

# Week 14: Lecture A

## Configuration-aware Fuzzing & Course Wrap-up

Monday, April 14, 2025

# How are semester projects going?

Making progress?



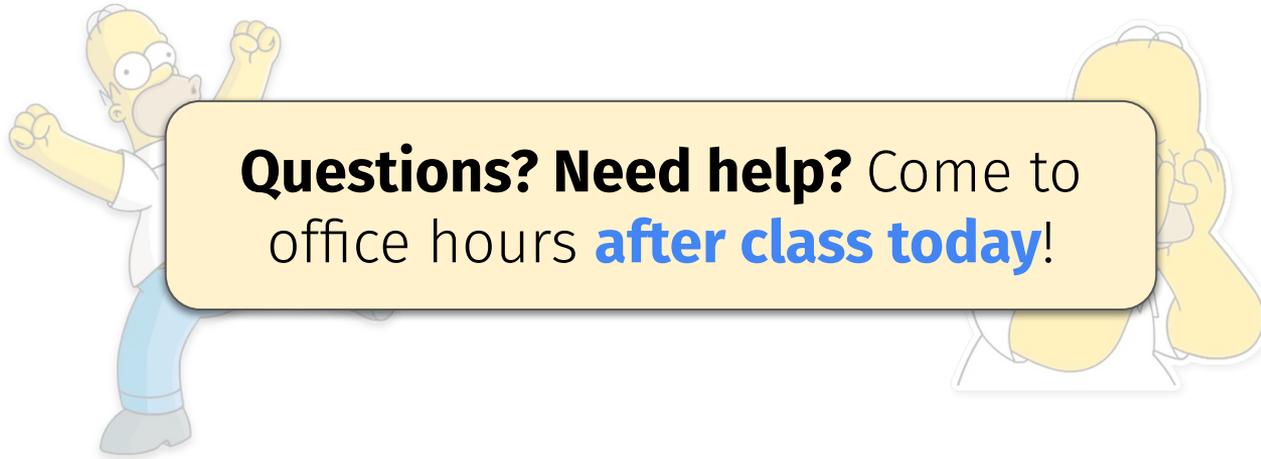
Stuck?



# How are semester projects going?

Making progress?

Stuck?



**Questions? Need help?** Come to  
office hours **after class today!**

# The Next Few Weeks

Part 3: New Frontiers in Fuzzing	
Monday Meeting	Wednesday Meeting
Mar. 31 <b>Kernel Fuzzing</b> ▶ Readings:	Apr. 02 <b>LLM-assisted Fuzzing</b> ▶ Readings:
Apr. 07 <b>Compiler Fuzzing</b> ▶ Readings:	Apr. 09 <b>Hardware Fuzzing</b> ▶ Readings:
Apr. 14 <b>Fuzzing Configurable Software</b> ▶ Readings:	Apr. 16 <b>▲ Final Presentations (Day 1)</b>
Apr. 21 <b>▲ Final Presentations (Day 2)</b> <b>▲ Final Reports due Tuesday by 11:59pm via Canvas</b>	Apr. 23 <b>No Class (Reading Day)</b>

# Recap: Project Schedule

- **Apr. 16th & 21st:** final presentations
  - **5–8 minute** slide deck and discussion
  - What you did, and why, and what results
  - **Report any bugs found** (and show you did so!)
- What's most important:
  - High-level technique
  - Challenges and workarounds
  - Key results (bugs found, other successes, etc.)
- **Project report** due by midnight last day of class
  - 3–5 pages describing your work and results
  - Reports of any bugs found



# Recap: Project Schedule

## Final Project (collected via Canvas)

**Instructions:** Using your skills from Labs 1–3, team-up in groups of **two to four students** to hunt bugs in a **real-world software** of your choice! Upon choosing a target, your team must figure out how to (1) harness it, (2) fuzz it, and (3) triage any discovered bugs. You may select any target you like (e.g., popular APIs, video games, kernels), **provided it has *not* been fuzzed before—or has demonstrably not been fuzzed effectively**. Failure to do your due diligence will cause your team to lose points—or worse yet—have to find a different target!

Halfway through the semester, your team will present a **5-minute project lightning proposal** to the class outlining your chosen target, your proposed approach, and the significance of your work. At the semester's end, you will prepare and deliver a **final in-class presentation** along with submitting a **final project report** outlining your ultimate approach, findings, and any discovered bugs.

**Heilmeier's Catechism** will serve as the high-level rubric for your proposal, presentation, and report—so be ready to answer *why* your project idea matters! But most importantly, **get creative and have fun**. Besides Heilmeier's Catechism, other important criterion include:

- **Responsible Disclosure:** Discovered bugs *must* be disclosed to developers! Include any bug report links (e.g., GitHub issues) in your final report. If bug reports are not public, you may include screenshots of your correspondence (e.g., emails) with developers.

