Beyond Compilers

This Time

Lecture Outline

Beyond Compilers

- Using compiler techniques for other stuff
- Next Steps

Final Words

- What next for EECS 665
- Logistical wrap-up

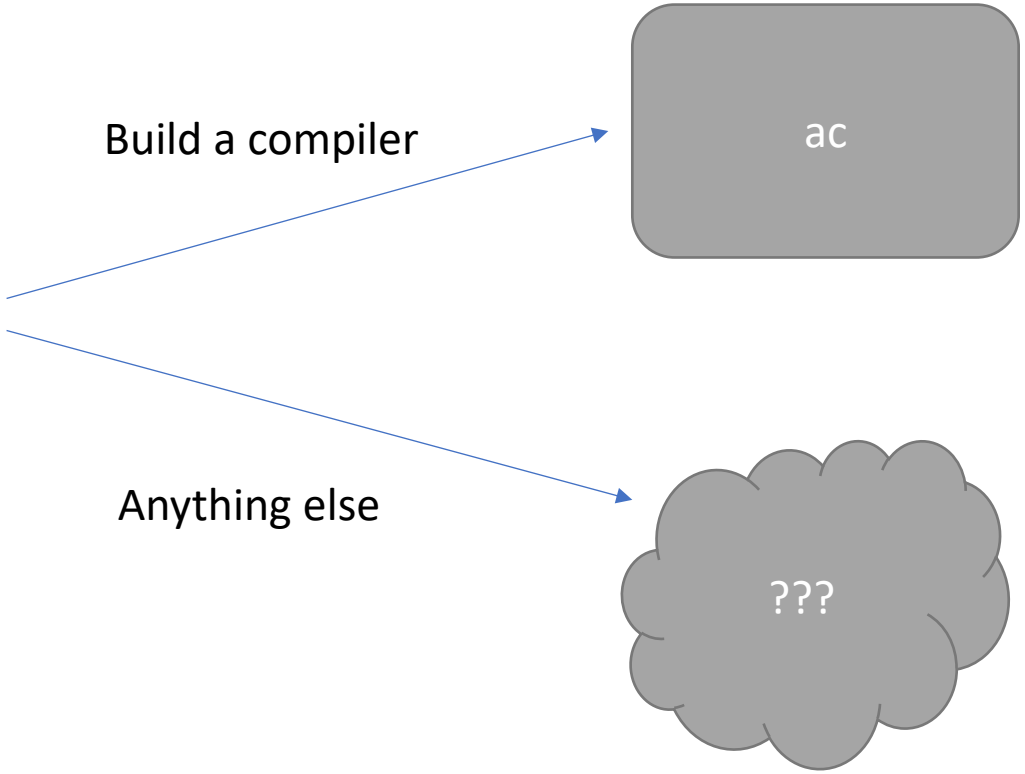
Just Drew Talkin'

Compiler Techniques Outside Compilers

Beyond Compilation

Why Compilers Matters (maybe?)

i.e. why Drew is qualified to teach compilers



This Lecture

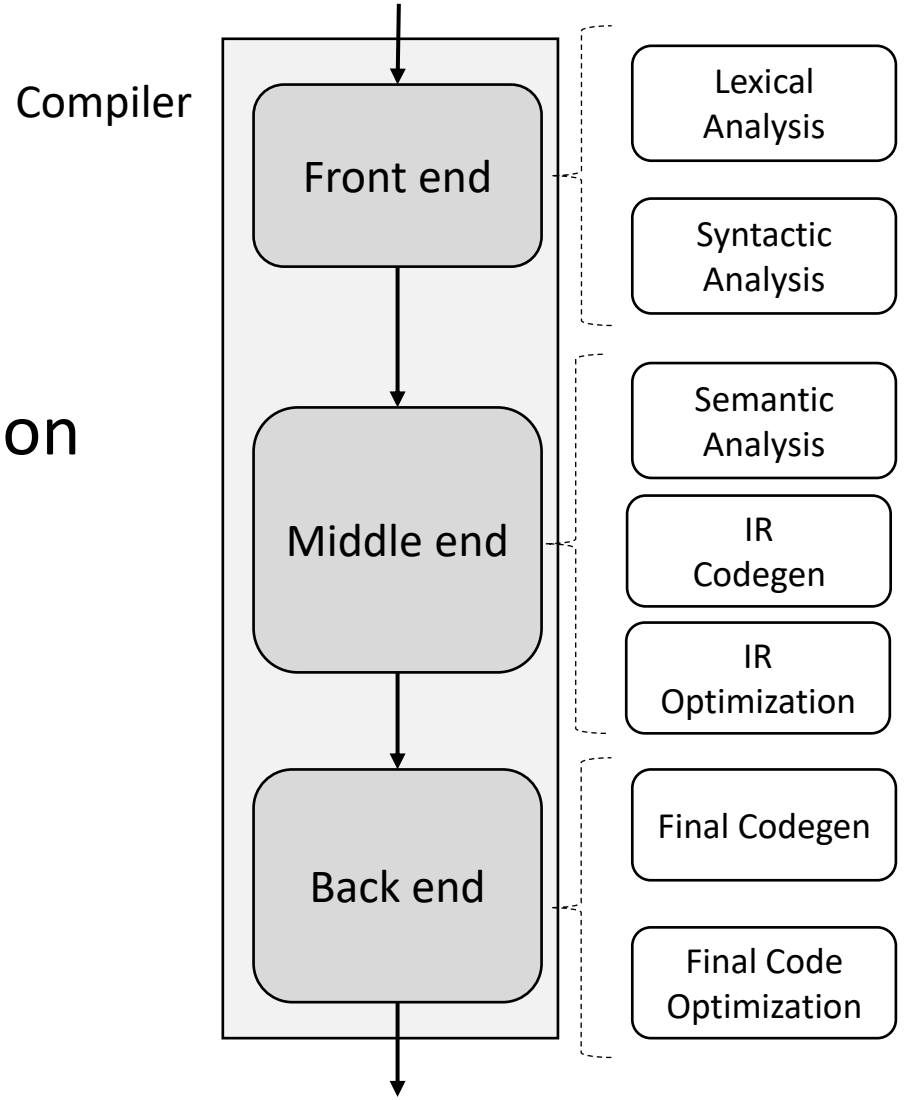
Beyond Compilation

Compiler Techniques

- Lexing
- Parsing
- Syntax-Directed Translation
- Semantic Analysis
- IRs
- Architecture

