

analysis within it.

But more sophisticated analysis

would improve precision.

Significant advances have

aided both the precision and

scalability of pointer

analysis in recent years.

Another kind of analysis is

called data flow analysis.

And it was developed in the 1970s as part

of research on optimizing compilers.

Like the flow analysis

we've already considered,

data flow analysis focuses on the flow of

values through variables in a program.

But it maintains information

about this flow bit differently.

One common data flow analysis

is called liveness analysis.

And it determines which variables,

at a given program point are still alive.

That is, whether their values

could still be red in the future.

Other variables are dead, meaning

that their values will be overridden.

Liveness analysis is an important

part of allocating variables to

machine registers during copulation.

Registers storing dead variables,

can be re-used.

Data flow analysis techniques

are also useful for

security questions and

many industrial tools use them.

For example, they can also answer

the tainted flow question.

Finally another significant

kind of analysis is called

abstract interpretation.

It was invented in part to provide

a theoretical explanation of

data flow analysis, but it has grown into

a style of analysis in its own right.

The key idea of abstract interpretation,

is that a static analysis is

an abstraction of all possible

concrete runs of a program.

To be practical,

the abstraction must discard information.

For example,

which calls correspond to which returns.

The key is to discard as much

information as possible for

purposes of scalability, while still being

able to prove the property of interest.

An abstract interpretation is at the core

of several industrial strength tools.

Static analysis is gaining traction

in practice with a variety of

commercial products from a variety of

companies as well as open-source tools.

Software has bugs.

Seems unavoidable.

The most common ways of finding

bugs are testing and code reviews.

But it's all users know,

software, nevertheless,

ships with bugs still unfixed.

In fact, many of these bugs

are known to the developers but

they were not deemed important

enough to fix prior to shipping.

Many other bugs are not known.

Why are they not found?

It could be that the bug is

in a rarely used feature.

For example, in a device driver for

a rare piece of hardware.

Or, the bug arises in rare circumstances.

For example,

when memory is nearly exhausted, or

the bug could arise nondeterministically.

The circumstances that make the bug

manifest could be unpredictable.

And thus make the bug hard to

reproduce and hard to fix.

One way we can try to find those bugs that

testing misses is to use static analysis.

And this conceptually makes sense,

because static analysis can consider

all possible runs of the program,

perhaps even those that are rare,

non-deterministic and so on.

We can feel heartened by the explosion

of interesting ideas and

tools that are being

considered in research.

And by the fact that companies are now

selling and using static analysis tools.

So there's great potential here

to improve software equality.

The question, though, is can static

analysis find deep difficult bugs?

So in principle, yes.

But, maybe in practice, not so often.

Okay, why is this?

Well, it's because commercial

viability implies that you have to

deal with developer confusion,

false positives, error management.

That is quite a lot of code that has

nothing to do with the core idea of

analysis itself but, rather,

the way that humans use the analysis.

One particular thing that companies

seem to do is to keep the false

positive rate down.

They do this by purposefully missing

bugs to keep the analysis simpler.

This turns out to be a good idea because

studies show that developers get tired of

dealing with alarms that don't

turn out to be true bugs.

And they're not interested

in sifting through,

say, nine alarms, even if on the tenth

time they find a really important bug.

So companies respond to this, and

they make tools that will miss bugs.

So one of the issues here that makes false

alarms hard to deal with is abstraction.

Abstraction is critical for

static analysis because it allows us to

simplify our view of what the program

does so that we can efficiently, or

efficiently enough,

model all possible runs.

But to do this we have to

introduce conservatism.

Now the various sensitivities like flow,

context, and

path sensitivity add precision.

So that reduces conservatism.

But these more precise abstractions

are more expensive, which means that they

won't be applicable to larger code bases,

at least not at short time scales.

And they still will not completely

eliminate the false alarms problem.

One particular problem that makes

false alarms hard to deal with is

that the static analysis technique

may use an abstraction that

is not familiar to the developer,

that is the developer is not able to

make sense of the alarm that the tool

produces because it was produced in a way

that the developer can't relate to.

And therefore can't easily triage to

determine whether the bug is true or no.

So for the remainder of this unit,

I'm going to talk about a technique called

symbolic execution, that's kind of

a middle ground between testing and

static analysis, and aims to retain

the best features of both techniques.

We start with the observation

that testing works.

Reported bugs are real bugs.

The problem is that each test only

explores one possible execution of

the program.

If we do assert f of 3 is equal to 5,

well, we've checked that f of 3 equals 5,

but not that f applied to other

input values also equals whatever it

is they're supposed to equal.

In short, tests are complete,

but they're not sound.

And, while we hope they generalize,

they don't provide any guarantees.

Now, symbolic execution

aims to generalize testing,

in a sense it's more sound than testing.

so we can write tests like this.

We can say that y is equal to alpha, which

represents an unknown or symbolic value.

And then we can assert that f of

y is equal to 2 times y minus 1.

This is basically saying, for all y,

f of y is equal to 2y minus 1.

Symbolic execution can

execute this program, and

confirm that this is indeed the case, thus

performing a more general sort of test.

The way that this is possible is

that the symbolic executor can

run the program to the point

that it depends on an unknown.

And then it will conceptually fork

the symbolic executor to consider all

possible values of that unknown that would

affect the control flow of the program.

So here we see the function f, and

we see that it depends on x,

it's argument at the first branch.

If x was symbolic,

as in our assert, then the symbolic

executor will consider both branches.

That is the case when x

is greater than 0 and

the case when it is not greater than 0.

And of course, in this example when

it does that it will discover that

the assertion will not hold for

non-positive values of x.

And that is, if we used a satisfiability

solver to give us a potential solution to

this path condition, we could produce

a test case that shows the failure.

And of course, this is very useful

because the developer can then run

the test case and use it to diagnose

the reason for the failure in the program.

Notice that each path through the tree

we just constructed represents a set

of possible executions.

More precisely, it represents

all of those executions whose

concrete inputs are solutions

to the path condition.

This set is a very precise

notion of code coverage.

Viewing symbolic execution

as a kind of testing.

Complete coverage of the program

would be all of its paths.

Symbolic execution allows us to

systematically consider

many of these paths.

Viewed as a kind of static analysis,

symbolic execution is complete in that

whenever a symbolic executor claims to

have found a bug, the claim is true.

However, it is rarely sound because

it is unlikely to cover every path.

The reason is that most programs do

not have a finite number of paths, and

so symbolic execution will not terminate

except by imposing a time limit or

other resource limit.

Symbolic execution's high precision,

the source of its completeness, can be

characterized as path, flow, and context

sensitivity in static analysis parlance.

So if symbolic execution

is such an old idea, and

it seems like such a good one,

well why didn't it take off?

Why haven't we been using it for

a long time?

Well, one reason is that symbolic

execution can be compute-intensive.

In the end, for big programs, we'll

consider many, many possible paths, and

along each or many of those paths,

we'll need to query an SMT solver

to decide which of the paths are feasible,

in which assertions could be false.

And in the end such queries

could involve much of

the program state which

could be very large.

In computer science and mathematical logic, the Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Examples of theories typically used in computer science are the theory of real numbers, the theory of integers, and the theories of various data structures such as lists, arrays, bit vectors and so on. SMT can be thought of as a form of the constraint satisfaction problem and thus a certain formalized approach to constraint programming.