**Final Reports** (3–5 pages) due by **11:59PM** on **Tuesday, April 21st**

**Merge** your Final Reports and Presentation Slides **into one PDF**

# Finalized Team Presentation Schedule

## Wednesday

<b>Project Title</b>	Alter Domus
Team Members	Braden Campbell, Chandler Turner
<b>Project Title</b>	Fuzzing <a href="#">Draw.io</a>
Team Members	Davit Zatikyan, Wilker Gonzalez
<b>Project Title</b>	Juceing
Team Members	Pablo Arancibia-Bazan, Alec Carton, Sean McGuirk
<b>Project Title</b>	Fuzzing PDFio
Team Members	Adwait Shinganwade, Brensen Villegas
<b>Project Title</b>	Fuzzing Blender
Team Members	Austin Garcia, Hyunwoo Lee
<b>Project Title</b>	Fuzzing MelonDS
Team Members	Jie Lin, Alexa Fresz, Corinna Healey, Leo Ramirez
<b>Project Title</b>	Fuzzing Polybar
Team Members	Kalyan Shankar Ragam, Praneeth Chavva, Koumudi Raju Kothapally, Aparna Gudivada
<b>Project Title</b>	Fuzzing Vulkan
Team Members	Ethan Collier, Henry Zheng
<b>Project Title</b>	Fuzzing Thinger.io
Team Members	Gavin Dibble, James Hart

## Monday

<b>Project Title</b>	Fuzzing Resume-Matcher
Team Members	Sunithya Penumarthy,Rahul Mandava,Harshita Samala,Leela Sowmya Jandhyala
<b>Project Title</b>	Fuzzing Libqrencode
Team Members	Vikas Kommalapati, Pranav Madhav, Kishore Kumar Kampalli, Anjali Kampalli
<b>Project Title</b>	Fuzzing llama-cpp
Team Members	Hao Ren, ChenCheng Mao, Ruiyang Xia
<b>Project Title</b>	Fuzzing Eclipse Paho C client library
Team Members	Nazmus Shakib Sayom, Jainta Paul
<b>Project Title</b>	Fuzzing GUI Applications
Team Members	Tanner Rowlett
<b>Project Title</b>	Fuzzing Stormlib
Team Members	Tinh Nguyen, Leo Leano, Austin Li, Vasil Vassilev
<b>Project Title</b>	Fuzzing Open5G
Team Members	Axe Tang
<b>Project Title</b>	Trying to improve fuzzer coverage on Tesseract
Team Members	Yash Lele, Rutuja Bhirud, Duncan Gilbreath
<b>Project Title</b>	Fuzzing on struct_mapping library
Team Members	Navya Dommalapati, Savant Mullapudi, Rajesh Vempalla, Shivahari Gundeti

# End-of-semester Course Evals

- **I want your feedback!**
  - 3rd time teaching this course 😊
  - **Help me improve the class!**
- Due by **May 6th**
  - <https://scf.utah.edu>
  - **Please please please!**



# End-of-semester Course Evals

- I want your feedback!

- 3rd time teaching this course 😊
- Help me improve

- Due by **May 6**

- <https://scf.utah.edu>
- Please please

**If 85% of class (42 of 49 students)** submits an eval, we'll add **2% extra credit** to your Participation grades!  
(equivalent to one lecture's worth of points!)

**HELP ME HELP YOU**

# Questions?



# Config-aware Fuzzing

# Software Variability: a Tale of Two Perspectives

- Variability at **run-time**:
  - Different ways of parsing input
  - Current focus of fuzzing research

```
readelf: Warning: Nothing to do.
Usage: readelf <option(s)> elf-file(s)
Display information about the contents of ELF format files
Options are:
-a --all                Equivalent to: -h -l -S -s -r -d -V -A -I
-h --file-header        Display the ELF file header
-l --program-headers    Display the program headers
  --segments            An alias for --program-headers
-S --section-headers    Display the sections' header
  --sections            An alias for --section-headers
-g --section-groups     Display the section groups
-t --section-details    Display the section details
-e --headers            Equivalent to: -h -l -S
-s --syms               Display the symbol table
  --symbols             An alias for --syms
  --dyn-syms            Display the dynamic symbol table
  --lto-syms            Display LTO symbol tables
  --sym-base=[0|8|10|16] Force base for symbol sizes. The options are
                        mixed (the default), octal, decimal, hexadecimal.
```

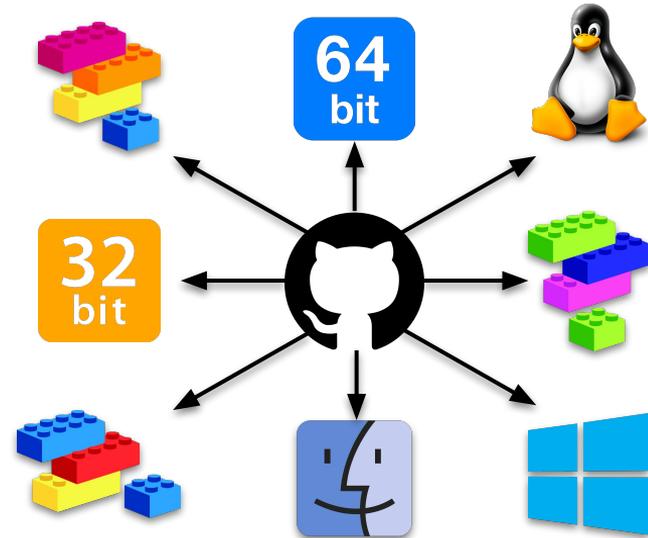
# Software Variability: a Tale of Two Perspectives

