COMPILER VERIFICATION - TECHNIQUES & TECHNOLOGIES

Rakesh Pothengil (Manager, System Software)

ABOUT THIS LECTURE

Objectives

- Introduce Compiler Verification techniques utilized in practice.
- Deep dive into two interesting and popular technologies:
    - Compiler Fuzzing
    - Live Code Mutation based Compiler Verification

VALIDATION? VERIFICATION? QA?

Validation Team:

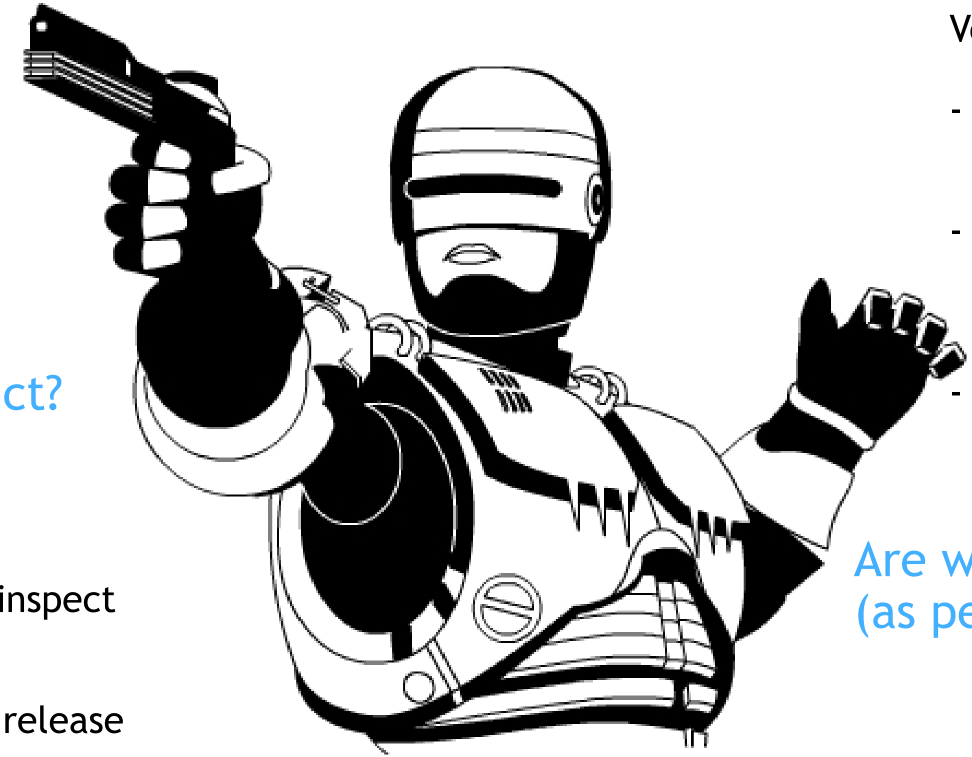
- “Should deter crime”
- “Shouldn’t be corrupt”
- “Should be controllable”
- …

Are we making the right product?

QA Team:

- “What should be the frequency to inspect the results of the diagnostic tests”
- “What tests to execute before the release of the next update?”
- …

Do we have a set of processes ready to ensure Product meets Quality requirements?



Verification Team:

- “Does reload happen within 1 ms”
- “Does Classification module perceive Criminals and Law enforcement differently”
- “Is the accuracy of hitting a target 100%”

Are we making the product right (as per spec)?

COMPILER VERIFICATION

Failure Types

Compilers are a complex piece of software!

Mis-compilations

- Compiler silently compiles to a wrong code.
- Difficult to detect as user doesn’t attribute a problem in compilers usually.
- Very harmful

Crashes

- Compiler doesn’t generate the target code.
- Comparatively less harmful.

COMPILER VERIFICATION TECHNIQUES

Language Conformance Tests

```
…
void foo(int n)
{
  switch (n)
  {
     case 22:
     case 33:
       f (1); // warning: fallthrough
     case 44:
       f (2);
       __attribute__((fallthrough));
     case 55:
       if (n < 10)
       {
          f (3);
          break;
       }
       else
       {
          f (4);
          __attribute__((fallthrough));
       }
      case 66:
        f(5);
        __attribute__((fallthrough));
        f(6);
      case 77:
        f (7);
    }
}
```

1. A null statement marked with the attribute token fallthrough, is a fallthrough statement. The fallthrough attribute token shall appear at most once in each attribute list, with no attribute argument clause.