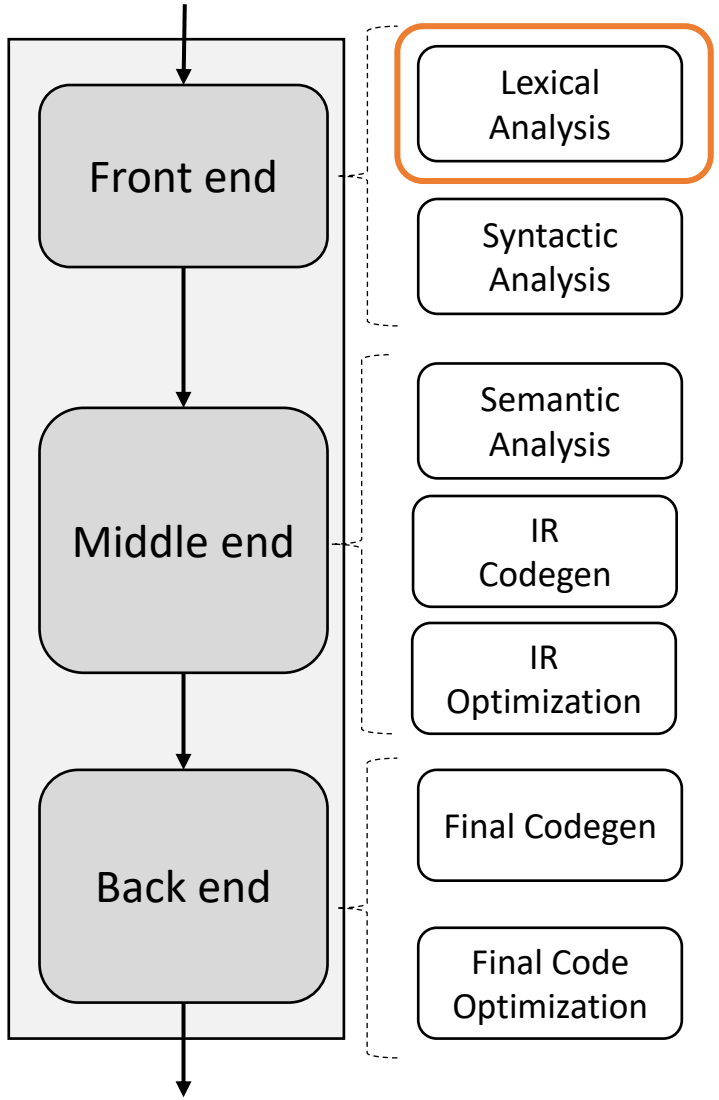
Utilities

- RE



Compiler Techniques Outside Compilers

Beyond Compilation



Signature Matching

Beyond Compilers

- Regular-expression based matching of network traffic
- Rather than producing tokens, produces actions

```
alert tcp $EXTERNAL_NET any ->
$HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-IIS cmd.exe access";
flow:to_server, established;
content:"cmd.exe"; nocase; classtype:web-
application-attack; sid:1002; rev:5;)
```

Policy Automata

Beyond Compilers

- Enforce stateful policies on system call sequences
- Expressible via finite state automaton

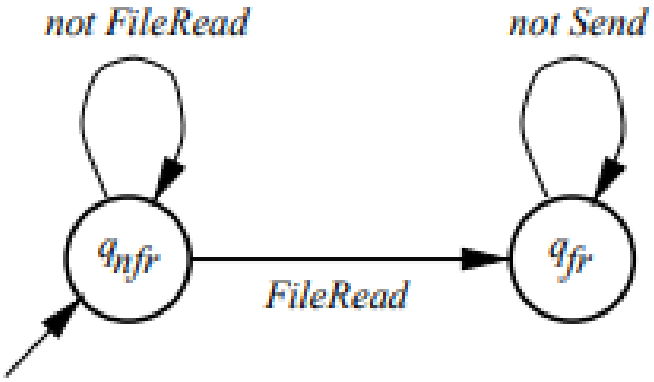
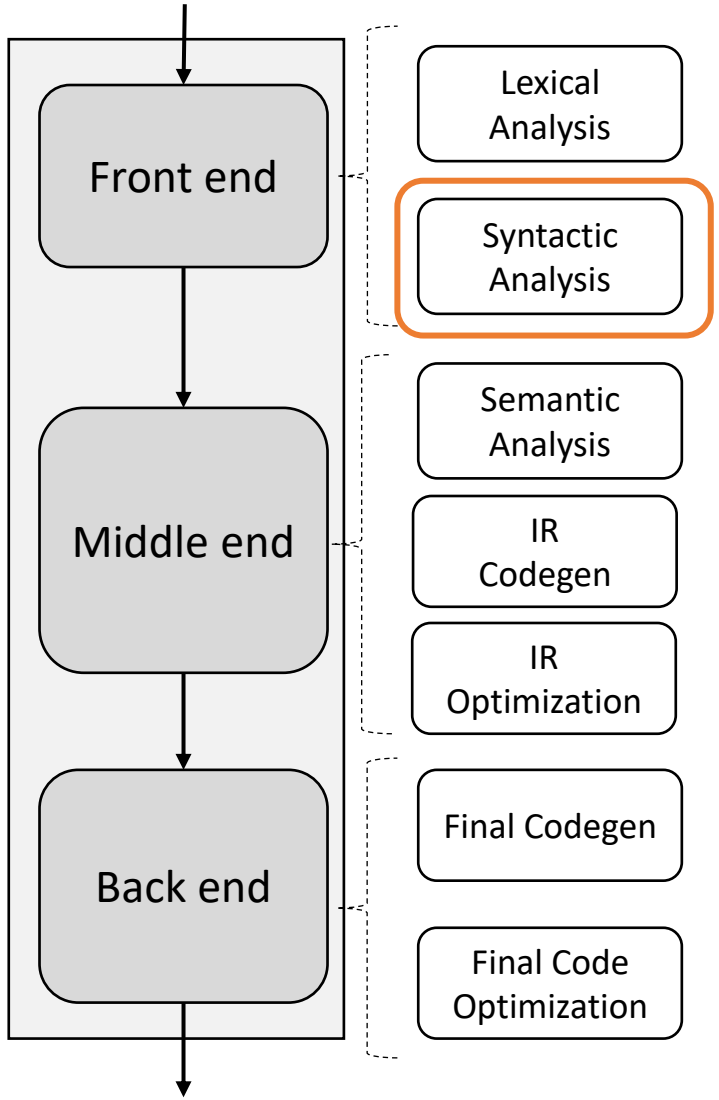


Fig. 1. No *Send* after *FileRead*.

Compiler Techniques Outside Compilers

Beyond Compilation



Protocol Normalization Using AGs

Beyond Compilation

- Uses SDT to rewrite grammar to a canonical form
- Actually uses BISON!
- More PoC than production-ready code

Protocol Normalization Using Attribute Grammars

Drew Davidson[1], Randy Smith[1], Nic Doyle[2], and Somesh Jha[1]

[1] Computer Sciences Department, University of Wisconsin, Madison, WI 53706
[2] ERBU XE Security group, CISCO systems

Abstract. Protocol parsing is an essential step in several networking-related tasks. For instance, parsing network traffic is an essential step for Intrusion Prevention Systems (IPSs). The task of developing parsers for protocols is challenging because network protocols often have features that cannot be expressed in a context-free grammar. We address the problem of parsing protocols by using attribute grammars (AGs), which allow us to factor features that are not context-free and treat them as attributes. We investigate this approach in the context of protocol normalization, which is an essential task in IPSs. Normalizers generated using systematic techniques, such as ours, are more robust and resilient to attacks. Our experience is that such normalizers incur an acceptable level of overhead (approximately 15% in the worst case) and are straightforward to implement.

1 Introduction

Parsing application-layer protocols is a fundamental step in several networking-related tasks. Programs that operate over application-level traffic semantics, such as systems that investigate Email traffic and Internet attacks, use a protocol parser as an integral component. Parsing network traffic is also an essential step for Intrusion Prevention Systems (IPSs) because protocols allow many representations of the same message. *Protocol normalization* is meant to reverse the transformations and obfuscations that an attacker performs on a message to a canonical form [7]. An IPS that does not perform normalization is vulnerable to evasion attacks [7,15,17]. In order to perform normalization, IPSs must know certain fields in a protocol, e.g., to normalize URLs an IPS system has to extract the URL field from HTTP traffic. In this paper we focus on protocol parsing in the context of intrusion prevention, but the results are applicable to related areas such as firewalls, URL filtering, and HTTP server load balancing.

At first glance implementing application protocol parsers seems like a straightforward task. One strategy would be to use standard parser generators such as *yacc* [9] or *ANTLR* [14] to implement an application protocol parser. This strategy often does not work, however, because many protocols have constructs that are not context-free. For example, data fields that are preceded by their actual length (which is common in several network protocols) cannot be expressed in a context-free grammar [13]. In this work we consider a systematic approach to the

M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789, pp. 216–231, 2009.