- Variability at **run-time**:
  - Different ways of parsing input
  - Current focus of fuzzing research
- Variability at **compile-time**:
  - Including & excluding certain code
  - Potentially huge attack surface
  - Not currently being explored

```
readelf: Warning: Nothing to do.
Usage: readelf <option(s)> elf-file(s)
Display information about the contents of ELF format files
Options are:
-a --all                Equivalent to: -h -l -S -s -r -d -V -A -I
-h --file-header        Display the ELF file header
-l --program-headers   Display the program headers
--segments             An alias for --program-headers
-S --section-headers   Display the sections' header
--sections             An alias for --section-headers
-g --section-groups     Display the section groups
-t --section-details   Display the section details
-e --headers           Equivalent to: -h -l -S
-s --syms              Display the symbol table
--symbols              An alias for --syms
--dyn-syms             Display the dynamic symbol table
--lto-syms             Display LTO symbol tables
--sym-base=[0|8|10|16] Force base for symbol sizes. The options are
mixed (the default), octal, decimal, hexadecimal.
```

# Compile-time Variability

- Maintaining **deployment-specific** codebases is unscalable
  - Support on-demand features, environments, architectures
- **Solution:** software variability
  - One codebase, multiple builds
- Mechanisms for variability:
  - **C and C++:** the preprocessor
  - **Rust:** conditional compilation



# Compile-time Variability Bugs

- Bugs triggerable only within a **specific variant** of the software

```
#ifdef TWL4030_CORE
int twl_probe()
{
    int *ops = NULL; ←
    #ifdef OF_IRQ
        ops = &irq_domain_simple_ops;
    #endif
    int irq = *ops; ←
}
#endif
```

<http://vbdb.itu.dk/linux/6252547.html>

## Can you spot the bug?

If TWL4030\_CORE and !OF\_IRQ,  
then int \*ops remains NULL...

= **NULL pointer dereference!**

With **thousands to millions** of possible variants, **concurrent testing** becomes **unscalable!**

# Compile-time Variability Bugs

- <http://vbdb.itu.dk/database.html>

## 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis

Iago Abal  
iago@itu.dk

Claus Brabrand  
brabrand@itu.dk

Andrzej Wąsowski  
wasowski@itu.dk

IT University of Copenhagen  
Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

### ABSTRACT

Feature-sensitive verification pursues effective analysis of the exponentially many variants of a program family. However, researchers lack examples of concrete bugs induced by variability, occurring in real large-scale systems. Such a collection of bugs is a requirement for goal-oriented research, serving to evaluate tool implementations of feature-sensitive analyses by testing them on real bugs. We present a qualitative study of 42 variability bugs collected from bug-fixing commits to the Linux kernel repository. We analyze each of the bugs, and record the results in a database. In addition, we provide self-contained simplified C99 versions of the bugs, facilitating understanding and tool evaluation. Our study provides insights into the nature and occurrence of variability bugs in a large C software system, and shows in what ways variability affects and increases the complexity of software bugs.

Features in a configurable system interact in non-trivial ways, in order to influence each others functionality. When such interactions are unintended, they induce bugs that manifest themselves in certain configurations but not in others, or that manifest differently in different configurations. A bug in an individual configuration may be found by analyzers based on standard program analysis techniques. However, since the number of configurations is exponential in the number of features, it is not feasible to analyze each configuration separately.

Family-based [33] analyses, a form of feature-sensitive analyses, tackle this problem by considering all configurable program variants as a single unit of analysis, instead of analyzing the individual variants separately. In order to avoid duplication of effort, common parts are analyzed once and the analysis forks only at differences between variants. Re-

# Is anyone fuzzing for compile-time variability bugs?

## ■ OSS-Fuzz

- x86
- x64
- ARM (maybe?)

```
export CFLAGS="$CFLAGS -DSQLITE_MAX_LENGTH=128000000 \  
-DSQLITE_MAX_SQL_LENGTH=128000000 \  
-DSQLITE_MAX_MEMORY=25000000 \  
-DSQLITE_PRINTF_PRECISION_LIMIT=1048576 \  
-DSQLITE_DEBUG=1 \  
-DSQLITE_MAX_PAGE_COUNT=16384"
```