2. A fallthrough statement may appear within an enclosing switch statement, on some path of execution immediately between a preceding statement and a succeeding case-labeled statement.

3. [Note: If an implementation would have otherwise issued a warning about implicit fall through on a path of execution immediately after a fallthrough statement, it is encouraged not to. end note]

COMPILER VERIFICATION TECHNIQUES (1/4)

Language Conformance Tests

Accurate

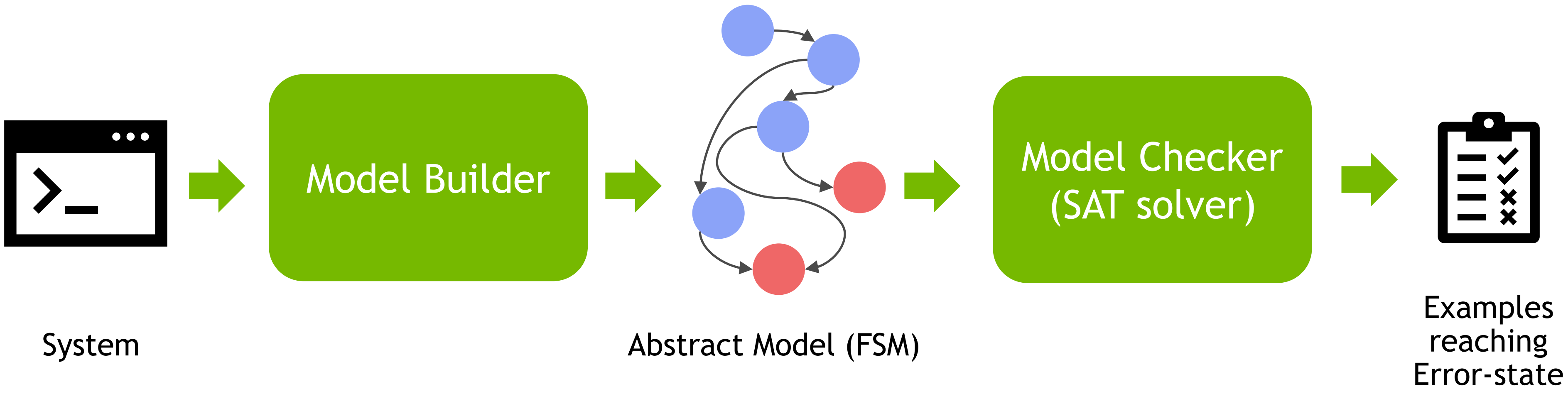
- Pinpointed tests for every feature
- Great Traceability wrt. Language Spec
- Coverage of all supported targets.

$$ Expensive

- Need to test Combination of features
- Skilled Resources
- Large number of targets to verify.
  E.g. GCC supports ~78 targets.
  “Common Tests” are ~1500. Effective tests: 117,000
  Now, consider compiler options!

COMPILER VERIFICATION TECHNIQUES

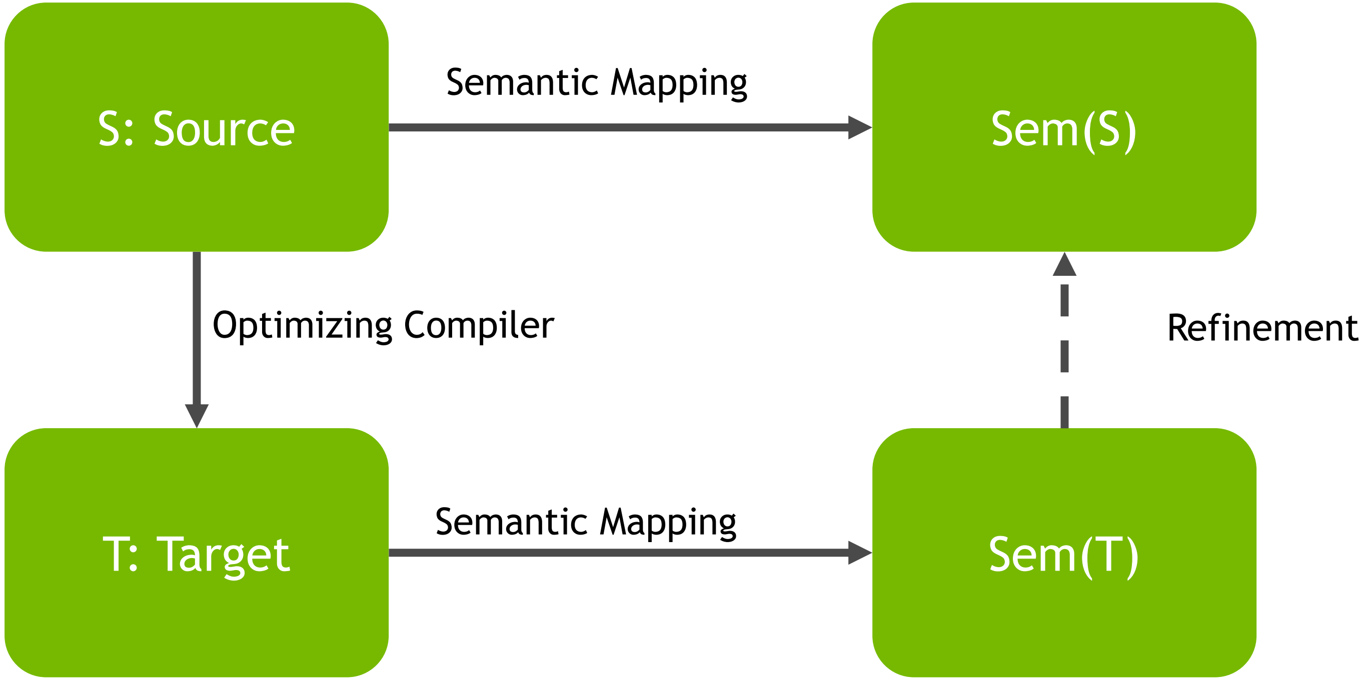
Formal Verification (Model Checking)