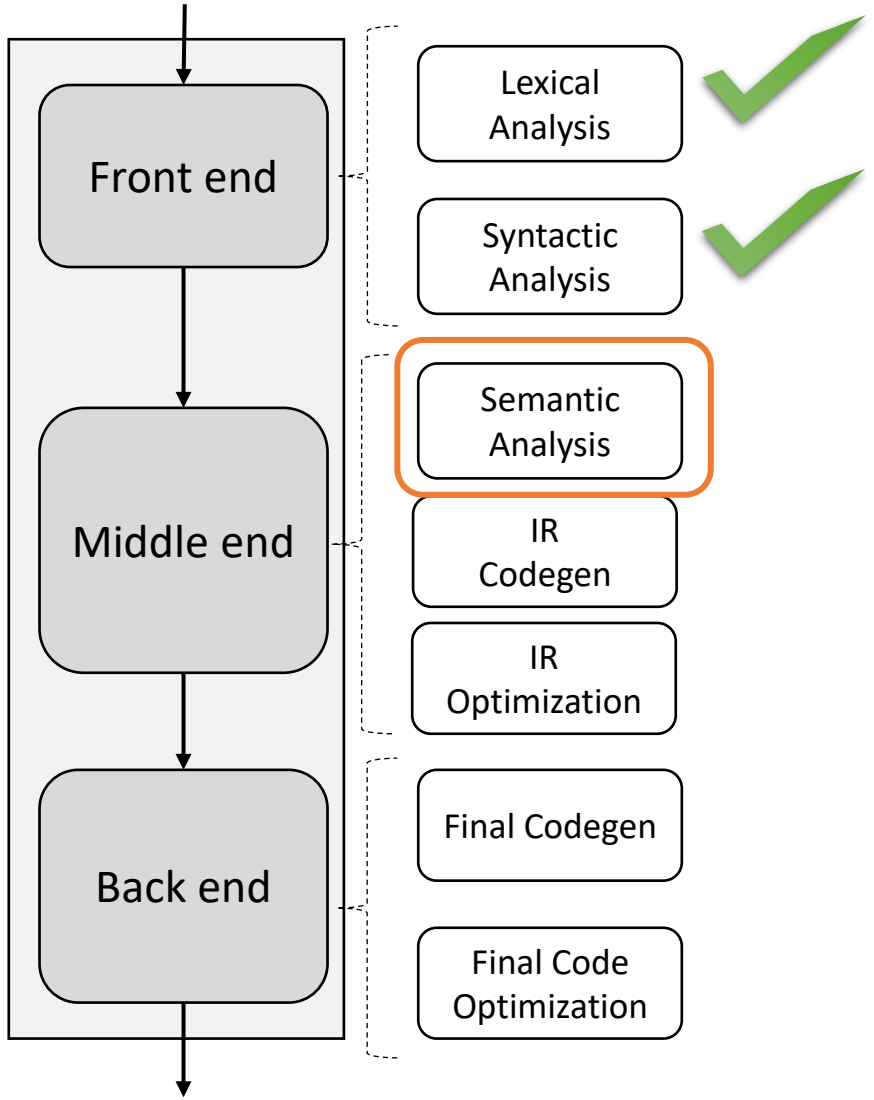
Clang-Format

Beyond Compilation

- Code (re)formatter
- Ensures that all code in a project is of a consistent style

Compiler Techniques Outside Compilers

Beyond Compilation



Static Analysis

Beyond Compilation

We did a bit already:

- Typechecking
- Data flow

We focused on faithfully translating program

- But what about when the program is garbage?

Information Leakage

Beyond Compilation

We'd like to know if a program can leak secret/private data

- It would be good to know that *before* running the program
- Use static analysis!
  - Can data that comes from a sensitive source reach an untrusted sink?

Static Instrumentation

Beyond Compilation

Add “reporting” to executables

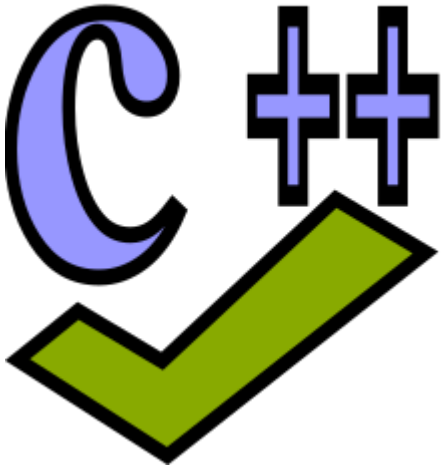
- *Test suite line coverage:* Is every line of the program exercised by the test suite?
- *Taint tracking:* determine if user input is ever touching protected memory

Static Analysis: cppcheck

Beyond Compilation

Tool for common CPP errors

- Check for semantic errors / likely logic issues
- Sorta like a standalone version of the warning phase of the compiler



Linters

Beyond Compilation

Check source code for likely bugs

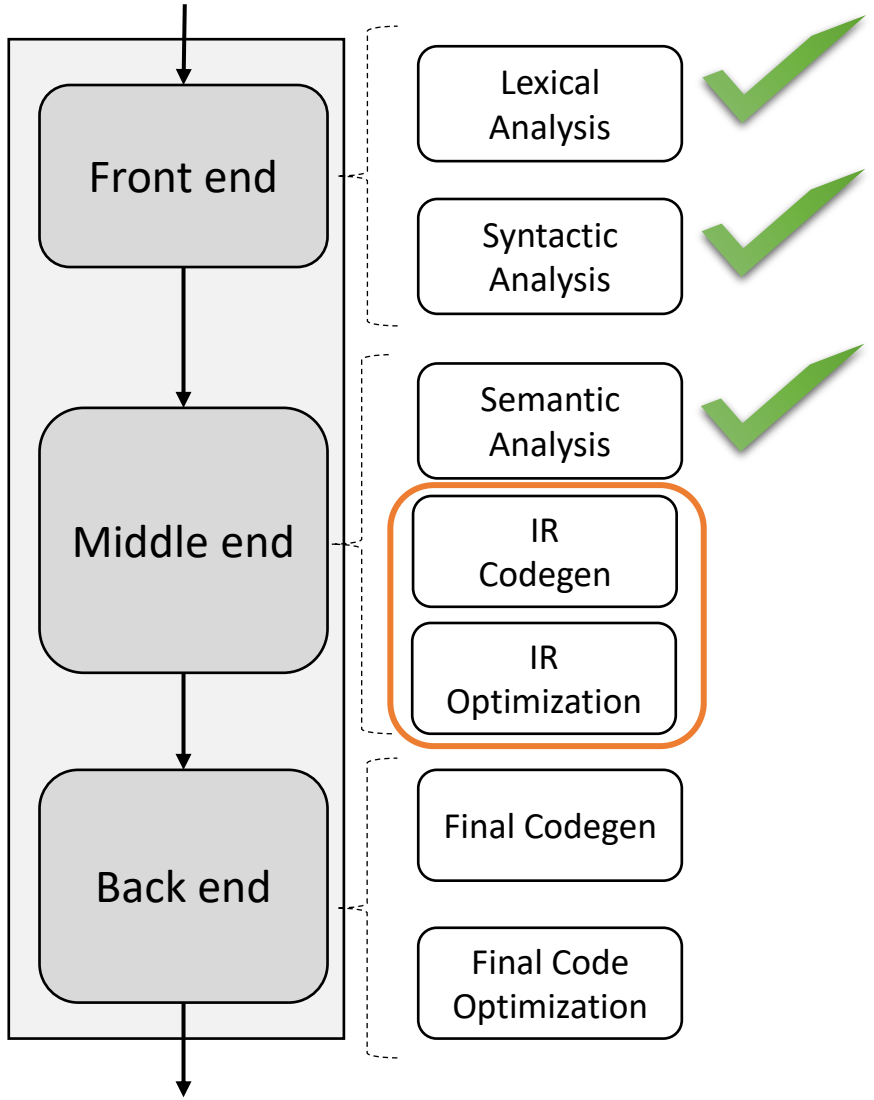
- cpplint
- pylint / flake8
- bandit

```
int a = 2;
if (a = 3) {
    //whatever
}
```

Dryer Lint (a different kind of lint)