The time to run has

been decreasing as the algorithms have

been improving over time.

And though small instances

have not improved so

much in time, since 2008 big problems

are now starting to be solved faster.

Symbolic execution has

experienced a resurgence of

interest in the last ten years or

so, starting in about 2005.

The main motivation behind it was to

show that symbolic execution could

find very interesting bugs that were

often missed by normal testing.

Now we'll look in greater detail

at how symbolic execution works.

We'll start with a typical

programming language.

Here we show a grammar of expressions,

denoted by the letter e.

Expressions consist of things

like integers n, variables x.

Expressions involving arithmetic

operators like addition.

And expressions involving comparisons,

like disequality.

To support symbolic execution, expressions

also include symbolic variables.

Which we denote with greek letters,

like Alpha.

Symbolic variables represent

inputs to the program.

These inputs could come from many sources,

including reads from files or

network sockets,

memory mapped files or devices.

Whereas normal tests must

provide specific inputs.

Symbolic execution models inputs using

symbolic variables, and thus can explore

how different executions are induced

by different values of these variables.

These values are discovered by solving for

constraints generated during execution

as we'll see in more detail shortly.

Now that we've defined

our symbolic language we need to

define its semantics.

One way to do that is to make or

modify a language interpreter

that can compute symbolically.

So normally a program's variables

contain values, but for

our symbolic interpreter they will also

be able to contain symbolic expressions.

Which are program expressions that

make mention of symbolic variables.

Now given a path condition, we can come up

with a solution to the constraints in it,

and these can be used as inputs

to a concrete test case that

will execute exactly that path.

Now assertions, like array bounds checks,

are conditionals.

So if we look at our original program,

we can think of just prior to the index

of the array a performing two

checks to see whether the index expression

z is within the bounds of the array.

That is, is z less than 0, and

is it greater than or equal to 4?

If the answer is yes, we've gone out of

bounds and we should abort the program.

So we can think of our symbolic executor

as inserting these extra statements, and

then performing its normal process

of generating a path condition.

Whereas a normal execution must take

either the true or false branch of

a conditional, a symbolic execution

could actually take both branches.

This is possible because when a symbolic

variable, or variables, appear

in a conditional scarred expression, there

could exist some solutions that would

make the guard true, and

some solutions that would make it false.

As such, a symbolic execute, a symbolic

executor could choose to go either,

or both ways.

How should it choose which and

in what order?

One point to make is that the executor

doesn't immediately know whether the true

or false branch or both is feasible,

when it reaches a conditional.

To know for sure it could invoke

the solver to check the satisfiability of

the path condition.

If a path condition concatenated with

the guard condition is feasible,

then the true branch can be explored.

Likewise if the negation of the guard

is feasible then the false branch can

be explored.

However making calls to the solver,

at each conditional can be expensive.

So, one idea is to just pick on branch and

go that way,

delaying the feasibility check, at

the risk of doing extra, unnecessary work.

A related approach is to use concolic

execution to determine the path to take.

The idea here is to run the program

concretely for actual inputs.

But consider some of those inputs as

if they were symbolic on the side.

The concrete inputs which

guarantee feasibility,

determine which direction to go.

The symbolic executor maintains symbolic

versions of these expressions and

the path conditional is usual.

Which it then uses to

generate subsequent tests.

That is the current path condition

when concatenated with the guard that

contains symbolic variables.

When that's feasible, then we have a

legitimate task that we can execute and so

we should add it to our list.

We'll add the alternative, that is,

going down the false branch in

the case that not p is feasible when

concatenated with the path condition.

And notice here that we could add

either or both tasks to the work list.

So, we can keep on running until

either the list becomes empty,

in which case we've considered

every path in the program.

Or eventually, we get tired and terminate

the program, say after some time out.

Now, in a practical implementation,

we have to consider not just

the main program we're executing,

but also libraries and native code.

At some point, the symbolic executor will

reach the edges of the application, and

the interpreter won't be able

to make further progress.

So this may happen with a library, or

the system calls, or assembly code, or

something like that.

So one thing we could do is,

we could actually pull that code in

two and symbolically execute it.

So for example,

if we're symbolically executing c code,

we could symbolically execute the c code

that implements the standard libraries.

But real implementations of the standard

libraries are often very complicated and

they might involve assembly code and

stuff like that.

And so a symbolic executor will often

get stuck running around loops and

not making any progress

getting out of that library.

So you could use a simpler version

of the library instead that's still

semantically accurate.

Or you could create a model of

the code that you want to execute.

For example, you could create a model

of a RAM disc to model the kernel

file system code, rather than trying to

say symbolically execute the Linux kernel.

So let's return to the idea of

concolic execution I mentioned before,

this is also called dynamic

symbolic execution sometimes.

And the idea here is, instead of

running the program according to

the algorithm that we just saw a moment

ago, we run the program concretely, but

we instrument it to sort of do

symbolic execution on the side.

That is, the instrumentation maintains

some shadow concrete program state

with symbolic variables.

The initial concrete state is going

to determine the path that we take so

it could be randomly generated.

But we will keep track of the choices

we made when guards involve our

shadow symbolic variables.

So we can build up a path

condition along the side.

So, a concolic executor will explore one

complete path at time, start to finish.

And once it's done so it can determine

the next path to explore by simply

negating some element of

the last path condition.

And then solving for that path condition

to produce concrete inputs for

the next test.

While we're symbolically executing

using concolic execution,

we'll always have the concrete

underlying values to drive the path.

But these turn out to be

useful in other ways too.

That is, the process of concretization

is very easy in a concolic execution.

And what I mean by concretization is

replacing symbolic variables with

concrete values that

satisfy the path condition.

Basically we're going to

drop symbolic-ness and

potentially miss some paths.

But we're going to simplify our

symbolic constraints by doing so.

So this allows us, for

example, to do system calls that

might involve symbolic variables.

We just pick reasonable

concrete values for

those variables and do the system calls.

In so doing, we lose the symbolicness but

we're able to continue to make

progress in trying to find bugs.

Replacing symbolic values

can also be useful when so,

SMT solver calls might be too complicated

if we left all of the values in.

a good symbolic

executor requires careful engineering.

We've seen that symbolic execution employs

an appealingly simple algorithm, but

with high computational costs.

In particular, the executor might traverse

many different paths in the program, and

it must make potentially

expensive calls to an SMT solver,

to determine which paths are feasible.

Considering the problem

of traversing may paths,

we will see that symbolic execution

boils down to a kind of search problem.

The goal is to search through

a large space of possibilities to

find events of interest.

For us these events are bugs.

We'll also take a minute to look

at how improvements in SMT solver

performance make the path feasibility

check possible even for larger programs.

A well known problem with symbolic

execution is the path explosion problem.

Essentially, for a symbolic executor to

consider the entirety program's space of

executions it needs to

consider every path.

But because most programs

have a huge number of

paths we can't usually run

symbolic execution to exhaustion.

In fact, programs can be

exponential in branching structure.

Now, if we compare symbolic

execution to static analysis,

we can see that there's a clear

benefit of static analysis.

That is, it will actually terminate

even when considering all possible runs.

And, it does this by approximation and

abstraction, approximating multiple loop,

loop executions or

branch conditions, and so on.