## ■ SyzKaller

- Many kernels
- Default configs only

stable-6.1-arm64-kasan-base.config

stable-6.1-arm64-kasan.config

stable-6.1-kasan-base.config

stable-6.1-kasan.config

# Is anyone fuzzing for compile-time variability bugs?

## ■ OSS-Fuzz

- x86
- x64
- ARM (maybe?)

```
export CFLAGS="$CFLAGS -DSQLITE_MAX_LENGTH=128000000 \  
-DSQLITE_MAX_SQL_LENGTH=128000000 \  
-DSQLITE_MAX_MEMORY=25000000 \  
-DSQLITE_PRINTF_PRECISION_LIMIT=1048576 \  
-DSQLITE_DEBUG=1 \  
-DSQLITE_MAX_PAGE_COUNT=16384"
```

## ■ SyzKaller

- Many kernels
- Default configs only

## ■ An **under-explored** class of bugs

- We need tools to find them!

stable-6.1-arm64-kasan-base.config

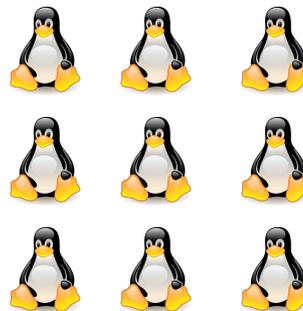
stable-6.1-arm64-kasan.config

stable-6.1-kasan-base.config

stable-6.1-kasan.config

# What would compile-time variability fuzzing look like?

- **Crude** approach:
  - Enumerate every config possible
  - Concurrently fuzz all the builds
  - Differential execution



# What would compile-time variability fuzzing look like?

- **Crude** approach:
  - Enumerate every config possible
  - Concurrently fuzz all the builds
  - Differential execution
- **Problem:** combinatorial explosion
  - Good luck building every config
  - Good luck fuzzing every build



Static Analysis of Variability in System Software: The 90,000 `#ifdefs` Issue\*

System software can be configured at compile time to tailor it with respect to a broad range of supported hardware architectures and application domains. The Linux v3.2 kernel, for instance, provides more than 12,000 configurable features, which control the configuration-dependent inclusion of 31,000 source files with 89,000 `#ifdef` blocks.



Linux provides more than  
**12,000** configurable features

# Toward Compile-time Variability-Aware Fuzzing

- **Idea:** transform **conditionally-compiled** code into **conditionally-invoked**

```
#ifdef TWL4030_CORE
int twl_probe()
{
    int *ops = NULL;

#ifdef OF_IRQ
    ops = &irq_domain_simple_ops;
#endif

    int irq = *ops;
}
#endif
```

Prior work in the SE community on  
“desugaring” preprocessor usage:

## SugarC: Scalable Desugaring of Real-World Preprocessor Usage into Pure C

Zachary Patterson  
The University of Texas at  
USA  
Zach.Patterson@utdallas.edu

### ABSTRACT

Variability-aware analysis is critical for analyzing real-world configurable C software. An important component of variability-aware analysis is real-world C software that uses both C code, by replacing the preprocessor with a runtime-variability. In this work, we present SugarC, a new desugaring tool. SugarC, that transforms real-world C code into pure C code with translation rules, performs static analysis, and introduces numerous challenges that appear in real-world programs. We present our experience with SugarC, and show that SugarC features more than two existing desugaring tools for real-world configurable C software.

## Effective Analysis of C Programs by Rewriting Variability

Alexandru F. Iosif-Lazar<sup>1</sup>, Jean Melo<sup>2</sup>, Alin Brabrand<sup>3</sup>, and Andrzej Wąsowski<sup>4</sup>  
<sup>1</sup> a IT University of Copenhagen, Denmark

**Abstract** Context. Variability-intensive programs (programs with many options (features) on and off, while reuse of the code). Inquiry. Verification of program families is challenging because of the number of features. Existing single-program analysis and program families, and designing and implementing the correct laborious.

**Approach.** In this work, we propose a range of variability-aware analysis tools into single programs by replacing compile-time variability. The obtained transformed programs can be subsequently analyzed by program analysis tools such as type checkers, symbolic execution, and model checking. Our variability-related transformations are effective between the outcomes in the transformed single program from the original program family is *equivalent*. **Grounding.** We present our transformation rules and the imperative language IMP. Then, we discuss our experience with an efficient and effective analysis and verification of real-world programs. We report some interesting variability-related analysis results. **Keywords.** Single-program C verification tools, such as FuzzC-ME, C

## Variability Encoding: From Compile-Time to Load-Time Variability

Alexander von Rhein<sup>1\*</sup>, Thomas Thüm<sup>1</sup>, Ina Schaefer<sup>2</sup>, Jörg Liebig<sup>3</sup>, Sven Apel<sup>1\*</sup>

<sup>1</sup>University of Passau, Innstraße 33, D-94032, Germany  
<sup>2</sup>University of Magdeburg, P.O. Box 4120, D-39016, Germany  
<sup>3</sup>TEC Braunschweig, Mühlentorstraße 23, D-38106, Germany