Compiler Techniques Outside Compilers

Beyond Compilation

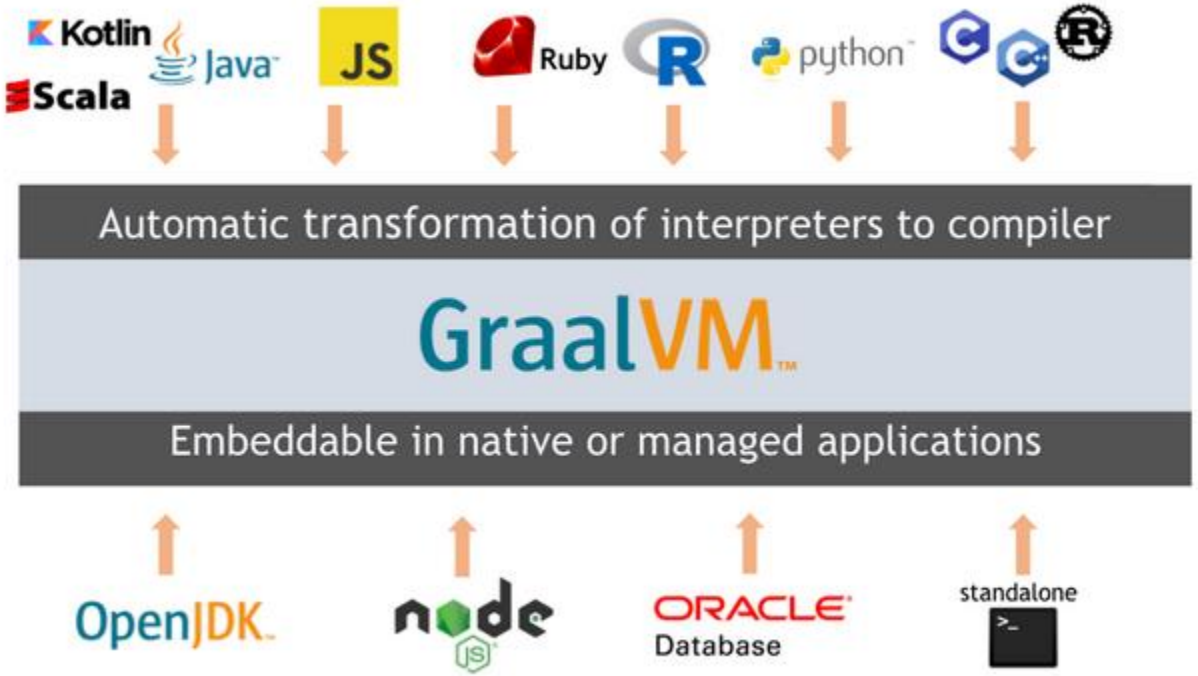


“Old School” Virtual Machines

Beyond Compilation

Abstract hardware architecture

- Compile once, run anywhere



FiE on Firmware!

Beyond Compilation

- Analyze behaviors of 100s of related IoT programs

FIE on Firmware:
Finding Vulnerabilities in Embedded Systems using Symbolic Execution

Drew Davidson Benjamin Moench Somesh Jha Thomas Ristenpart
University of Wisconsin–Madison, {davidson,bsmoench,jha,rist}@cs.wisc.edu

Abstract

Embedded systems increasingly use software-driven low-power microprocessors for security-critical settings, surfacing a need for tools that can audit the security of the software (often called firmware) running on such devices. Despite the fact that firmware programs are often written in C, existing source-code analysis tools do not work well for this setting because of the specific architectural features of low-power platforms. We therefore design and implement a new tool, called FIE, that builds off the KLEE symbolic execution engine in order to provide an extensible platform for detecting bugs in firmware programs for the popular MSP430 family of microcontrollers. FIE incorporates new techniques for symbolic execution that enable it to verify security properties of the simple firmwares often found in practice. We demonstrate FIE's utility by applying it to a corpus of 99 open-source firmware programs that altogether use 13 different models of the MSP430. We are able to verify memory safety for the majority of programs in this corpus and elsewhere discover 21 bugs.

1 Introduction

Embedded microprocessors are already ubiquitous, providing programmatic control over critical, increasingly Internet-connected physical infrastructure in consumer devices, automobiles, payment systems, and more. Typical low-power embedded systems combine a software-driven microprocessor, together with peripherals such as sensors, controllers, etc. The software on such devices is referred to as *firmware*, and it is most often written in C.

The use of firmware exposes embedded systems to the threat of software vulnerabilities, and researchers have recently discovered exploitable vulnerabilities in a wide variety of deployed embedded firmware programs [12, 18, 19, 21, 22, 24, 27]. These bugs were found using a combination of customized fuzz testing and manual reverse engineering, requiring large time investments by those with rare expertise.

To improve firmware security, one possible approach would be to use the kinds of source-code analysis tools that have been successful in more traditional desktop and server settings (e.g., [2, 4, 8, 9, 11, 13, 17, 26, 28, 31, 36]). These tools, however, prove insufficient for analyzing firmware: the microcontrollers used in practice have a wide range of architectures, the nuances of which frustrate tools designed with other architectures in mind (most often x86). Firmware also exhibits characteristics dissimilar to more traditional desktop and server programs, such as frequent interrupt-driven control flow and continuous interaction with peripherals. All this suggests the need to develop new analysis tools for this setting.

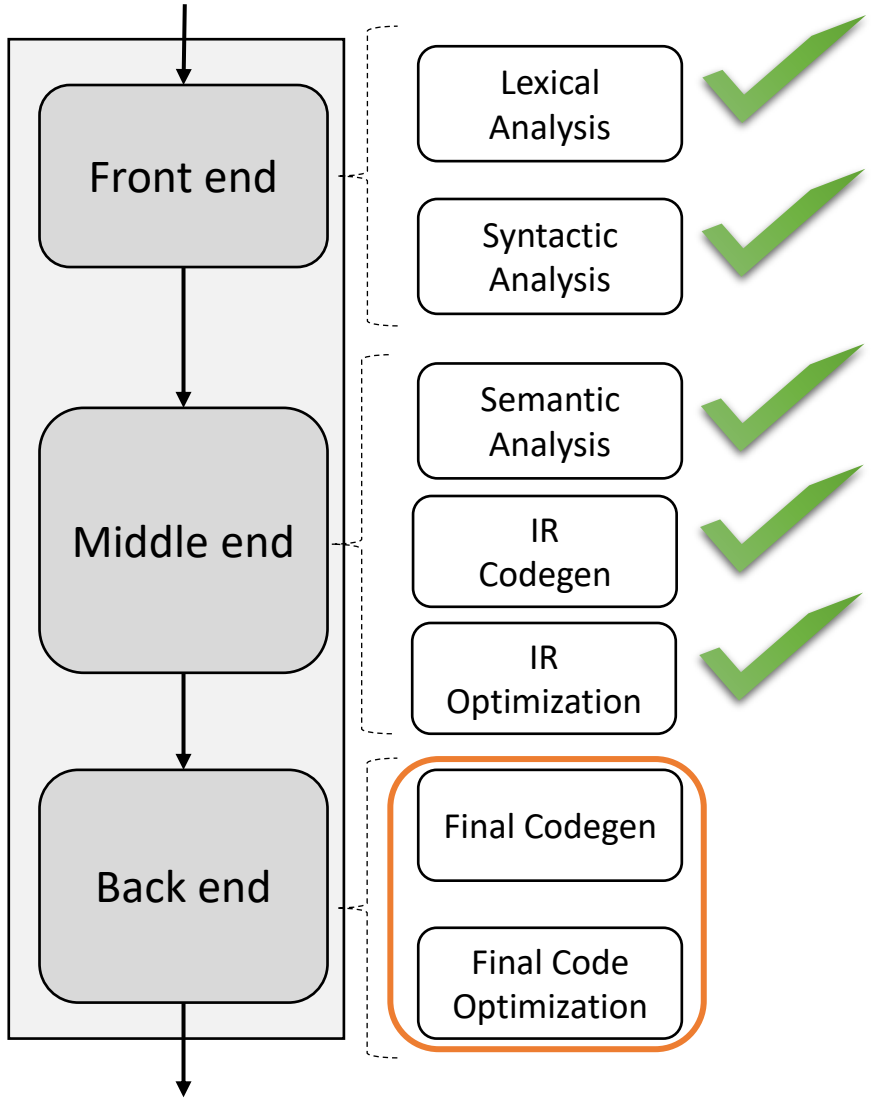
We initiate work in this space by building a system, called FIE, that uses symbolic execution to audit the security of firmware programs for the popular MSP430 family of 16-bit microcontrollers. We have used FIE to analyze 99 open-source firmware programs written in C and of varying code complexity. To do so, FIE had to support 13 different versions of the MSP430 family of 16-bit RISC processors. Our analyses ultimately found 20 distinct memory-safety bugs and one peripheral-misuse bug.