But as we discussed to

motivate symbolic execution,

static analysis as use of extraction

can lead to false alarms.

So what can we do to improve the way

that symbolic execution operates to

maximize it's benefits?

Well to understand that we have to look

at how symbolic execution is viewed as

a search algorithm.

So the simplest way to perform

symbolic execution is to

use either depth-first search,

or breadth-first search.

That is, recalling our algorithm for

symbolic execution, we can use, for

depth-first search,

a worklist that is a stack.

So it will consider the most recent node,

that is the most recent program state,

when deciding what to do next.

Or we can make it a queue,

where we prioritize things that

we put in that queue earlier.

Now the potential drawbacks of using

either one of these two strategies,

is that neither of them are really

guided by any higher level knowledge.

That is, we aren't telling

how we might be looking for

particular bugs or that we want to get

to a particular part of the program.

Instead it will just blindly follow

the program structure in considering what

paths that it wants to execute.

DFS in particular could easily

get stuck in one part of

the program by continuing to go deeper and

deeper.

So think about that loop example.

We could go around the loop over and

over and over again where as in

breadth-first search, we'll consider

both one further loop execution and

the case that we get out of the loop.

Therefore, of these two, breadth-first

search is probably a better choice.

On the other hand,

it is more intrusive to implement.

For example,

it can't easily be made concolic.

So what better search

strategies might we perform?

Well, one thing we can do is to

try to prioritize our search,

to steer it towards paths more likely

to contain assertion failures,

sInce that's ultimately

what we're trying to find.

And recall that assertion failures

could involve array bounds checks,

null pointers checks, and so on.

We only want to run for

a certain amount of time, and

so, the prioritization ensures that,

that time is used best.

To consider these different

search strategies,

think of program execution

as a kind of DAG.

So the nodes in the graph

are programme states and

an edge between those nodes means that

one state can transition to the next.

Therefore, we can think of symbolic search

as a kind of graph exploration algorithm,

trying to pick among the various possible

paths in the overall search space.

One useful technique to

employ is randomness.

We don't know a priori

which paths to take, so

adding some randomness

is probably a good idea.

This is something that

modern stat solvers do and

it works very effectively in that setting.

So, one idea is to pick the next path

to explore uniformly at random, and

we can call this the Random Path Strategy.

Another thing we can do is

randomly restart the search if

we haven't had anything

interesting in a while.

So imagine that we explore depth first for

a while, but as we go further and

further down in the search of a particular

path, we may increase our chances of

stopping and going back to the start

to try a different path altogether.

And we could choose among equal

priority paths at random.

That is, if we had a prioritization

strategy that say, favored coverage, or

we wanted to try paths that we haven't

covered those could statements before.

If we had equal priority paths,

we can just flip a coin to

pick one versus the other.

So one drawback of randomness

is reproducibility.

That is, because we may have made random

choices to figure out a particular bug,

if we run the symbolic executor on

the same program two times, it's not

guaranteed to find bugs that it found

before, and in fact might find new bugs.

This doesn't necessarily sound so

bad except it's complicated for

software engineering, for

example once we fix the bug that we find,

how can we run the symbolic executor again

to confirm that it's not there anymore.

So it's probably good to use

pseudo-randomness based on a seed, and

then record which seed is

picked when bugs are found.

And that way you can confirm that

you don't hit the bug again, or

if you re-run you'll find

the same bugs you found before.

Now a moment ago I hinted at

the idea of using coverage to

guide your prioritization strategy.

And here we'll consider it in a bit

more detail, that is, we should try to

visit statements we haven't seen before

when choosing which paths to follow.

So the approach is to

score a statement based on

the number of times it's been traversed,

and

then pick the next statement to explore

that has the lowest current score.

So this might work because errors

are often in hard to reach part of

the program, and

the strategy will favor going to parts

of the program it hasn't seen before.

On the other hand, we may never

be able to get to a statement if

a proper precondition is not setup,

and so just favoring going closer to

that statement is not necessarily

going to get us there.

Another strategy is called

generational search.

You can think of it as a kind of hybrid

between breadth-first search and

a coverage-guided search.

It works like this.

At generation 0 we pick a program and

a random input to it and

run it to completion.

At the end we take paths

from generation 0,

negate one branch condition in

a path to yield a new path prefix.

Find a solution for that prefix and

then take the resulting path.

Will semirandomly assign to any

variables not constrained by the prefix.

For generation n we'll similarly

apply this approach, but

we'll branch off of generation n minus 1.

And we'll use a coverage heuristic to pick

the priority amongst the different

paths that we choose.

This generational search is often

used with concolic execution.

Now, it's probably obvious at this point,

no one search strategy wins all the time.

Instead different search strategies

may find some bugs that other

search strategies will not find.

So one obvious idea is to combine

searches by doing multiple searches at

the same time.

Essentially have parallel processes

where we switch from one to the next.

Depending on the conditions

needed to exhibit the bug,

one search may find it as opposed

to another search finding it.

You could even imagine using different

algorithms to reach different parts of

the program, depending on the properties

of the program that you're trying to

find bugs in.

SMT stands for Sat Modulo Theories,

where by theories,

we mean mathematical theories

beyond just boolean formulas.

Such theories could include a theory of

arithmetic over integers, for example.

These theories can sometimes

be encoded as SAT problems.

And so

some SMT solvers are basically just front

ends that translate the SMT

problem into a SAT problem.

For example, the theory of bit vectors.

That is, arrays of bits with operations

on them, like addition, subtraction and

so on, can be encoded as a SAT problem.

It turns out that modern SMT

solvers implement a variety of

optimizations either in the SAT engine or

in the translation to it.

And these optimizations can make

a big difference in performance.

One example is to make use of

axioms that witness equivalences.

Another example is to directly

support in the SMT solver a theory of

arrays which model

modifications to memory.

The alternative would to be to have

the symbolic executor simulate memory

directly, rather than rely on the solver.

But then,

optimizations become less apparent.

Another important optimization

is to cache solver queries.

It turns out that symbolic executors

will submit queries repeatedly that

contain identical subexpressions, and this

makes sense because the path condition

is built up incrementally over time.

Finally, we can make the SAT solver's

job easier by eliminating some

symbolic variables from consideration.

For example, if we were testing

whether a guard is feasible,

we only care about the existence of

a solution to the variables in the guard.

Such a solution may involve other

variables which can be syntactically

related to the ones in the guard.

That is, we keep under

consideration expressions that

have variables that are transitively

related to variables in the guard.

For other expressions these

variables can be ignored.

In total the optimizations can

make an enormous difference.

SMT solvers have been a popular

topic of research and

development over the last several years.

There's several that you

can get free online.

Z3 is quite sophisticated,

it's developed at Microsoft research.

Yices, developed at SRI,

has been around for

quite a while and

continues a steady improvement.

STP, which is the SMT solver used by

XC and KLEE is available for free.

And CVC is yet another.

While smarter search strategies and

SMT based optimizations will help,

ultimately path based search is limited.

To see why, consider this program.

We can see that it has a loop.

It's going to run around

the loop 100 times.

And each time it goes around the loop it

may either enter the if statement or not.

This means it has 2 to the 100

possible execution paths.

Now the program is asserting that it's

a bug if the counter is ever equal to 75.