COMPILER VERIFICATION TECHNIQUES

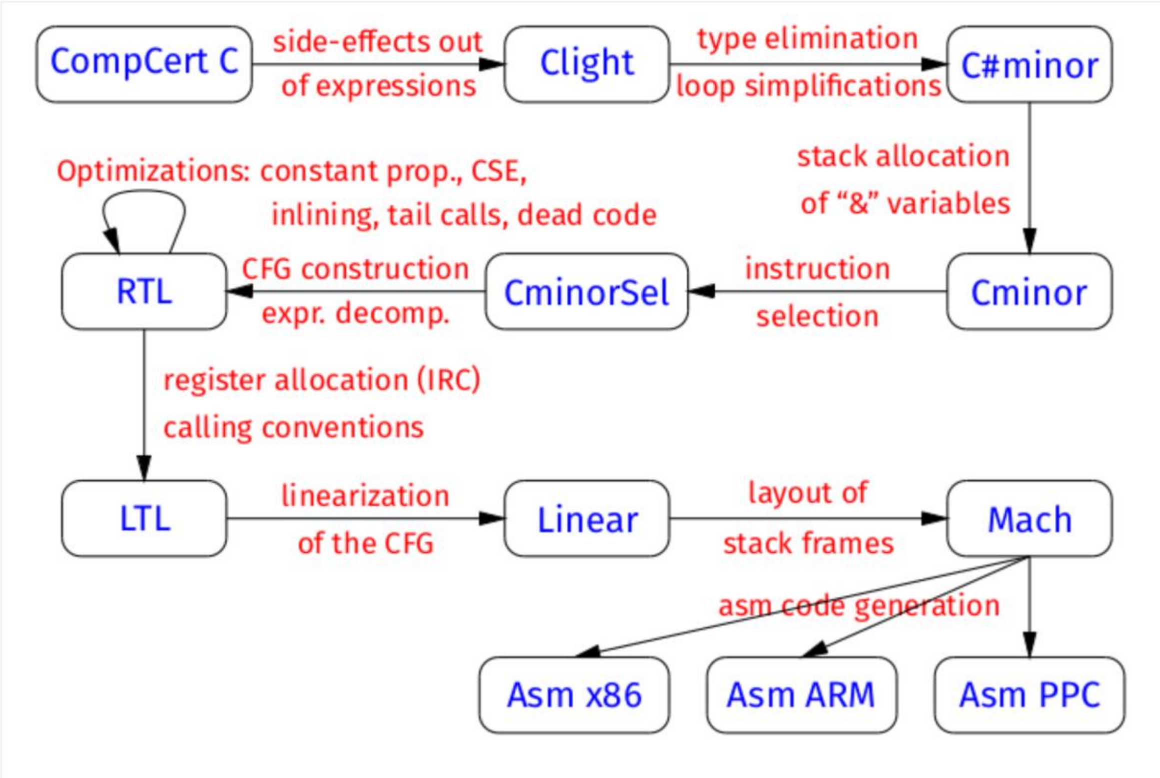
Formal Verification (Translation Validation)

Prove the various intermediate representations are semantically equivalent



COMPILER VERIFICATION TECHNIQUES

COMPCERT – Formally Verified Optimizing Compiler for C



COMPILER VERIFICATION TECHNIQUES

Formally Verified Software

Accurate

- Proved for correctness and has a very high degree of confidence.