We designed FIE to support analysis of all potential execution paths of a firmware. This means that, modulo standard but important caveats (see Section 6), FIE can verify security properties hold for the relatively simple firmware programs often seen in practice. For example, we verify memory safety for 53 of the 99 firmware programs in our corpus.

Overview of approach: FIE is based on the KLEE symbolic execution framework [10]. In addition to the engineering efforts required to make KLEE work at all for MSP430 firmware programs, we architected FIE to include various features that render it effective for this new domain. First, we develop a modular way to specify the memory layout of the targeted MSP430 variant, the way in which special memory locations related to peripherals should be handled, and when interrupt handlers should

Compiler Techniques Outside Compilers

Beyond Compilation



Assembly Code: Low Level Security

Beyond Compilation

- Buffer overflows
- ROP
  - Blind ROP

Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, Dan Boneh

Stanford University

***Abstract*—We show that it is possible to write remote stack buffer overflow exploits without possessing a copy of the target binary or source code, against services that restart after a crash. This makes it possible to hack proprietary closed-binary services, or open-source servers manually compiled and installed from source where the binary remains unknown to the attacker. Traditional techniques are usually paired against a particular binary and distribution where the hacker knows the location of useful gadgets for Return Oriented Programming (ROP). Our Blind ROP (BROP) attack instead remotely finds enough ROP gadgets to perform a `write` system call and transfers the vulnerable binary over the network, after which an exploit can be completed using known techniques. This is accomplished by leaking a single bit of information based on whether a process crashed or not when given a particular input string. BROP requires a stack vulnerability and a service that restarts after a crash. We implemented Braille, a fully automated exploit that yielded a shell in under 4,000 requests (20 minutes) against a contemporary nginx vulnerability, yaSSL + MySQL, and a toy proprietary server written by a colleague. The attack works against modern 64-bit Linux with address space layout randomization (ASLR), no-execute page protection (NX) and stack canaries.**

I. Introduction

Attackers have been highly successful in building exploits with varying degrees of information on the target. Open-source software is most within reach since attackers can audit the code to find vulnerabilities. Hacking closed-source software is also possible for more motivated attackers through the use of fuzz testing and reverse engineering. In an effort to understand an attacker's limits, we pose the following question: *is it possible for attackers to extend their reach and create exploits for proprietary services when neither the source nor binary code is available?* At first sight this goal may seem unattainable because today's exploits rely on having a copy of the target binary for use in Return Oriented Programming (ROP) [1]. ROP is necessary because, on modern systems, non-executable (NX) memory protection has largely prevented code injection attacks.

To answer this question we start with the simplest possible vulnerability: stack buffer overflows. Unfortunately these are still present today in popular software (*e.g.,* nginx CVE-2013-2028 [2]). One can only speculate that bugs such as these go unnoticed in proprietary software, where the source (and binary) has not been under the heavy scrutiny of the public and security specialists. However, it is certainly possible for an attacker to use fuzz testing to find potential bugs through known or reverse engineered service interfaces. Alternatively, attackers can target known vulnerabilities in popular open-source libraries (*e.g.,* SSL or a PNG parser) that may be used by proprietary services. The challenge is developing a methodology for exploiting these vulnerabilities when information about the target binary is limited.

One advantage attackers often have is that many servers restart their worker processes after a crash for robustness. Notable examples include Apache, nginx, Samba and OpenSSH. Wrapper scripts like `mysqld_safe.sh` or daemons like `systemd` provide this functionality even if it is not baked into the application. Load balancers are also increasingly common and often distribute connections to large numbers of identically configured hosts executing identical program binaries. Thus, there are many situations where an attacker has potentially infinite tries (until detected) to build an exploit.

We present a new attack, Blind Return Oriented Programming (BROP), that takes advantage of these situations to build exploits for proprietary services for which both the binary and source are unknown. The BROP attack assumes a server application with a stack vulnerability and one that is restarted after a crash. The attack works against modern 64-bit Linux with ASLR (Address Space Layout Randomization), non-executable (NX) memory, and stack canaries enabled. While this covers a large number of servers, we can not currently target Windows systems because we have yet to adapt the attack to the Windows ABI. The attack is enabled by two new techniques:

1) Generalized stack reading: this generalizes a known technique, used to leak canaries, to also leak saved return addresses in order to defeat ASLR on 64-bit even when Position Independent Executables (PIE) are used.
2) Blind ROP: this technique remotely locates ROP gadgets.

Both techniques share the idea of using a single stack vulnerability to leak information based on whether a server process crashes or not. The stack reading technique overwrites the stack byte-by-byte with possible guess values, until the correct one is found and the server does not crash, effectively reading (by overwriting) the stack. The Blind ROP attack remotely finds enough gadgets to perform the `write` system call, after which the server's binary can be transferred from memory to the attacker's socket. At this point, canaries, ASLR and NX have been defeated and the exploit can proceed using known techniques.

The BROP attack enables robust, general-purpose exploits for three new scenarios:

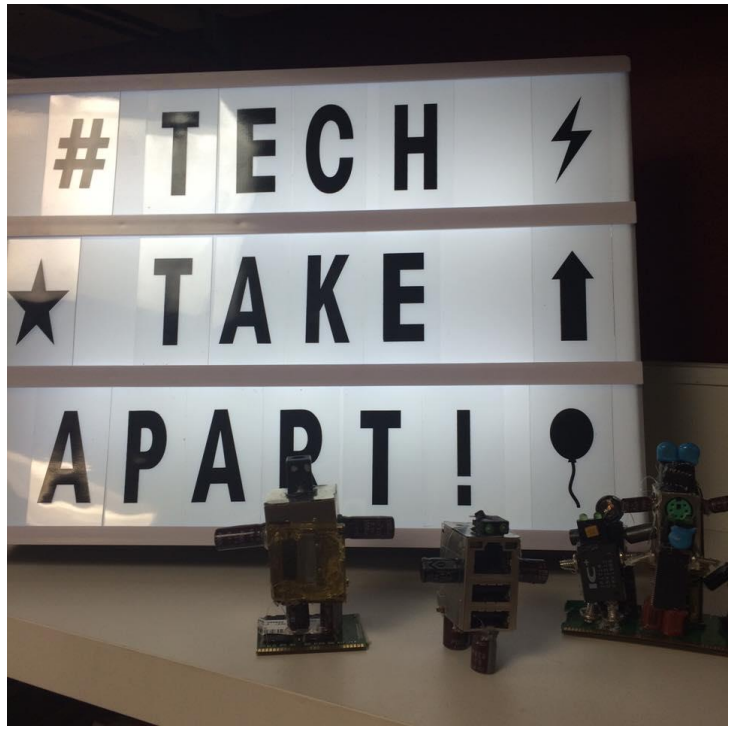
1) Hacking proprietary closed-binary services. One may notice a crash when using a remote service or discover one through remote fuzz testing.
2) Hacking a vulnerability in an open-source library thought to be used in a proprietary closed-binary service. A popular SSL library for example may have

Decompilation

Beyond Compilation

Run the compiler toolchain in reverse

- Binary to assembly
  (disassembler)
- Assembly to source
  (decompiler)