So this bug is going to be hard to find.

That is, there are 2 to the 78 paths that

reach the buggy line of code, that is,

100 choose 75.

Out of the 100 executions,

75 of them need to take the true branch.

But if that's true, the probability

of finding the bug is very low,

because 2 to the 78 paths will

find it out of 2 to the 100 total.

So the chances are 2 to the negative

22 that we will find the bug.

In short, independent of search strategy,

we'll have

a very difficult time finding the bug

using a path-based search algorithm.

So this is really just a fundamental

limitation of path-based search in

symbolic execution.

In the mid 2000s, two key systems

triggered a revival of symbolic execution.

And these systems were DART,

developed by Godefroid and

Sen and first published in 2005,

and EXE, by Cadar,

Ganesh, Pawlowski, Dill, and

Engler, published in 2006.

These systems demonstrated the promise

of symbolic execution by showing it

could find real bugs in interesting and

complicated systems.

And as such it spurred

interested in the topic and

many new systems have developed since.

One important system that has made

big impact in practice is SAGE.

It is an concolic executor,

developed at Microsoft Research, and

grew out of Godefroid's work on DART.

It uses the technique of generational

search that we talked about before.

SAGE primarily targets

bugs in file parsers, for

example parsers like jpg, Microsoft Word,

Microsoft PowerPoint and other documents.

And is a good fit for concolic execution

because parsing files is likely to

terminate and it only needs to consider

input/output behavior rather than

complicated and interactive system calls.

SAGE has transitioned from research

prototype to production tool.

It's been used on production

software at Microsoft since 2007.

Between 2007 and 2013, SAGE was

used remarkably often at Microsoft.

For example, it's been run for

more than 500 machine years as part of

the largest fuzzing lab in the world.

It's generated 3.4

billion plus constraints,

making it the largest

SMT solver usage ever.

It's been applied to hundreds

of applications, and

found hundreds of bugs that were missed

by other bug finding techniques.

For example, one third of all

Win7 WEX bugs were found by SAGE.

Bug fixes shipped quietly to one

billion PC's around the world,

saving millions of dollars both for

Microsoft and the world.

And SAGE is now used daily

in Windows Office and

all of the big Microsoft products.

KLEE is another mature tool.

In this case it grew

out of the work on EXE.

It symbolically executes LLVM bitcode.

LLVM is a compiler framework that's

now in regular use at Apple.

It compiles source language programs to

an intermediate representation called

LLVM bitcode which is

stored in a .bc file.

And KLEE will run on the bc file.

It works in the style of the basic

symbolic executor that we saw before.

Where it uses fork to

manage multiple states.

That is,

rather than having an explicit work list,

each time that it could go both directions

on a branch it actually forks another

version of itself which

proceeds along that branch.

And a separate tool manages

all of the distinct copies of

KLEE that are running at once.

KLEE employs a variety of search

strategies, primarily random path and

coverage guided.

And it mocks up the environment

to deal with system calls and

file access and so on.

And you get it as part of

the LLVM distribution.

So looking at how KLEE actually works,

in the original KLEE paper,

published in 2008.

They applied KLEE to Coreutils,

which is the suite of small programs that

runs on Linux and Unix distributions.

So these are programs like mkdir and

paste and sed and ls and things like that.

What this graph is showing is

the amount of coverage from KLEE

generated tests compared to the manual

test suite that comes with Coreutils.

What this chart shows is

that KLEE is better at

testing than the manual test suite.

And we make that determination by looking

at the lines of code covered by tests.

In particular the lines covered by

KLEE tests when from them when we

subtract the lines covered by manual

tests we see that for all but

nine programs KLEE covers more

lines than the manual case.

So since these second

generation tools of KLEE and

SAGE, there have been

further developments.

One of them is the Mayhem system developed

at Carnegie Mellon by Dave Brumley and,

and his collaborators.

And it runs on actual executables.

That is binaries.

Like KLEE it uses a breadth first

search style search, but it

also uses native execution, combining the

best of symbolic and concolic strategies.

And as an interesting twist,

it will automatically generate

exploits when bugs are found,

to shine a light on the fact that these

bugs could, in fact be security critical.

In further developments

from the Mayhem work,

the same group produced Mergepoint,

which uses a technique called veritesting,

and in this case it combines symbolic

execution with static analysis.

What it does is use static analysis for

complete code blocks.

So it will analyze a straight

line piece of code and in so

doing basically convert it

into a formula that can

be used by an SMT solver to ask

a question about the programs execution.

It will use symbolic execution for

hard to analyze parts of the program.

In particular, loops which, for

which it may be hard to determine

how many times it will run.

As well as, complex pointer arithmetic,

and system calls, and so on.

So this produces a better balance

of time between the solver and

the executor resulting in better bug

finding, that is bugs are found faster and

more of the programs are covered

in the same period of time.

Incredibly, Mergepoint found more

than 11,000 bugs in 4,300 distinct

applications in a Linux distribution

including new bugs in highly tested code,

like the Coreutils from KLEE.

So there are many other

symbolic executors.

Cloud9 is basically a parallel KLEE.

jCUTE and Java PathFinder

are symbolic execution for Java.

Bitblaze is a binary analysis framework,

so

like Mayhem it works

on binary executables.

And Otter is a symbolic execution

framework for C, that is directing and

that you can, tell it which line at

the program you'd like to execute and

it would try hard to get to those lines,

when performing a search.

Pex is a symbolic executioner for

.NET programs.

symbolic execution.

A powerful technology that can be used

to find security critical bugs in

real software.

Symbolic execution can be viewed, on the

one hand, as a generalization of testing.

It uses static analysis

to develop new tests that

explore different program paths.

We can also view it as

a kind of static analysis.

But one with very high precision and

no guarantee of termination.

And if it does terminate it's

results are both sound and complete.

Symbolic execution is used in practice

today to find important bugs in

production code.

The SAGE tool is used

heavily at Microsoft.

And the Mergepoint tool regularly

analyzes large Linux repositories.

Symbolic execution tools of good

quality are freely available.

And we can expect that they,

like static analysis tools,

will penetrate more deeply into

the main stream in the near future.

Coverity,

a leading provider of software quality and

security testing solutions.

Among these methods is

static code analysis and

Coverity, their code advisor tool,

is an industry leader in static analysis.

Well static analysis is

really just a broad name for

a suite of different technologies that

analyzes programs without executing them.

And that can encompass many different

kinds of things including you know,

things that your compiler typically would

do if you're using a statically typed

language to check the types

are used properly.

But it can also include very

advanced techniques that such as

data flow analysis,

control flow analysis that go

beyond what the compiler would

typically do you know to find defects.

And so it's a very broad rubric, and

underneath it you can take a look at many

different specific technologies all around

that's actually helping them find more

about sort of many, many different kinds.

Inside Code Advisor we use

dataflow analysis to find how data

flows through the application and

potentially find tainted dataflows, for

example, where potentially untrusted data

is used in a, in a, in an unsafe fashion.

We also use, other forms of analysis to

detect control flows where operations are,

are used incorrectly.

So for example, you might have

memory leaks in a program which

allow the program to essentially lose

track of memory that's been allocated.

And you know many many other

kinds of defect types as well.

So there's, there's essentially

many different kinds of

analyzers that fall under

the rubric of static analysis.