Cumbersome

- Interpretation of results is cumbersome.
- False positives arising due to incorrect application of Formal Verification.

COMPILER VERIFICATION TECHNIQUES

Random Testing

- Random Testing is becoming a dominant approach in Compiler Verification.
- Involves automatic creation of random programs (also called fuzzing) to test the compiler.
- Key aim is to generate valid and invalid combinations of all syntactic elements a source program provides to test if Compiler can accept them.
- Fuzzers like Csmith have found over 400 unknown compiler bugs!

THE STORY OF FUZZ TESTING

An age-old practice forged with time

1950

Trash-deck Testing

Pulling cards from the trash and supplying it to the input program to check undesired behavior – Gerald Weinberg

1983

The Monkey

Steve Capps develops “The Monkey” to generate random inputs for MAC OS applications

1988

“Fuzzing”

Barton Miller coins the term at University of Wisconsin for his class project.

2012

“Mainstream” Popularity

ClusterFuzz, AFL, Project Springfield, OSS-Fuzz

CHARACTERISTICS OF A COMPILER FUZZER

What defines an ideal Fuzzer?

1. Cost-Effective

Easy to implement and extend

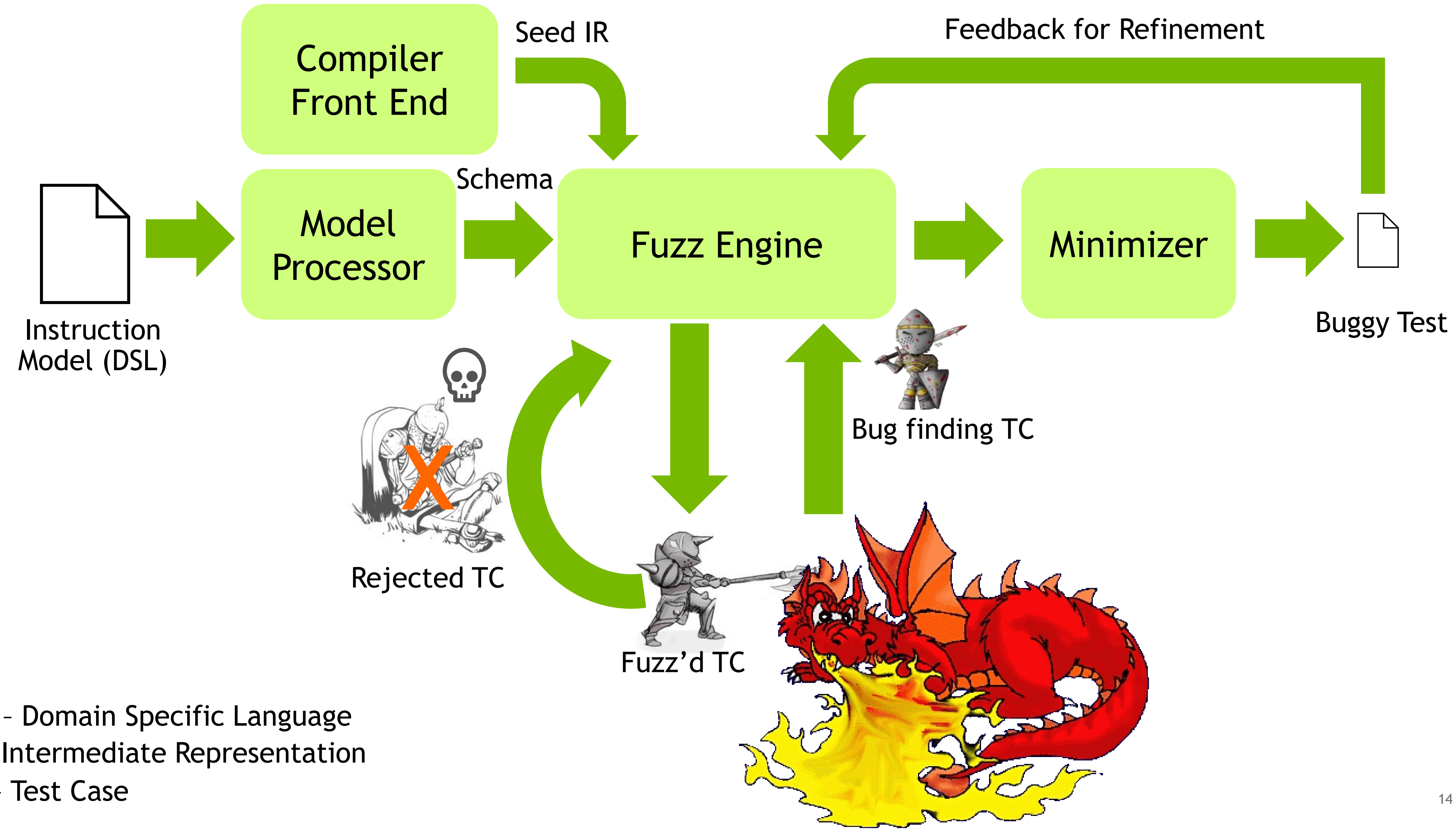
2. Interpretable Testcases

Not 1000s of lines of code.

3. Plausible Output

Should be close to user's usage of the constructs.

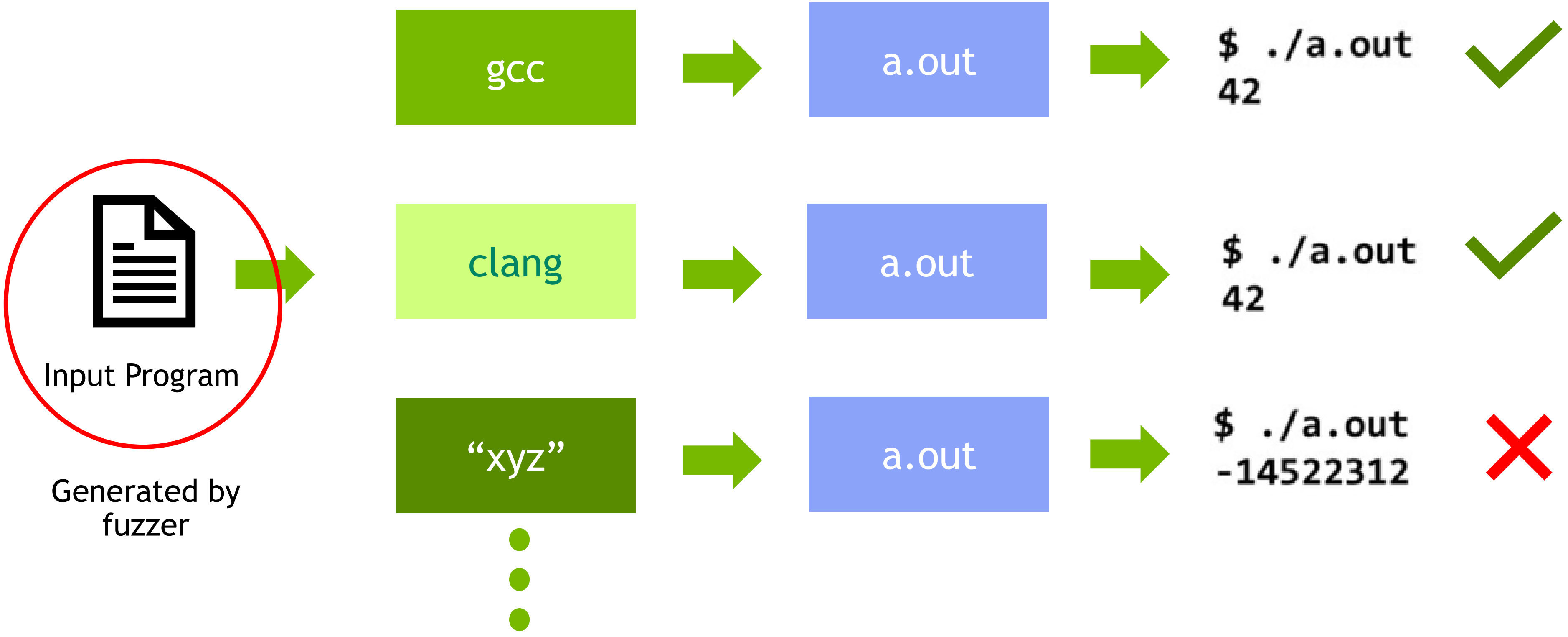
ANATOMY OF A COMPILER FUZZER



DSL – Domain Specific Language
IR – Intermediate Representation
TC – Test Case

DIFFERENTIAL TESTING

Provide an input on similar applications (compilers), expecting each one to give the same result.