Decompilation

Beyond Compilation

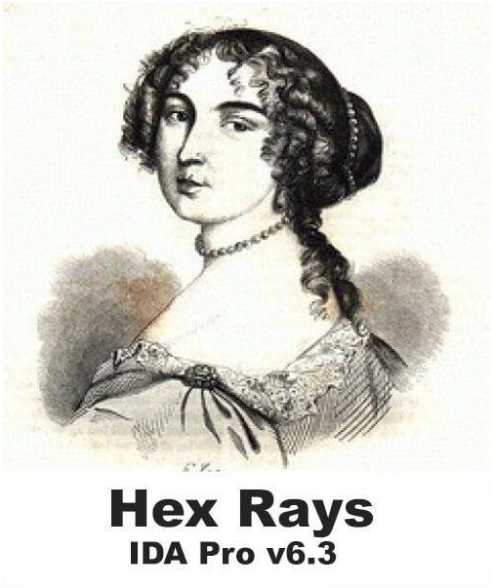
Run the compiler toolchain in reverse	Popular tools	
	(Commercial)	(Free)
• Binary to assembly *(disassembler)*	IDA Pro	Ghidra
• Assembly to source *(decompiler)*	Hex Rays	

Reverse Engineering Tools

Beyond Compilation

IDA Pro

- Long considered the best and tool by default
- Builds CFG from binary

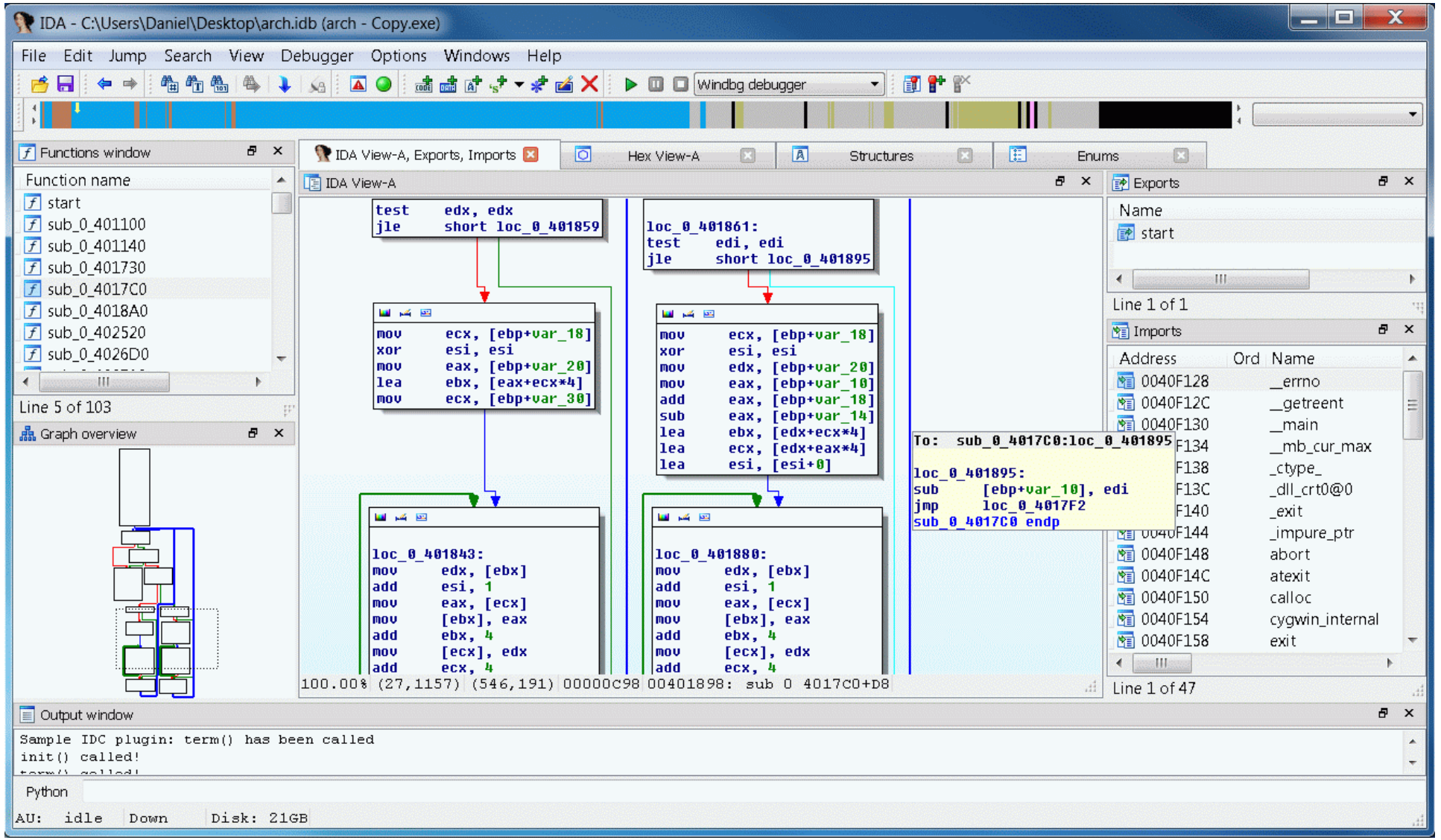


Ghidra

- Developed by the NSA
- Released (open source) April 4, 2019

Reverse Engineering Tools

Beyond Compilation



Reverse Engineering Tools: Why?

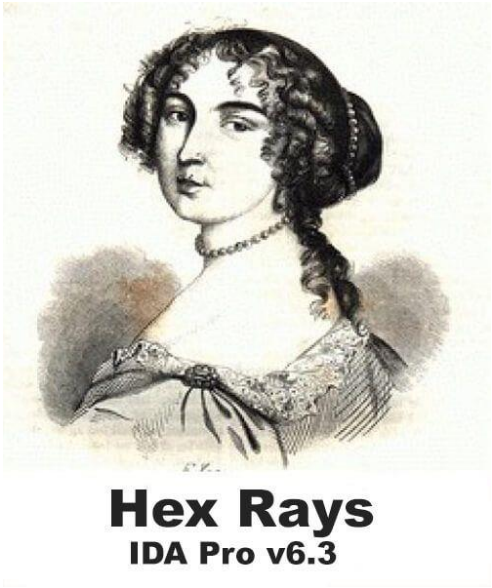
Beyond Compilation

IP theft

(duh)

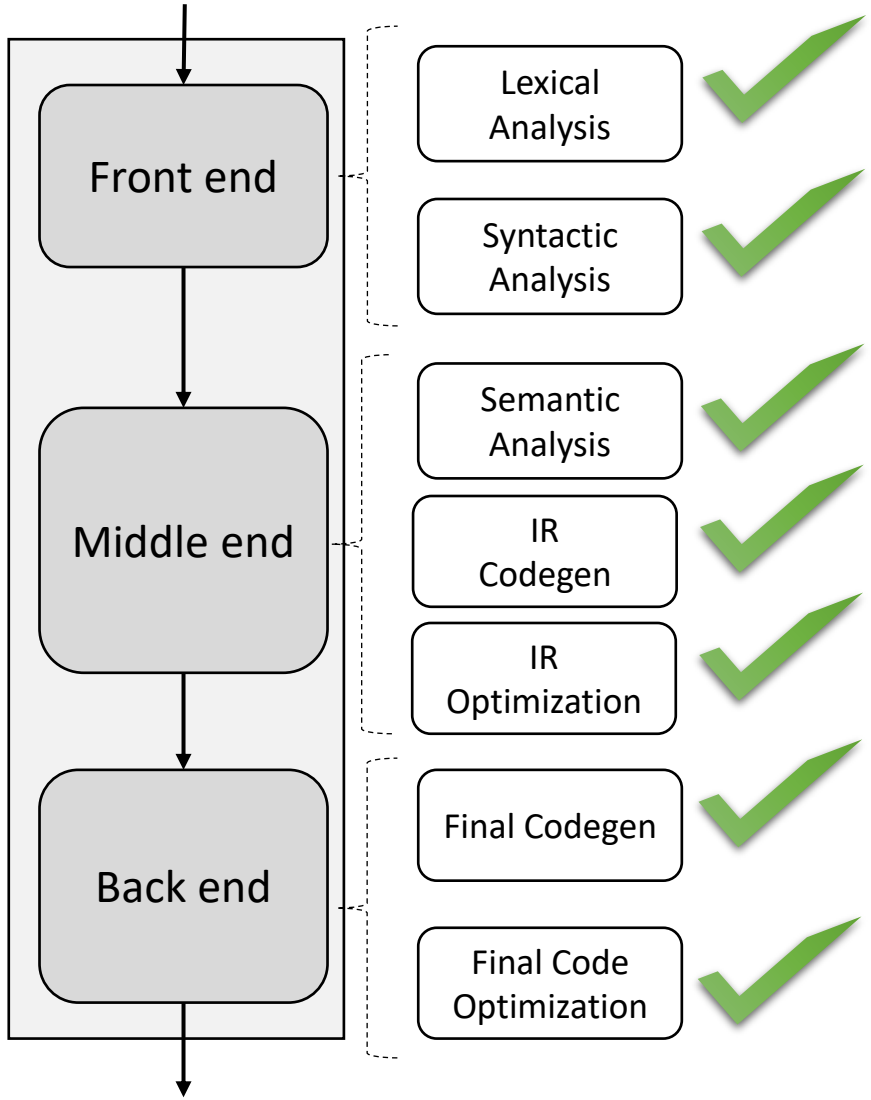
Ok, but what *legit* uses?