So you gave us some nice examples

of what code advisor looks for.

Tainted data flows is something that

we talk about in the course a lot, that

is that many security issues arise because

data that should not be trusted that comes

say, from the user over the Internet,

is trusted to conform to a certain format.

And then is used to overrun a buffer or

used as a format string or

something like that.

So I can imagine that static analysis and

identifying so-called tainted or

untrusted data flows should really useful.

And that leaves me to wondering how,

how general is the technology that

the tools that you developed use?

In other words is there sort of a core

technology that many different security

issues take advantage of, or

are each of the analyzers you spoke

of really sort of independent?

I would say that there's

a handful of core engines, right?

In, in our static analysis

product in particular.

One of them is a, a global intraprocedural

data flow analysis engine, right,

that can track data across procedure

calls and you know, through the heap and

things like that to some extent.

And that's very useful for

problems like SQL injection scripting.

These typical tainted data flow problems.

There's also a separate analyzer that

essentially analyzes the program bottom up

by summarizing function behaviors, and

then using those summaries when

analyzing callers of those functions.

And that, that's also intra-procedural,

but it's not producing it, a global

dataflow graph, and that's very useful for

detecting control flow type problems.

And usually we would consider you know,

certain kinds of, of issues such as

certain kinds of buffer overflows

fall into that, under that category.

So there's a few core engines that

are very commonly used across all of

our different checks.

And then there's some odd balls like,

things like dead code for

example; use a somewhat different,

analyzer engine.

Because we found that detecting dead

code is a little different from

the other properties we're looking for.

Another exception is copy and paste that

uses a different detector than others,

because copy and pasters work differently

than, than typically floaters.

in the the dead code elimination

one, or dead code detection,

is that something that you might run

before running the other analysis so

they end up looking at less code?

That is the question

sometimes we get asked if you

know that helps make

the analysis more efficient.

But we found that the, it's not really

useful for that purpose because it's

very hard to tell what code is not

being executed from our perspective.

From a purely static point of view any

function that is potentially exported

is potentially culled because this might

be a library that we're analyzing.

It's very difficult to tell if it's, you

know, something we can truly just skip.

So we don't really skip anything

when running our other analyzers.

What do you think I can imagine

that static analysis would

be useful because it's,

it's analyzing the code before it runs.

How does it fit into the pantheon

of things that you might do.

To assert that, to give yourself

satisfaction that your program is

secure compared to things like manual

code reviews, or penetration tests.

How does static analysis fit in,

how does it compare to those things?

there is no panacea for security.

There's no single technique or

approach that will solve all of

your security problems, and

that is also true of static analysis.

I would, I would,

I would characterize it as,

sort of part of a complete breakfast,

if you will, right.

So you have many different techniques.

yes, manual review is definitely

a technique that should and, and

can, and does get used in practice.

But it's, the problem with manual review

is that it's very inefficient, right?

It's very costly to review every

single line of code with an expert who

actually understand all of the different

potential security vulnerabilities.

Really there's very few people

in the world that know about any

particular type of vulnerability in depth.

So it's very hard to get

the level of review and

the coverage that you need

through manual analysis.

Dynamic analysis is another approach

that is very commonly used.

And this, this can mean different things.

One of the most common meanings of

it today is, for web applications to

have a sort of scanner that will probe

an application by crawling through

the different pages your application has

and essentially trying to attack it.

with, with malicious inputs to

see how the application responds.

When it works that's fantastic because

then you get an actual example of

an attack and as a developer you have

verifiable proof that, you know,

you have a vulnerability, right?

And that's nice but the,

the that's the plus side.

The downside of it is you don't

get very good coverage of

your application typically.

So it's hard for dynamic analysis

to reach all the nooks and

crannies of the behaviors

of your application.

Static analysis you know, the strengths

are that it covers all of the application.

Right it can analyze every piece of code.

Every potential corner case.

The, the, the downside if you will

is that because it's trying to

analyze a program in abstract.

It can have what's known

as false positives, right?

These are reports that,

it thinks are real defects or

vulnerabilities but

aren't really a problem in practice and

it's up to developers to

determine if they're real.

And depending on the analyzer

you use it can take time and

memory to analyze a large program.

And so scalability for very large

programs can be a concern as well.

So imagine that, that scalability also

trades off and maybe the false positives

issue as well with the precision of

the analysis that is to say sort of the,

the, how fine grain the properties

are that it tries to figure out.

I think, that's one of the central

challenges of building static analysis is

how to make it scale up to realistically

large programs, at the same

time having a reasonable false positive

rate so developers don't lose interest.

And also not missing things, right?

There's false negatives as well,

you might miss defects,

and yeah, in real life static analyzers

don't find everything in practice.

And that's, like, a deliberate

trade off in order to be able to

have these other properties

that we also find desirable.

So it's a balancing game and

I think from a commercial standpoint that

balancing game is where we spend a lot of

our time trying to optimise for, you know,

lowering false positive rates while

at the same time finding more bugs.

And also scaling up to large programs.

These are conflicting factors.

So you just mentioned as

a commercial company you

obviously have commercial clients.

But my understanding is you have

a program called Coverity Scan that also

provides a service to

the open source community.

Could you also tell us

a little bit about that?

Absolutely one of the things

we found is that a lot of

commercial software is not

fully commercial software.

They use a lot of open source it,

some statistics show that

even very large commercial projects

might contain 60 to 80% or

more of open source

software inside of them.

If you look at this pure

line of code count.

So we think of it,

what we did was we developed a scanning

project for open source projects.

It was originally started with

the Department of Homeland Security.

But ultimately we decided

to fund it ourselves.

And what it does is allows open source

projects to submit their code to us on

a regular basis, get the analysis results

in the cloud, and fix them for free.

So it's great marketing for us.

But it's also, I think, a good service for

the open source community.

They've fixed tens of

thousands of defects.

Some quick statistics on scan you know,

we just crossed 500 million lines of

code in scan that's being

analyzed on a weekly basis.

And over 2,500 projects across

many different languages.

C, C++, and Java.

Right now so,

it's a very successful project, and

if you have some open source

code that you want scanned for

free, you can certainly submit it to, to

ScanThat converter common and try it out.

Let's see, I'm whether,

you mentioned a moment ago,

doing your analysis on C and C++ in Java.

Is it, is it the case that static

analysis is more difficult on some of

those languages as compared to other ones?

You know, C and C++ are generally very

difficult languages to analyze, right?

Because they don't have memory safety.

But also,

they're good languages to analyze.

Because they don't have memory safety.

from our perspective, you know,

the worse the language, in a sense,

the more ways that you can, yeah,

shoot yourself in the foot.

The better for us in that we

can find more defects for you.

And so, hard is good in that sense.

You know, it's also just difficult

to parse languages like C and C++.

Java's got a, you know,

just a lot less diversity when it

comes to the compilers and the kind of

different language variance there are.

But then there's all these

dynamic languages, right?

Think JavaScript or Python and Ruby.

These have different challenges for static

analysis and I think there are static

analyzers out there that, that do analyze

these programs in these languages.

But they are generally

more shallow in nature,

because it's hard to really get

the same level of data that you

do from a static language in these dynamic

images just purely from static analysis.

So it can be difficult.

I think its still a research area for us.