TEST CASE MINIMIZER

Delta Debugging

- Utilizes Divide & Conquer.
- Divide the code into 2 similar sized "chunks" (δ1 and δ2). There are 3 outcomes
- Case 1: Reduce to δ1. The test of δ1 fails - δ1 is the smaller test case.
- Case 2: Reduce to δ2. The test of δ2 fails – δ2 is the smaller test case.
- Case 3: Ignorance. Both tests or are unresolved – neither δ1 nor δ2 qualify as possible simplifications.
- Case 1 & Case 2: Continue the search with failing subset.
- Case 3: Increase the size of the "chunk" and continue.
- Read More: Simplifying and isolating failure-inducing input – Andreas Zeller, Ralf Hildebrandt

TEST CASE MINIMIZER

Delta Debugging - Variants

1. Text Based Reducers: ddmin- Text oriented chunk removal.

2. AST Based Reducers: HDD - Removes the AST to generate variants.

3. Language Specific Reducer Tools – topformflat – puts syntactically related tokens in one line.

4. AST Based Reducers using Runtime Information – examine dead code regions and remove them first.

MINIMIZATION

Challenges

The Validity Problem

```
int main()
{
  int x;
  x = 2;
  return x+1;
}
```

```
int main()
{
  int x;
  return x+1;
}
```

COMPILER FUZZING

Key Challenges

Implementing a fuzzer is effort-intensive and complex!

- CSmith
  - 41K lines of code in C++
  - Just upgrading from C to OpenCL took 9 months with additional 8K LOC.
- How to generate more meaningful programs?
- How to target Compiler Optimizations effectively?

Fuzzer Developer

- Should be an expert of the target language.
- Should be able to model every aspect of the grammar of a programming language.
- Should understand compiler nitty grities - to generate “intelligent” programs to test optimizations

TRADITIONAL FUZZERS

CSmith : Case Study

1. Cost-Effective ✗

40K+ LOC implemented to handle a subset of C.

2. Interpretable Testcases ✗

Average size is 1200 LOC (excluding headers). ~4 hours/test spent in reduction.

3. Plausible Output ✗

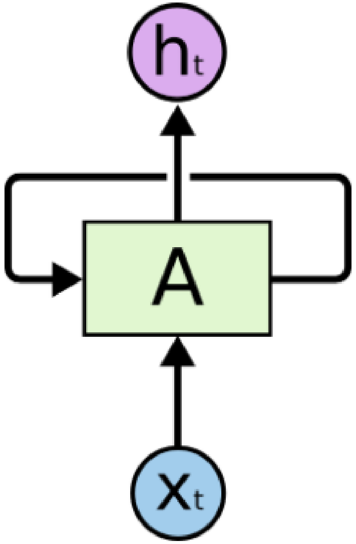
Obscure combination of constructs.

RECURRENT NEURAL NETWORKS (RNN)

Thoughts have persistence

- Problem: Classify what kind of event is happening at every point in a movie?
- RNN is a network that works on the present input by taking into consideration the previous output (feedback).
- Good enough to predict the next word given – “The clouds are in the *sky*”
- Not good for “I grew up in France … I speak fluent *French*.”
- LSTM : Long Short Term Memory Networks – Capable of learning Long

Term dependencies. Very effective for Text Prediction.



DEEP SMITH - SYSTEM OVERVIEW

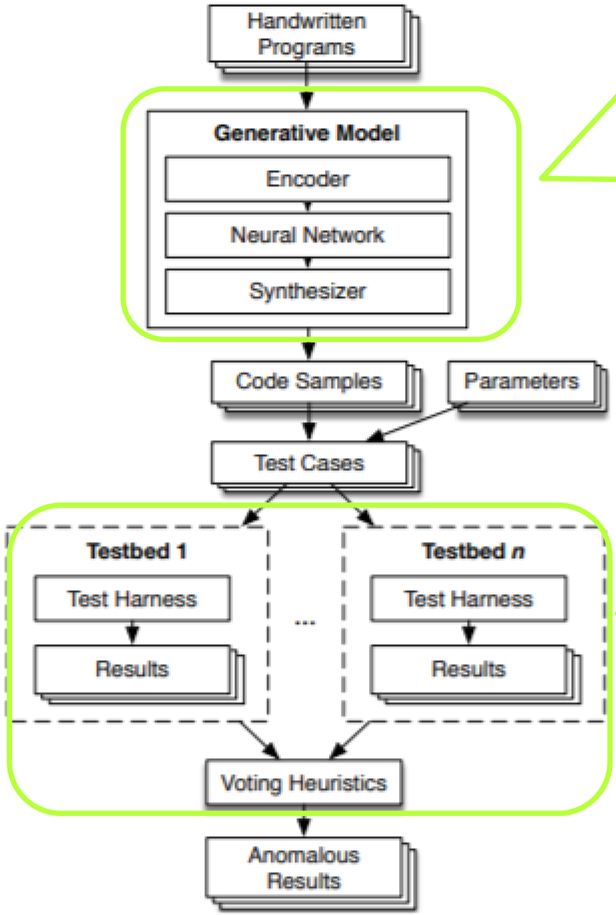


Figure 1: DeepSmith system overview.