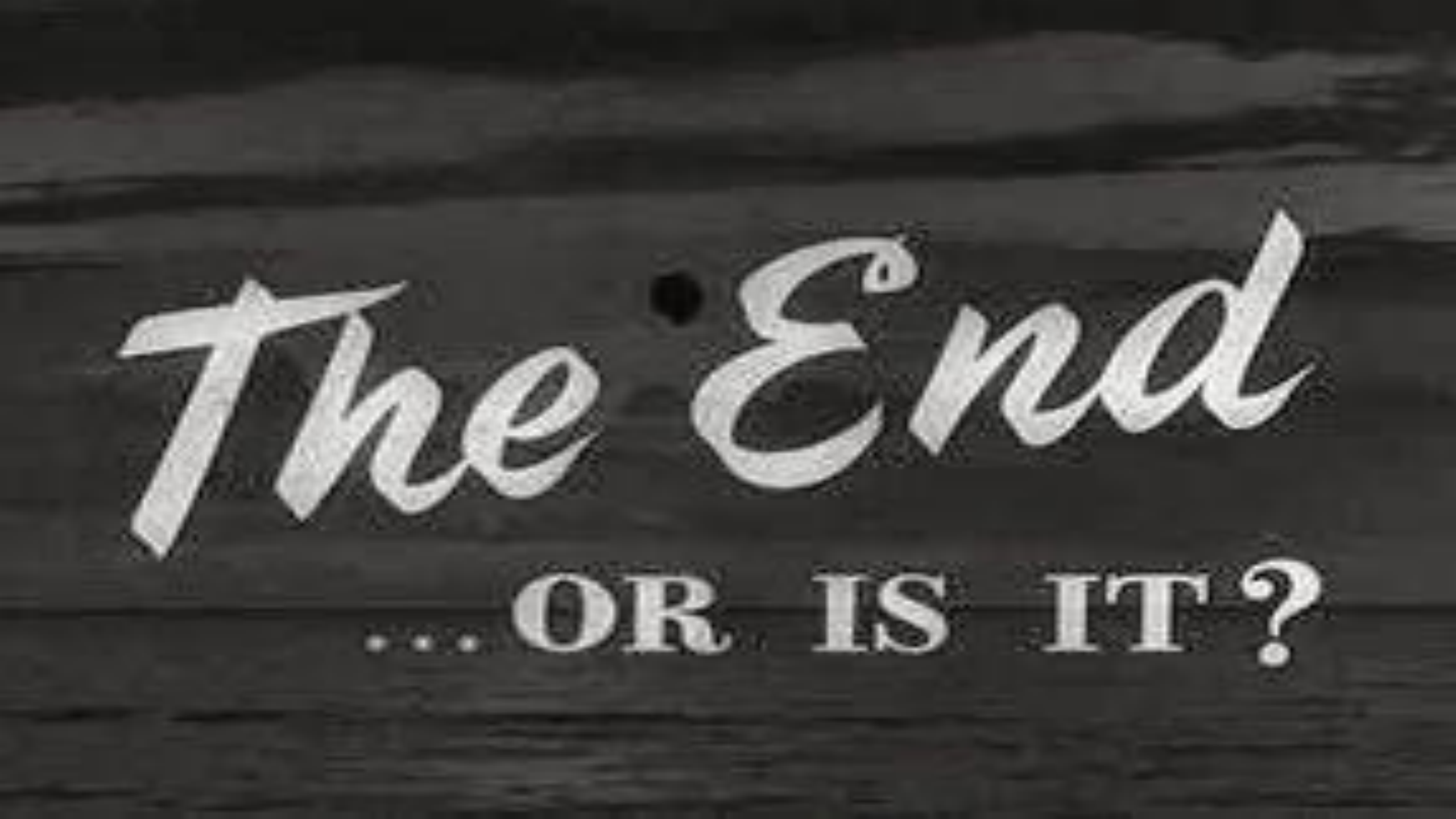
- Malware analysis
- Code “improvements”



Compiler Techniques Outside Compilers

Beyond Compilation




The End
....OR IS IT?

Where to go Next

Next Steps

Our compiler gives you a basic foundation:

- There's more to nearly every area of the course than what we covered
- Hopefully this course serves as a framework for refinement

What We Didn't Cover

Next Steps

Topics we glossed over:

- Object oriented code
- Many optimizations
- Compiling non-imperative code
  - Logic programming (e.g. datalog)
  - Functional programming (Haskell et al)

We glossed over some topics

Is this Stuff Useful?

Next Steps

From what we've covered:

- I bet you have could invent a programming language
- I hope you have a deeper understanding of how programs work
- I think you gained some tools in your toolbelt

Continuing the Journey

Beyond Compilation: Next Steps

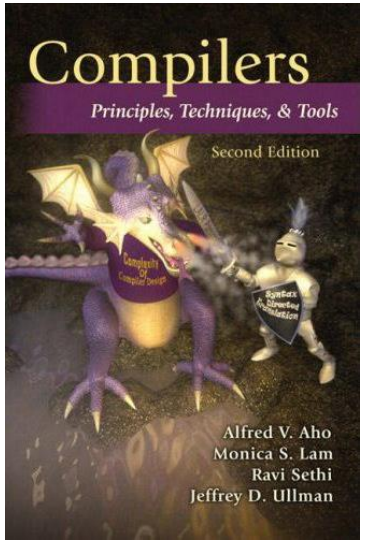
If you <u>like</u> compiler construction

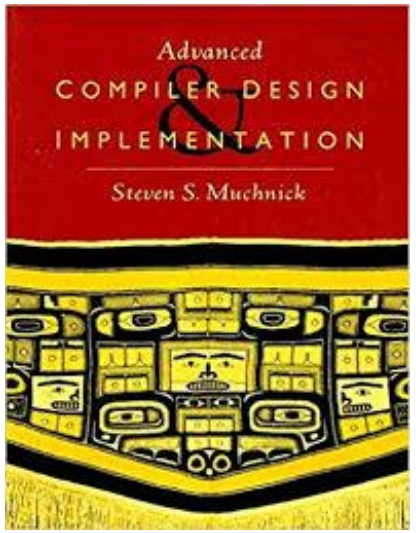
Classic compilers texts

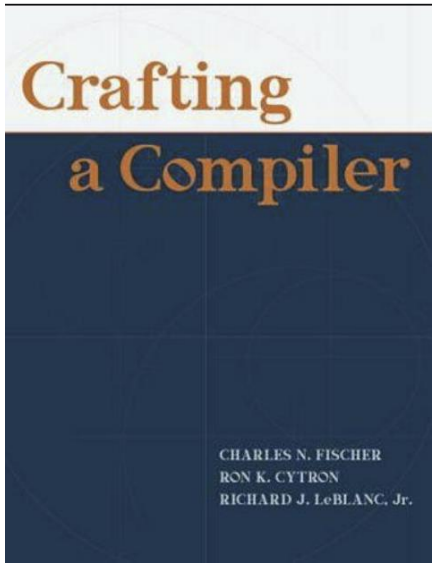
- Compilers: Principles, Techniques and Tools
  - Aho, Lam, Sethi, and Ullman
- Advanced Compiler Design & Implementation
  - Munchnick
- Crafting a Compiler
  - Fischer and Cytron