I mean we're definitely exploring that and

I think for the academic community as well

its still very much a research topic.

Do you, are suggesting

then that you have, plans or

you would hope to at sometime in

the future analyse things like Python or

JavaScript other pretty popular languages

but, ones that don't fit into this sort of

statically typed traditional compiled

language sort of, framework?

I have to be a little careful

with talking about the future.

But I would, I would say that, that we're

certainly open to looking at all of,

all kinds of software and

all languages ultimately.

So you briefly mentioned a little

while ago dynamic analysis as compared

to static analysis and you discussed

a little bit about some of the trade-offs.

Do you support dynamic analysis in

you your, in your code advisor tool?

And if so what sorts of things do you use

it for as opposed to static analysis?

Yeah, there are some,

some uses of dynamic analysis within our

static analysis product, and we use it

primarily to, vet certain things that

are hard to determine purely statically.

So for example,um,

inside programs there might be

validation routines that checks that

something is valid credit card number, or

that it's a valid escaping routine for

cross hash scripting in certain contexts.

These kinds of routines you

can analyze statically but

we found that a relatively simple

dynamic approach testing approach can be

very effective at determining if these

are for example, proper escaping routines.

And so they don't produce a sound answer,

but almost always they are very,

very correct.

Because it's very unlikely that

we'll miss something if we've per,

fairly exhausted it.

So, it,

that's how we use dynamic analysis.

So there's certain core problems that are

easier for dynamic than static.

It sounds like these two

things are complimentary that is,

I could imagine you could use your static

tainting analysis to find out, well is

a sanitizer routine is an escaping routine

actually being used on all program paths?

And then separately using the dynamic

analysis to determine, well is

this actually a good escaping routine,

now that we know that it's being called.

Precisely, precisely.

And that's exactly, that's generally,

you know, at a high level of

abstraction that's, that's what we're

trying to do, is essentially elaborate

the strengths of both to augment

the analysis and make it smarter.

I know that in

the area of data race detection,

trying to find bugs in concurrent

programs, there seems to be more emphasis

these days anyways, on using dynamic

analysis to find those sorts of bugs.

Because it's very hard to

consider all of the dif,

the different threat

interviewings at compile time.

Is that, is that something that

that you guys do as well, or

is that maybe something that

you might do in the future?

so we do have analyzers

that detect concurrency bugs.

So things like risk conditions,

dead locks, and so forth.

It's an area that is difficult I would

say, but we do have detectors that do

a pretty good job at finding a certain

class of these kinds of errors.

In general it's, it's one of those

areas where static analysis can

provide some coverage if you write your

application code in a certain matter.

So for example if you have

fairly consistent use of

locks in your application there are many

kinds of defects it can detect for you.

But if you have fancier concurrency

mechanisms you have lock free data

structures and you have to be very

careful how you use them, and, and.

Like things like this it can be more

difficult for static analysis to just,

you know, understand all those

nuances in your application.

So while we, we generally concurrency

as in the broad strokes it depends

on how you use concurrency and what you

mean by it in a specific application.

It sounds like it comes back to this

point that static analysis has to

look at fairly limited properties.

And in the case that you can express

the property in a simple way,

like this lock is always held when

this variable is always accessed.

Then that's something static

analysis kind of turns around for.

But when the property is a little

bit more subtle, it's harder to

get static analysis to manage all of those

different tradeoffs, scalability.

I,

I wouldn't claim that it's necessarily not

possible or that it's hard, it's just from

a commercial standpoint there are fewer,

fewer people doing that

kind of advanced work.

And they generally have smaller and

smaller pieces of code, right,

in which to look at with those

complex kinds of uses, usages.

And so as a result from

a commercial standpoint it

might not be worth it to kind of delve too

deep into corner cases, actually, right.

The majority of the code that

uses concurrency for example,

Java is using fairly straightforward

synchronization primitives that are built

into the language and

those we can handle just fine.

So, I think it's more of a tradeoff

in terms of what's worth,

what's worth investing in

from a commercial standpoint.

But yeah, I mean there are algorithms

that can deal with complex you know,

data structures.

And concurrency primitives that have

been explored in research and if there's

enough demands, someday some concurrency

group that becomes really popular,

it's something we would

certainly consider.

so you just mentioned that

commercial viability was one factor in

determining whether you design

an analysis or not for your product.

Could you

say a little bit more about maybe other

factors that go into deciding how,

how do you decide when to do something

as a static analysis to extend your,

your product to, to look for some problem.

One of the factors is, you know,

the the criticality of the defects.

Right.

So

recently we had heart bleed right which is

a pretty severe security vulnerability.

And we quickly determined that

there was some way that we

can detect this defect and so we,

we augmented our analysis to do that.

And so there are cases like that and

there's other cases where we tried to

develop things that our

customers have asked for.

Right?

So we listen to our customers.

They give us feedback about

whatever kind of defect that

they're interested in finding.

They give us some examples, and we examine

that, and we build prototype chuckers, and

if those chuckers, can be developed in a

way that don't excessive false positives,

and still detect the errors

that they're interested in,

then we consider shipping them,

right, over time.

But there are cases, there's certainly

cases where people sent us examples of

defects which we'd give up on, right?

And you know, they just either not

detectable easily with static analysis or

we don't have the time to get to them or

the false positive read might be too high,

right?

In the latter case we also do things like

have optional flags that essentially

allow only certain customers to turn

those on because they really want it.

But if you, if you did it with

everybody's code you'd end up

with a lot of false positives so

there, there's that option as well.

Are you able to,

to trial run your detectors on

your existing customer's code?

In other words, let's say that customer

A requests a particular feature.

And you want to decide, okay is it

going to work with the you know,

false positives and so on.

I imagine you could certainly use the scan

project to help you out with that.

You have all this open source code

to help you determine false alarms.

Are you also able to ask your other

customers to trial run these ideas and

see how well they, they pan out?

so we,

we have a couple of mechanisms for that.

One is we can help the specific

customers that ask for

it to try out you know,

a beta version of the analysis.

Another thing that we do is

we run a lot tryouts, right?

So these are cases where we go out

to cust, to, not customers yet.

Prospects I would say.

And we give them a chance to run

our analysis and see the results.

And sometimes as part of trials

the the sales engineer that we

have that helps with the trial will

kind of let them know if they're,

if they're interested in a particular kind

of defect that we know we have you know

prototype for we might turn it on.

And, and give them an opportunity to

see what the results might look like.

Alright so they actually use that.

So that does happen as well and

it gets us some feedback about,

about things that would be hard to gather

otherwise as you might imagine since

these code bases are mostly proprietary.

Yeah I was thinking that,

I believe it's the case that,

I'm forgetting the name of the company.

They have, they have the business

model where you send them the code and

they run the analysis on it and

send you back a report.

Veracode.

That's the one.

>> Yeah, Veracode.

That's it.

So they would have the benefit of

they have access to all of this code.

Whereas I suppose if your, you send

to your product to your customer you,

you give them the option of whether

they send data back to you or not.

Right so we so one of

the differences between Vericode and,

and us is that they

analyze binaries right.

And so as a result they don't actually

need the source code or at least not

the full source code of the application

whereas we're elementing the source code.

So, it's the one difference and, and

the net effect of that is that we send our

product to the customer and they run it on.