### Abstract

Many software systems today are configurable. Analyzing configurable systems is challenging, especially as (1) the number of system variants may grow exponentially with the number of configuration options, and (2) often existing analysis tools cannot be used for configurable systems. Recent work proposes to automatically transform compile-time variability into load-time variability—called *variability encoding*—with the goal of reusing existing analysis tools for analyzing configurable systems and improving analysis performance compared to analyzing all system variants in a brute-force manner. However, it is not clear whether one can automatically find an efficiently analyzable load-time configurable system for any given compile-time configurable system. Also, for many analyses, we need guarantees that the load-time configurable system precisely encodes the behavior of all system variants that can be statically derived. We address both issues (1) by developing a formal model of variability encoding based

# Toward Compile-time Variability-Aware Fuzzing

- **Idea:** transform **conditionally-compiled** code into **conditionally-invoked**

```
#ifdef TWL4030_CORE
int twl_probe()
{
    int *ops = NULL;

#ifdef OF_IRQ
    ops = &irq_domain_simple_ops;
#endif

    int irq = *ops;
}
#endif
```

Prior work in the SE community on  
“desugaring” preprocessor usage:

SugarC: Scalable Desugaring of Real-World Preprocessor Usage

Zachary Patten  
The University of Utah  
USA  
Zach.Patten@utah.edu

Is **desugaring** amenable  
to **dynamic testing**?

ABSTRACT

Variability-aware analysis of real-world configurable C software. An important component of variability-aware analysis is the desugaring of real-world C software that uses both compile-time and runtime variability, by replacing the preprocessor with translation rules, performing static desugaring, and introducing no runtime variability. In this work, we present SugarC, a benchmarked desugaring tool, that transforms real-world C code with preprocessor usage into single programs by replacing compile-time variability with translation rules, performing static desugaring, and introducing no runtime variability. In this work, we present SugarC, a benchmarked desugaring tool, that transforms real-world C code with preprocessor usage into single programs by replacing compile-time variability with translation rules, performing static desugaring, and introducing no runtime variability. In this work, we present SugarC, a benchmarked desugaring tool, that transforms real-world C code with preprocessor usage into single programs by replacing compile-time variability with translation rules, performing static desugaring, and introducing no runtime variability.

variable options (features) on and off, while reuse of the code. Inquiry. Verification of program families is challenging because of the number of features. Existing single-program analysis and verification tools are not designed for program families, and designing and implementing the correct analysis tools is laborious.

Approach. In this work, we propose a range of variability-aware analysis tools that transform real-world C code with preprocessor usage into single programs by replacing compile-time variability with translation rules, performing static desugaring, and introducing no runtime variability. The obtained transformed programs can be subsequently analyzed by existing program analysis tools such as type checkers, symbolic execution engines, and model checkers. Our variability-related transformations are grounded in the outcomes in the transformed single program from the original program family is equally. Grounding. We present our transformation rules and the imperative language IMP. Then, we discuss our experience with efficient and effective analysis and verification of real-world programs. We report some interesting variability-related analysis results. We compare our analysis results with existing single-program C verification tools, such as FRAMA-C, and

\*University of Passau, Innsbrg 33, D-94032, Germany  
\*University of Magdeburg, P.O. Box 4120, D-39016, Germany  
\*TU Braunschweig, Mühlenpforderstraße 23, D-38106, Germany

Abstract

Many software systems today are configurable. Analyzing configurable systems is challenging, especially as (1) the number of system variants may grow exponentially with the number of configuration options, and (2) often existing analysis tools cannot be used for configurable systems. Recent work proposes to automatically transform compile-time variability into load-time variability—called *variability encoding*—with the goal of reusing existing analysis tools for analyzing configurable systems and improving analysis performance compared to analyzing all system variants in a brute-force manner. However, it is not clear whether one can automatically find an efficiently analyzable load-time configurable system for any given compile-time configurable system. Also, for many analyses, we need guarantees that the load-time configurable system precisely encodes the behavior of all system variants that can be statically derived. We address both issues (1) by developing a formal model of variability encoding based

# Toward Compile-time Variability-Aware Fuzzing

- **Idea:** transform **conditionally-compiled** code into **conditionally-invoked**

```
#ifdef TWL4030_CORE
int twl_probe()
{
    int *ops = NULL;

#ifdef OF_IRQ
    ops = &irq_domain_simple_ops;
#endif

    int irq = *ops;
}
#endif
```

Parse the AST  
to identify all  
preprocessor  
tokens



Transform all  
preprocessor  
code blocks

```
if (getenv("TWL4030_CORE")){
int twl_probe()
{
    int *ops = NULL;

if (getenv("OF_IRQ")){
    ops = &irq_domain_simple_ops;
}

    int irq = *ops;
}
}
```

# Toward Compile-time Variability-Aware Fuzzing

- **Idea:** transform **conditionally-compiled** code into **conditionally-invoked**

```
if (getenv("TWL4030_CORE")){
int twl_probe()
{
    int *ops = NULL;

    if (getenv("OF_IRQ")){
        ops = &irq_domain_simple_ops;
    }

    int irq = *ops;
}
}
```