- **Encoder** encodes Handwritten Programs to numeric sequences to be fed into the machine learning model.
- **Neural Network** uses LSTM – which associates current input with previous input in a sequence. Builds the ability to predict the next sequence based on a stream of tokens (Much like a "text predictor").
- **Synthesizer** – Based on a seed, the model starts predicting the sequence of tokens that should come next. It does so till "{" and "}" are balanced. This sequence is translated to the text to create a Test Case.

- **Testbed** - Executes generated test on various compilers.
- **Voting Heuristics** – Chooses which results are the expected ones and which are erroneous

Read More: "Compiler Fuzzing through Deep Learning" presented in ISSTA '18. – Chris Cummins et. al

VOCABULARY ENCODING

Translating a Program to a solvable ML Problem?

- **Key Principle:** Convert a program to numeric sequences to feed as an input to a Neural Network.

1. Preprocess
    1. Expand macros and remove conditional compilation and comments.
    2. Consistent naming convention – Rename program variables to an arbitrary but consistent pattern based on their declaration. E.g. {a,b,c,…,aa,ab,ac…} for variables and {A, B, C, …, AA, AB, AC …} for functions.
    3. Uniform coding style to ensure consistent use of braces, parentheses and white spaces.
2. Encode
    1. Programming language’s features are treated as individual tokens and remaining are character-level.

VOCABULAR ENCODING

Preprocessing

```
#define MY_CONST 3.14
// A simple kernel
kernel void Foo (global float * input, const float x) {
  input[get_global_id(0)] *= MY_CONST + x;
}
```

```
kernel void A (global float * a, const float b) {
  a[get_global_id(0)] *= 3.14 + b;
}
```

VOCABULARY ENCODING

```
kernel void A (global float * a, const float b) {
  a[get_global_id(0)] *= 3.14 + b;
}
```

Vocabulary

kernel: 0
‘space’: 1
void: 2
A: 3
(: 4

global: 5
float: 6
*: 7
a: 8
,: 9

const: 10
b: 11
): 12
{: 13
\n: 14

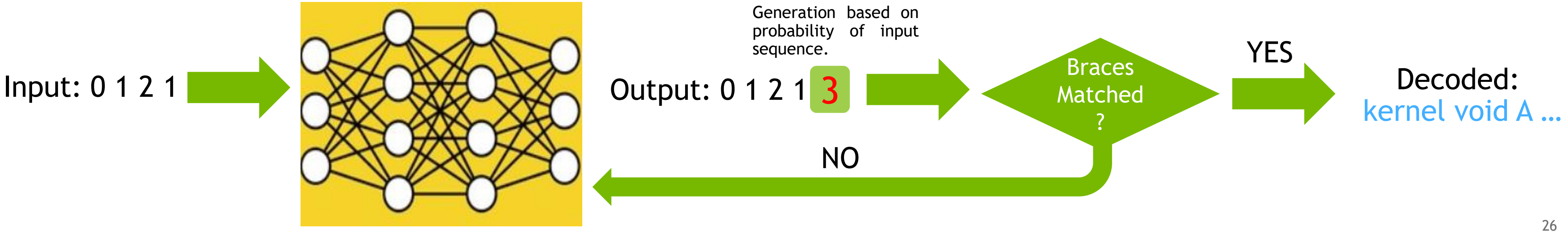
[: 15
…

Encoded 0 1 2 1 3 1 4 5 1 6 1 7 1 8 9 1 10 …

NEURAL NETWORK + SYNTHESIZER

LSTM (Long Short-term Memory)

- Used 30M token corpus by mining OpenCL programs on github.
- Learns probability distribution over the corpus.
- Took 12 hours of training time on GPU (Nvidia Tesla P40).
- Trained Network is provided a “seed token” that resembles start of the program. On each token that is generated, bracket depth counter is implemented.
  - { increases counter and } decreases the counter and process stops when they are balanced.