2003 the world was

a very different place.

I think static analysis was not really

much of a commercial market at all.

There were a handful of products.

If you look at C++ there was products

like Lint and, there's a few others

on the market place, but the aggregate

size of the market was very, very small.

I'd say commercially at least there

were very few people working on

static analysis as a product, right.

And you know, since then,

you know, Coverity grew to

about 300 people as of 2014.

We were just acquired by

Synopsis this year so

I it, it's,

it's kind of not exactly the end, right?

Because it will continue

under the Synopsis rubric but

it's, it's certainly been a long journey.

And I think the market has

matured a huge amount right.

I think back in 2003 static analysis

was considered kind of you know,

not you know,

wasn't really commonly used I would say.

At least not in practice in

large development organizations.

And today I would say if you're doing

C++ development it's not you know,

absolutely required, right?

But it's getting close to

that point where, you know,

it's kind of scary to not do static

analysis if you're developing a,

a million lines of code application or

up certainly.

And, you know, I think,

I think that's a huge change from what it,

from where we started and

it's a good one, I think.

If it means software engineering

practices are gradually maturing and

new tools are being integrated in.

The other thing is security has just grown

in terms of it's importance so much.

I think think in the past when it all

started it was all about quality, right?

It was about finding crashes and

things that could cause your customers

to come back and scream at you.

But now security is such a high

profile problem and, and

something that's first and

foremost in developers' minds.

At least within organizations like,

maybe not individual developers, but

in terms of the organizations they work

for they care a lot about this problem.

And it's really changed the focus of,

of what kind of static analysis is,

is being worked on in a lot of companies,

including ours.

spoke a little bit before

about the core technology, like say, the,

the tainting analysis.

>> Yeah.

>> Did you find that when the security

became more of a focus, and, and maybe

Coverity started working more on security?

Did you have to develop new core analyses,

or did you find that many of the,

the analyses you'd already developed

could be retooled to work for security?

>> It was a combination.

There were certainly algorithms that

work fine for security checks and

there were other one where we had

to invent entirely new engines.

So that tainting engine I was talking

about was something we added specifically

for security.

And that was something we had

in some form in the past.

We got much more serious

about it.

We realized that

security issues really required you

to have a good tainting analysis.

And you know, I think that it's among,

it's among many problems that we had to,

to, to tackle when we

change gears like that.

Another is the reporting, for example.

The kind of reports that security

organizations care about and,

and what's reported on them.

It's very different than one on quality.

A quality-oriented kind of

report might look like.

So I guess the, this question, there,

there are two questions that

are two sides of the same coin.

So, what is what challenges do you see for

the industry going ahead and

on the flip side of that what are some

barriers to the practical adoption of,

of static analysis today as,

as far these companies believe.

>> Yeah.

In terms of looking ahead I

think there's lots of room for

expanding use of static analysis.

you, you mentioned languages before.

I think, that's a huge one,

right, in terms of

getting more access to static analysis

across these new dynamic languages where

they don't even have the basic you know,

type analysis in many cases.

Right, that, that we take for

granted in static languages.

But I think that if you look at the, the

industry as a whole, what's keeping static

analysis from being used more right, and,

you know, being more common practice.

I think part of the answer is that it's

just a change in culture and mentality.

It's very difficult for

people who aren't used to dealing with

this new source of defects, right?

To deal with that, right?

Suddenly they turn this

analysis engine on and

they're faced with a few thousand bugs.

And they don't know what to do with that.

They already have too many

bugs in their database and

they can't fix them all anyway,

so why add a few thousand more?

but, you know, the thing about static

analysis bugs is that they're,

they're pinpointed in the code, right?

There's an actionable kind of, there's

no need to kind of reverse engineer how,

you know, how the bug occurs.

It's kind of a root cause.

And so, you know,

changing the mentality about how

you can efficiently address these

defects earlier in the cycle.

And getting people to realize that if

they're proactive about fixing those,

they might have fewer of those hard to,

you know,

hard to understand defects

in the field of futures.

It's a, it's difficult.

It's kind of like getting

people to exercise.

Or, or, dig it, you know, do,

you know, g, have a good diet.

It, it, before you have

a problem it might seem like oh,

that's other people's problem, not mine.

But until, until it's your problem, right.

And then you take it seriously.

So I think that, that gradually

organizations are realizing they do

need to be more proactive about ensuring

security and quality of applications and

that's where we can help.

>> So, this been great I it's

been really interesting to

hear about especially the a commercial

perspective on static analysis.

I wonder if you in your experience in

working with companies and thinking about

how static analysis fits in the overall

process of making programs more secure,

there are other tools or techniques that

companies talk about that you have to

think about when you write your tools so

that you fit within the overall ecosystem.

Things we, we mentioned already maybe

testing, penetration testing perhaps, or

manual code review.

Are there other things

that that you hear about?

Maybe the use of certain operating

systems or deployment environments.

Things like that,

that static analysis has to think about.

>> Yeah I mean, I think that most sort of,

direct touch points for static

analysis include built systems, right?

So built systems are complicated

in integrating with them.

So that you can actually see all

the code is not trivial, usually.

Another example is the use of,

bug tracking systems.

If, if you want to get your bugs

fixed often you've got to push them

into a system that people are already

using for, assigning work to each other.

Right?

A third is source control systems, right?

These are systems that keep track of,

you know,

who wrote what code and

without the information it's hard to

assign these defects that pop out

of static analysis to a person.

Right?

And

if they're just sitting there in

the ether nobody works on them,

so these are the common

integration points that we have.

But the big picture is that you,

you need to really integrate with the flow

that developers use in each company, and

it's, each company has its own unique

you know, often home-grown kind of

methodology for how they do development,

at least in the details, right?

And integrating with all of those

details is, is, often a challenge and

is part of the reason why, you know,

professional services or, or just,

you know, someone champion inside

the company that understands how to

put this new thing, this foreign thing,

into the organization, is,

is often required to get static

analysis to be effective.

>> Okay, great.

I wonder if there is there's anything

we haven't talked about that you think

is important that people ought to know

about static analysis or about Coverity.

Just any, any parting thoughts?

You know, one thing that we just

launched recently is a new version,

it's not, it's not Coverity Scan,

but it's called Codespotter.

Right?

And it's essentially a free service for,

not just open-source projects, but

kind of homegrown projects as well, so

if you have some code at GitHub

that's written in Java today,

then you can use Codespotter on it and

get analysis results.

And so that's an interesting new dynamic I

would say is this idea of putting code in

the cloud and development in the cloud.

And it's, it's a trend that I

think actually does benefit static

analysis technologies

in that it's emerging.

we're, we'd like to kind of write that

trend as well and, and help, you know,

developers that want to do

that kind of development.

It moved more of their flow

into the cloud, and, and

static analysis should be a part of that.

So we're excited to see that.

I think, you know,

kind of the big picture is that,

you know, if you care about security,

if you care about quality, you know,

there's like I said many

different things you can do.

And I think static analysis

is one of those things and

its one of the things that works,

you know, while you code in a sense.

Right.

Like while you're first putting out

that code you can actually get

feedback really in the process.

And, unlike techniques that wait until,

let's say, an expert comes in a few

months later or years later to look

at your code, you can find it now.