- Use fuzzing to **differentially test** different feature combinations

TWL4030\_CORE && OF\_IRQ  
!TWL4030\_CORE && !OF\_IRQ  
!TWL4030\_CORE && OF\_IRQ  
TWL4030\_CORE && !OF\_IRQ



# Toward Compile-time Variability-Aware Fuzzing

- Idea: transform **conditionally-compiled** code into **conditionally-invoked**

```
if (getenv("TWL4030_CORE"))  
int twl_pr  
{  
    int *ops  
  
    if (getenv  
        ops = &irq_domain_simple_ops;  
    }  
  
    int irq = *ops;  
}  
}
```

**Goal:** enable a **software product line** to be expressed and fuzzed **via a single executable**

- Use fuzzing to **differentially test**

tions

```
!TWL4030_CORE && !OF_IRQ  
!TWL4030_CORE && OF_IRQ  
TWL4030_CORE && !OF_IRQ
```



# Current Work: Variability Desugaring

## ■ Current support for:

- `#ifdef`
- Function decls/defs
- Variable decls/defs
- Nested macros

## ■ Working on:

- Other macro types
- Duplicate-named vars/funcs
- Other non-trivial cases

```
#ifdef F00
void foo(){
    printf("foo");
}
#endif
```

```
int main(){
#ifdef F00
int y = 2;
x = y;
foo();
#endif
}
```

```
void foo(){
    assert("F00");
    printf("foo");
}
```

```
int main(){
    int y;
    if ("F00"){
        y=2;
        x=y;
        foo();
    }
}
```

# Current Work: Differential Execution

- **Initial approach:** replaying inputs
  - Gather high-coverage inputs
  - Replay each with instrumentation
  - Randomize enabled/disabled macros
  - Filter-out invalid configs pre/post fuzzing
  - Use standard bug oracles (e.g., ASAN)
- **Smarter approach:** track variability
  - Need to identify metrics of “interesting”
  - Prioritize configs that execute new paths
  - Prioritize configs that change program state

# Future Directions

- **AFL + SyzKaller** implementations
  - Prototype a many-build approach
  - Eventually pair with desugaring
- **Directed** Variability Fuzzing
  - Pick subset of features to test
  - Constrain to specific path/region
- **Cross-platform** Variability
  - Test Windows and Linux builds
  - Explore other architectures

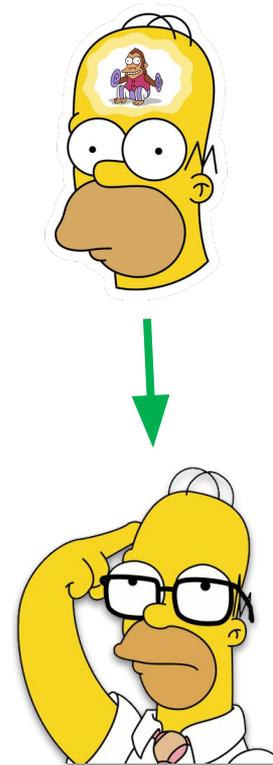
# Course Wrap-up

# You've finished the course!



# What did we learn?

- **Weeks 1 – 3: Systems Research 101**
  - Ideas, writing, presenting, reviewing
- **Weeks 4 – 9: Fuzzing Fundamentals**
  - Generation, feedback, bugs & triage, harnessing, roadblocks, fuzzing science
- **Weeks 10 – 12: Emergent Enhancements**
  - Optimization, directed fuzzing, hybrid fuzzing
- **Weeks 13 – 16: New Frontiers in Fuzzing**
  - Kernels, compilers, hardware, and more!



# Goal #1: become better researchers

Part 1: Course Introduction	
Tuesday Meeting	Thursday Meeting
Aug. 23 <b>Course Introduction</b> ( <a href="#">slides</a> )	Aug. 25 <b>Research 101: Ideas</b> (s
Aug. 30 <b>Research 101: Writing</b> ( <a href="#">slides</a> )	Sep. 1 <b>Research 101: Review</b> (and a guest lecture by
Sep. 6 <b>Research 101: Presenting</b> ( <a href="#">slides</a> ) <b>Select papers to present by 11:59pm</b>	Sep. 8 <b>Fuzzing Introduction</b> (s <b>Beginner Fuzzing Lab</b>



## The Heilmeier Catechism

- What are you trying to do?
- Articulate objectives using absolutely no jargon.
- What are the limits of current practice?
- How and why do you think it will be successful?
- If successful, what difference will it make?
- What are the potential "exams" to check for success?