Cutting Edge Developments

- SIGPLAN Proceedings
- LLVM Docs







Continuing the Journey

Beyond Compilation: Next Steps

If you <u>like</u> compiler construction

Classic compilers texts

- Compilers: Principles, Techniques and Tools
  - Aho, Lam, Sethi, and Ullman
- Advanced Compiler Design & Implementation
  - Munchnick
- Crafting a Compiler
  - Fischer and Cytron

Cutting Edge Developments

- SIGPLAN Proceedings
- LLVM Docs

If you <u>dislike</u> compiler construction

Sorry ☹

Maybe the real Compilation…
was the friends we made along the way

Regular Languages

Lexical Analysis

Context-Free Grammars

Syntactic Analysis

Syntax-Directed Translation

Semantic Analysis

Intermediate Representations

Hardware Architecture

Machine Code

Optimization / Program Analysis

Unfinished Business

My Final Words

Grades:

- Project Grades forthcoming

Curve:

- Projects may be individually curved
- Minimal curve will be out in time for the final

Official Course Evaluations

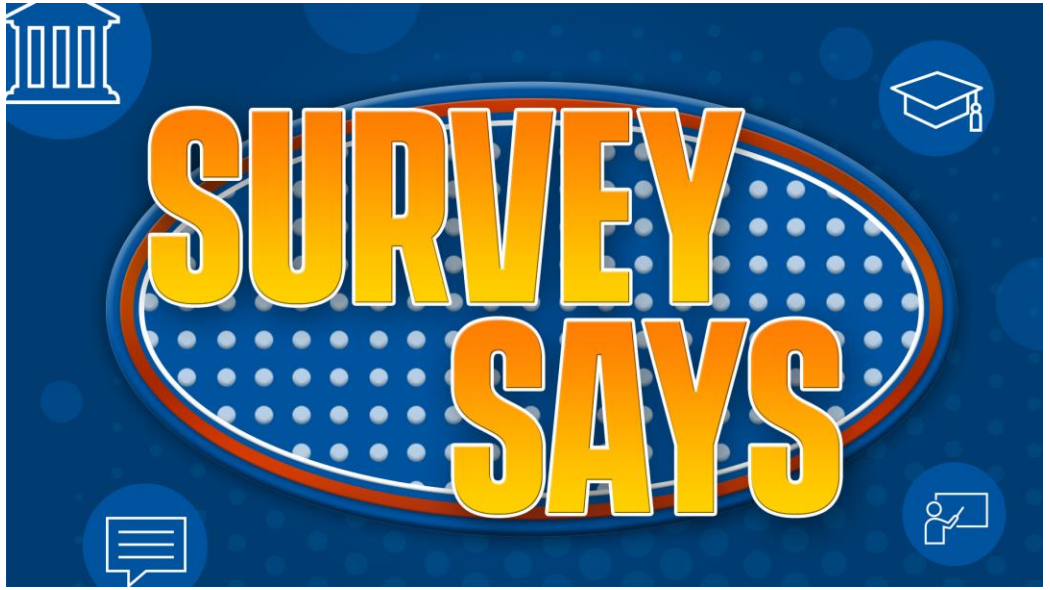
My Final Words

Your chance for a performance review

- I review these every semester
- The department chair reviews these every semester

Please do them!

- There are 2 surveys
  - University
  - Department



Unofficial Course Evaluations

My Final Words

https://compilers.cool/final-survey

Things you should know:

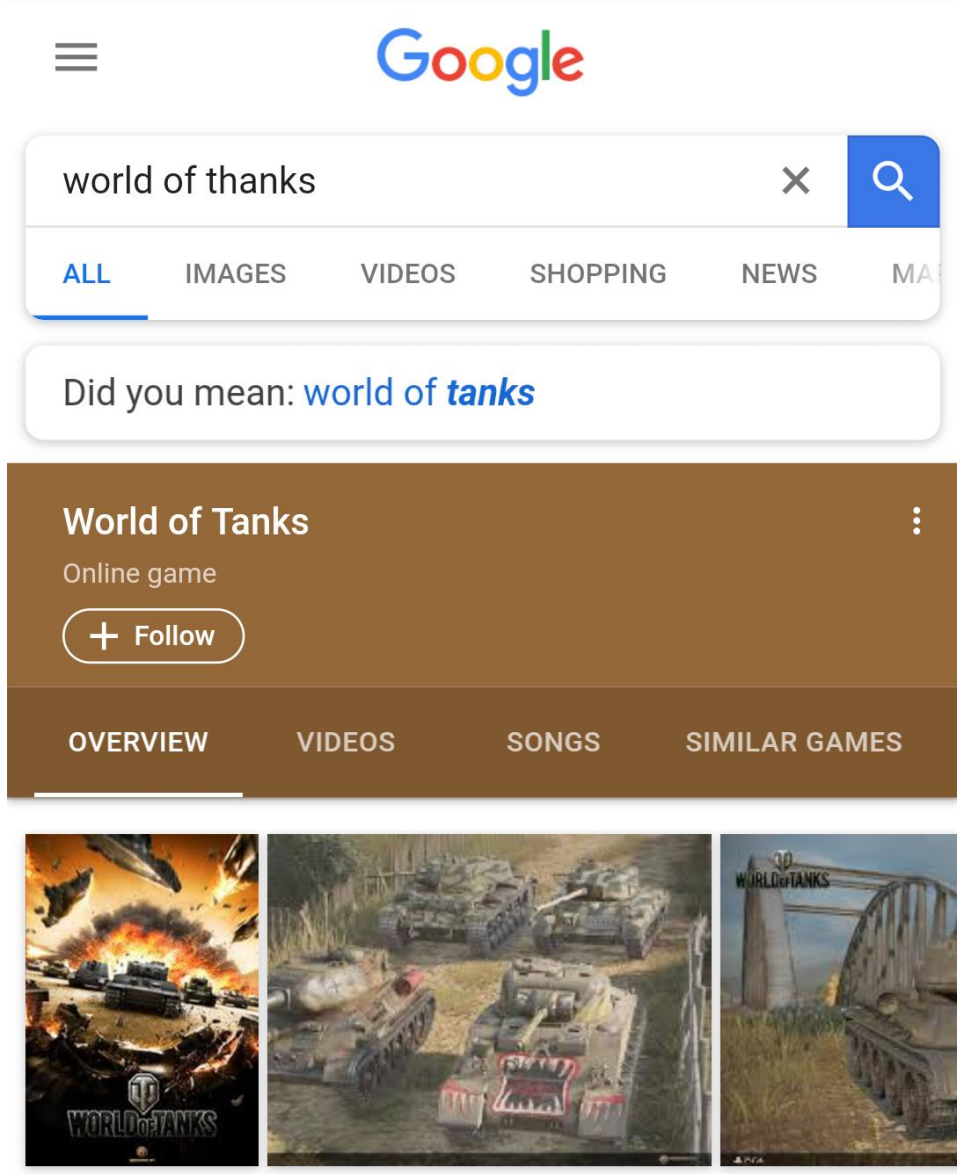
- Still completely anonymous
- I won’t look at the responses until after grades are assigned

Less about absolutes than improvement:

- Don’t worry about my feelings
- Just for me (though I might pull quotes for future semesters)

A World of Thanks!

- Special thanks to people who asked/answered questions in class or on Piazza
- Extra special thanks to people who suggested improvements and gave feedback



No Google. I did not mean "World of Tanks"