And I think that's something I hope

developers can keep in mind that you know,

security is part of their jobs and

as well as quality.

And you can't just rely on, you know,

outside experts to just to do it for you.

It really requires developers

get proactively involved and

to the extent that we can

help that we're happy.

But I am also if developers were just

more aware of the problem and, and

just you know, can learn more about and be

more proactive about fixing bugs earlier.

>> Fantastic.

That's I have to say, that's, part of my

goal in this class is that too often,

people think of security as something you

can solve by putting a box on the edge of

your network and, really, that's,

that's not going to scale.

The way you're going to solve your

security problems is to build that network

better in the first place.

>> Exactly.

>> So one maybe fun question is have you

used Coherity Code Advisor on itself?

Right, it's a program and

you can run it and evaluate itself.

How does it do if you do?

>> yes.

We do analyze [LAUGH] our own products.

And we do have a, dashboard that's sitting

in our kitchen that everyone can see it

at lunchtime, and so we keep a close

eye on our analysis results.

And we've found bugs in

our own product before.

Thankfully before shipping for

the most part.

>> [LAUGH].

>> And we track and we release,

we say no new defects and

we whittle down the legacy

defects if there are any,

many products are at zero your

know findings at this point.

So, we, we definitely believe in it.

I mean, we're believers in what

we build and we use it for sure.

(From Prof. Michael Hicks interviewing Coverity co-founder, Andy Chou).

Penetration testing, or pen testing for

short, is a direct assessment of the

security of a complete software system.

Its goal is to find

evidence of insecurity,

typically taking the form of

exploitable vulnerabilities.

Pen testing is a black hat activity done

by so-called red teams or tiger teams, and

employed for the good purpose of finding

security defects prior to deployment.

What is the target of a penetration test?

The focus can be any of several

different levels of a system made up

of executable components.

We can fuzz test single programs or we can

look at complete applications, like web

applications, consisting of communicating

programs, the browser and server.

We can even look at entire networks,

looking for

weaknesses that cross

application boundaries,

where one system could be exploited

to then take advantage of another.

In all cases, we're looking at

whole programs, not parts thereof,

like code fragments or

configuration files, libraries and so on.

Now who are pen testers,

and how do they work?

Well pen testers are teams

that use guile and

automated tools to find security issues.

A good pen tester is creative,

thinking carefully about how a system is

put together to find assumptions

that turn out to be weaknesses.

Good tool support is essential for

an effective pen testing engagement.

As the pen tester comes up with

hypotheses about potential weaknesses,

tool support can be used to

systematically probe a target and

see if those weaknesses are present.

Pen testing is carried about by the, late

in the development process by a team that

is different from the one

that built the system.

Having a separate team ensures that

the target is given a fresh look.

Developers can have blind spots

about their own software and

a separate team can help see past those.

A separate team is also useful in that pen

testing requires a specialized skill set

and this skill set can be developed

across multiple projects.

The pen testing team may be told a lot or

a little about the system that they

are testing, and given a lot or a little

access to its deployment environment.

We might, on the one hand, try to simulate

the access that an attacker should have.

For example, only from outside a company

firewall, via forward facing components.

On the other hand,

we might provide more access or

information to model an attack

by a company insider.

In general, giving an adversary

more powers gives a better

idea of how the system holds up

if some defenses fail to hold.

That said, if the design was against a

weaker threat model than it's being tested

against, then it shouldn't be surprising

that we'll find additional issues, and

discovered issues should be assessed and

kept in perspective.

By the 1970s, the government was regularly

using teams to assess the security of

computer systems by

trying to penetrate them.

These teams were referred to as red teams,

or tiger teams.

Today, penetration testing is a mature

field, with services provided by

independent companies, as well as

divisions within larger organizations.

Much of the buzz about cybersecurity,

particularly among students,

focuses around penetrations,

which are gamified in CTF, or

capture the flag competitions,

like DefCon.

There is also a penetration

testing certification organized by

the Information Assurance

Certification Review Board, or IACRB.

The certification is based on

both conceptual material and

a demonstration of skills by

hacking into a target VM.

Why should we do pen testing?

Well, it has several benefits.

First, penetrations are certain.

After all, they have been demonstrated.

Hopefully, they also come

with some reusable evidence.

For example, evidence of an SQL injection

could be the exploit payload entered in

a form field of a web application.

As such, unlike errors produced by,

say, code reviews or

static analysis tools,

penetrations are not hypothetical.

By virtue of the fact that

they are applied to whole,

deployable components they are also

relevant under realistic configurations.

And, just as they are not hypothetical,

they're not wrong.

Penetrations are not, or

are rarely, false alarms.

Pen testing has non-technical

benefits as well.

One of them is a feel good factor.

After completing an engagement those

responsible for the target system can feel

like they have really improved security or

will soon once they fix the issues.

And this is because they have found real

vulnerabilities that probably would have

gone unfixed and could really

have been exploited in the wild.

Before this point,

such vulner, vulnerabilities were

only hypothetical, not made manifest.

Now on the other hand,

penetration testing is not a panacea.

We have to be careful not to assume we're

getting more out of it than we really are.

Most importantly,

penetration testing will not find all of

a system's security problems.

As such, an absence of any

discoveries does not imply

the absence of vulnerabilities.

Likewise, fixing

vulnerabilities that were found,

if any, does not mean that none are left.

Penetration testers may not even have

looked for certain sorts of problems,

depending on the rules of the engagement

and the assumptions of the threat model.

Security, to the extent that pen testing

has established it, is ephemeral.

We cannot rest on our laurels.

When you change the software, the

configuration, the network topology, and

so on, you potentially

create new vulnerabilities.

Well, an important thing to keep in

mind is that security, in general,

is not compositional.

This observation was first

made by Leslie Lamport,

recent Turing Award winner,

back in the 1970's.

What it means is that two components

that are secure on their own

are not necessarily secure

when used in combination.

Penetration testing looks at the whole

system and that's its advantage, but

subsequent changes might break a component

and thus compromise the system.

Even worse, a change to one component

might not break that component, but

could break the whole system anyway

due to the lack of compositionality.

As such, we must employ

the other processes discussed in

this course to reduce the chances as much

as possible that we break components.

And we must employ whole system testing

to ensure that the composition works too.

In short, despite its limitations,

penetration testing is something

very much worth doing.

We will see that pen testing

is both art and science.

As an art form it relies on the creativity

and ingenuity of the pen tester.

As clearly successful techniques emerge,

pen testing moves to becoming a science.

And the fruits of this science

are turned into tools and

automation that subsequent

pen testers can use.

And we'll look at several

tools in particular.

First, Nmap is a tool for scanning

networks, probing for computers and

other devices that might

be targets of attack.

Zap, another tool to look at,

is a web proxy that intercepts

communications between a web browser and

a web server, allowing the pen tester to

see what's going on and to manipulate

it looking for vulnerabilities.

Third, metasploit is a tool for

developing and deploying exploits.

It is highly configurable with a tool

kit that pen testers can use to find and

exploit vulnerabilities.

Now in the second part of the unit

we'll consider fuzz testing or

fuzzing, a technique that many pe,

penetration testing yules, tools employ.

Fuzz testing works by

corrupting inputs and

interactions between target components.

The goal is to see whether such

corruptions will cause the system to

break in a way that could

be exploited in an attack.