**Feel free to bookmark or download the Research 101 slides!**

[cs.utah.edu/~snagy/courses/cs5963/slides](https://cs.utah.edu/~snagy/courses/cs5963/slides)

# Goal #2: exposure to different perspectives



### Fuzzing Hardware Like Software

Timothy Trippel<sup>1</sup>, Kang G. Shin<sup>1</sup>  
*Computer Science & Engineering  
 University of Michigan  
 Ann Arbor, MI  
 (trippel,kashin)@umich.edu*

Alex Chernyakovsky,  
 Garret Kelly, Dominic Rizzo  
*OpenFlare  
 Google, LLC  
 Cambridge, MA  
 (achernya,gar,domrizzo)@google.com*

Matthew Hicks  
*Computer Science  
 Virginia Tech  
 Blacksburg, VA  
 mhicks2@vt.edu*

**ABSTRACT**  
 Hardware flaws are persistent and often patchable once fabricated, and any flawed formally verified software executing on verification time dominates implementation random testing, due to its scalability to cover, given its undetected nature, this type of errors. Instead of making incremental improvements, hardware verification approaches...

### Novelty Not Found: Fuzzer Restarts to Improve Input Space Coverage (Registered Report)

Xinyi Xu  
*CISPA  
 Germany  
 xinyixu@cispa.de*

Moritz Schloegel  
*CISPA  
 Germany  
 moritz.schloegel@cispa.de*

Lukas Bernhard  
*CISPA  
 Germany  
 lukas.bernhard@cispa.de*

Thorsten Holz  
*CISPA  
 Germany  
 holz@cispa.de*

**KEYWORDS**  
 fuzzing, coverage, restarts, input space, novelty

### A Comprehensive Study of Fuzzing (Registered Report)

Thuan Pham  
*phamthuan@unimelb.edu.au  
 The University of Melbourne  
 Parkville, Australia*

Dongge Liu  
*dongge.liu@google.com  
 Sydney, Australia*

Benjamin I.P. Robinson  
*benjamin.robinson@unimelb.edu.au  
 The University of Melbourne  
 Parkville, Australia*

Y. Murray  
*murray@unimelb.edu.au  
 The University of Melbourne  
 Parkville, Australia*

**ABSTRACT**  
 vulnerability discovery. Popular CGF fuzzer such as LAFuzz [1], AFL++ [1, 2], and Honggfuzz [3] have discovered thousands of vulnerabilities in large real-world systems [1, 2, 3]. The state-of-the-art in fuzzing has seen significant advancements, with hundreds of research papers and dozens of tools being published to improve the techniques in various aspects [4]. These efforts have focused on enhancing fuzzing in areas such as feedback collection [16, 18, 20, 26], corpus management [19], seed selection algorithms [21, 23], input generation algorithms [15, 20, 16, 18, 27], and so on. Additionally, researchers have attempted to extend the applicability of fuzzing to challenging target programs such as network protocols [16, 43], database systems [16, 50], IoT devices [46], complex [21] device drivers [1], and heterogeneous applications [44]. Another noteworthy research direction is parallel or distributed fuzzing [22, 28, 29], which aims to improve fuzzing efficiency by leveraging multiple CPUs to perform fuzzing.

### AURORA: Statistical Crash Analysis for Automated Root Cause Explanation

Tim Blazytko, Moritz Schloegel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner and Thorsten Holz  
*Ruhr-Universität Bochum, Germany*

**ABSTRACT**  
 Given the huge success of automated software testing techniques, a large amount of crashes is found in practice. Identifying the root cause of a crash is a time-intensive endeavor, causing a disproportion between finding a crash and fixing the underlying software fault. To address this problem, various approaches have been proposed that rely on techniques such as reverse execution and backward taint analysis. Still, these techniques are either limited to certain fault types or provide an analyst with assembly instructions, but no context information or explanation of the underlying fault. In this paper, we propose an automated analysis approach that does not only identify the root cause of a given crashing input for a binary executable, but also provides the analysis results in a human-readable format. We identify a novel insight that seems reasonable to conjecture that the fuzzer which is better in terms of code coverage is also better in terms of bug finding—but is this really true? In Figure 1, we show the coverage achieved and the average number of bugs found for each benchmark. The ranks are visibly different. To be sure, we also conducted a pair-wise comparison between any two fuzzers where the difference in coverage and the difference in bug finding are statistically significant. The results are similar. We identify a strong agreement on the superiority or ranking of a fuzzer when compared in terms of mean coverage versus mean bug finding. Inter-rater agreement assesses the degree to which

### Bloat Accelerating Binary-only Fuzzing (Registered Report)

Anh Nguyen-Tuong  
*University of Virginia  
 Charlottesville, Virginia  
 anhnguy@virginia.edu*

Matthew Hicks  
*Virginia Tech  
 Blacksburg, Virginia  
 mhicks@vt.edu*

Jason D. Hiser  
*University of Virginia  
 Charlottesville, Virginia  
 hiser@virginia.edu*

**ABSTRACT**  
 Coverage-guided fuzzer's aggressive, high-volume testing has helped reveal tens of thousands of software security flaws. While executing billions of test cases translates fast code coverage testing, the nature of binary-only targets leads to reduced testing performance. A recent advancement in binary fuzzing performance is Coverage-guided Tracing (CGT), which brings orders of magnitude gain in throughput by restricting the scope of coverage tracing to only when new coverage is guaranteed. Unfortunately, CGT is not only a basic block coverage granularity—it does not feature requisite fine-grain coverage metrics (edge coverage and hit counts). It is this limitation which prohibits nearly all of today's state-of-the-art fuzzers from attaining the performance benefits of CGT. This paper tackles the challenge of enhancing CGT by extending

### 1 INTRODUCTION

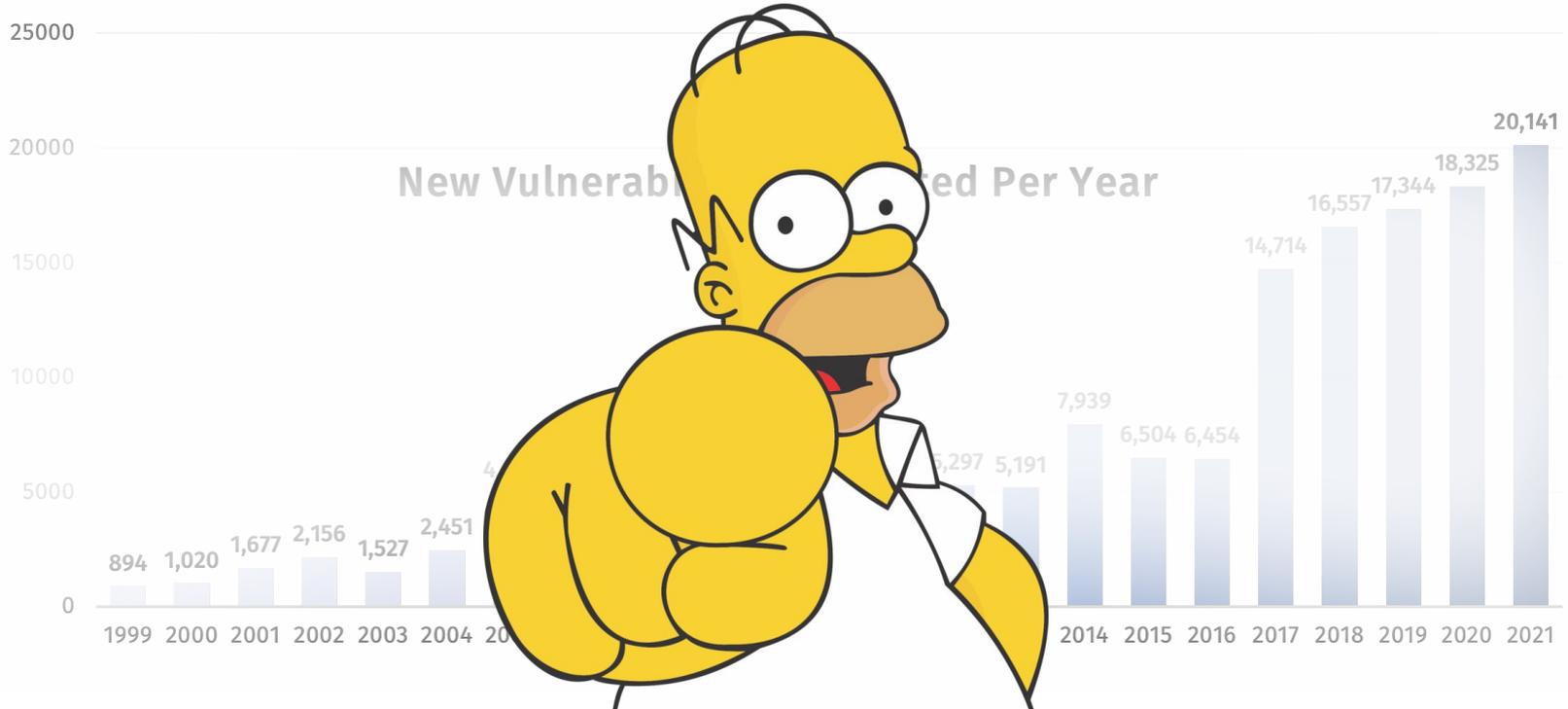
In the recent decade, fuzzing has found widespread interest. In 2000s, we have large continuous fuzzing platforms employing 100k's machines for automatic bug finding [23, 24, 46]. In academia, almost 50 fuzzing papers were published in the top conferences for Security and Software Engineering [42]. Hopefully, none of them finds any bugs. If indeed they don't, we should be questioning ourselves on the effectiveness of the current

## Goal #3: learn state-of-the-art tools

```
== ASAN: heap-use-after-free on address  
0x61900000047f at pc 0x00000040a52c bp  
0x7fff9200dbf0 sp 0x7fff9200dbe0  
READ of size 1 at 0x61900000047f thread T0  
#0 0x40a52b in src/main.cpp:30  
#1 0x40e088 in std_function.h:297  
#2 0x40d605 in std_function.h:687  
#3 0x40b8d5 in src/...  
#4 0x7f9a498ff412 in ...:308
```



# Now go forth and teach others!



# Thank you!