TEST HARNESS + VOTING HEURISTICS

- Test Harness accepts a Fuzzed program and generates input for it.
  - Python code of a few 100 lines and uses domain knowledge to create the inputs.
- Voting Heuristics
  - Uses Differential Testing that checks the majority of the outcomes.
  - Heuristics determine build failures, undefined behavior and removes false positives to arrive at the “correct” behavior.

DEEP SMITH EVALUATION

- Conducted 2000 hours of automated testing across 10 OpenCL compilers.
- Found bugs in all compilers and reported 67 bugs to compiler vendors.
- Findings
  - DeepSmith finds bugs by creating tests resembling hand-written code which CLSmith cannot. E.g.
    - Using thread identity to modify control flow.

      ```
      int g = get_global_id(0);
      if (g < e – d – 1) …
      ```

    - Pattern is based on real-world code which cannot be generated by a typical fuzzer.
      - 46% of OpenCL code on github used this pattern. DeepSmith “learned” this pattern and used in several cases.

CONCLUSIONS

DeepSmith

1. Cost-Effective ✓

Automatic inference of language from examples.

2. Interpretable Testcases ✓

102x less code than the state-of-the-art fuzzer.

3. Plausible Output ✓

Learns from hand-written code and therefore outputs similar looking tests.

CURIOUS?

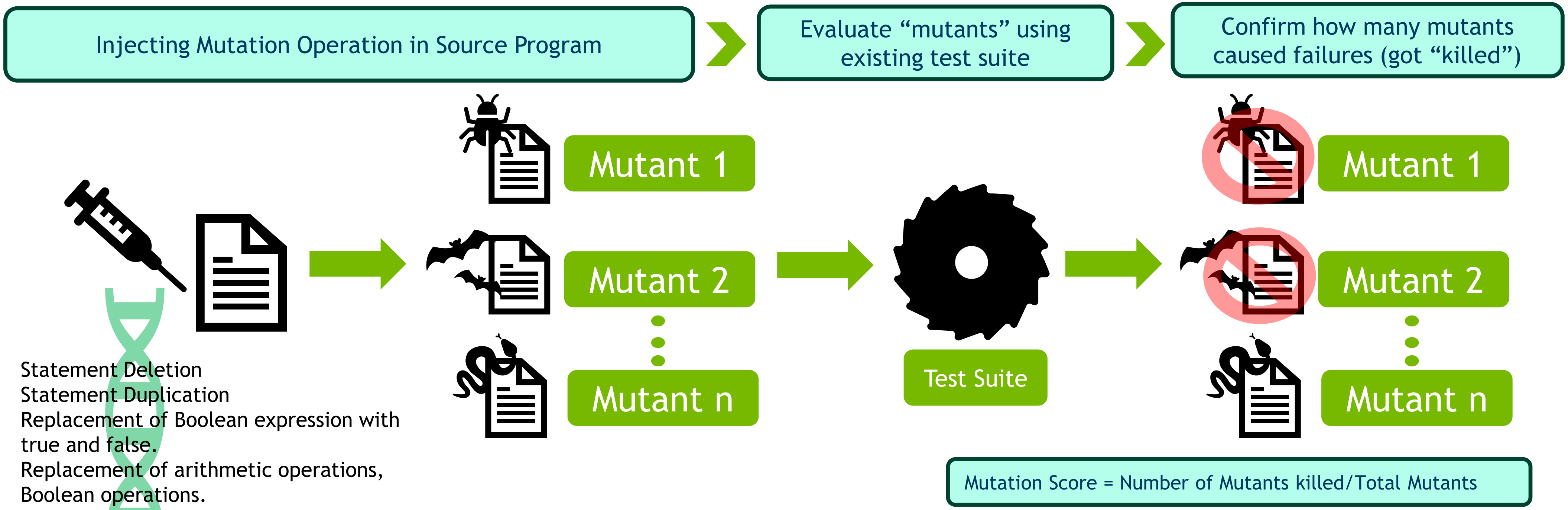
Try this out for your favourite programming language

- Using the framework created for DeepSmith, create a fuzzer for your favourite Programming Language.
    - Download DeepSmith from github (https://github.com/ChrisCummins/phd/tree/master/deeplearning/deepsmith).
    - Create a vocabulary encoding module to provide consistent inputs for training for your chosen language.
    - Supply the encoded module to train the NN (on Nvidia GPUs?).
    - Try to get fuzzing!

COMPILER VERIFICATION THROUGH EMI BASED LIVE CODE MUTATION

MUTATION TESTING

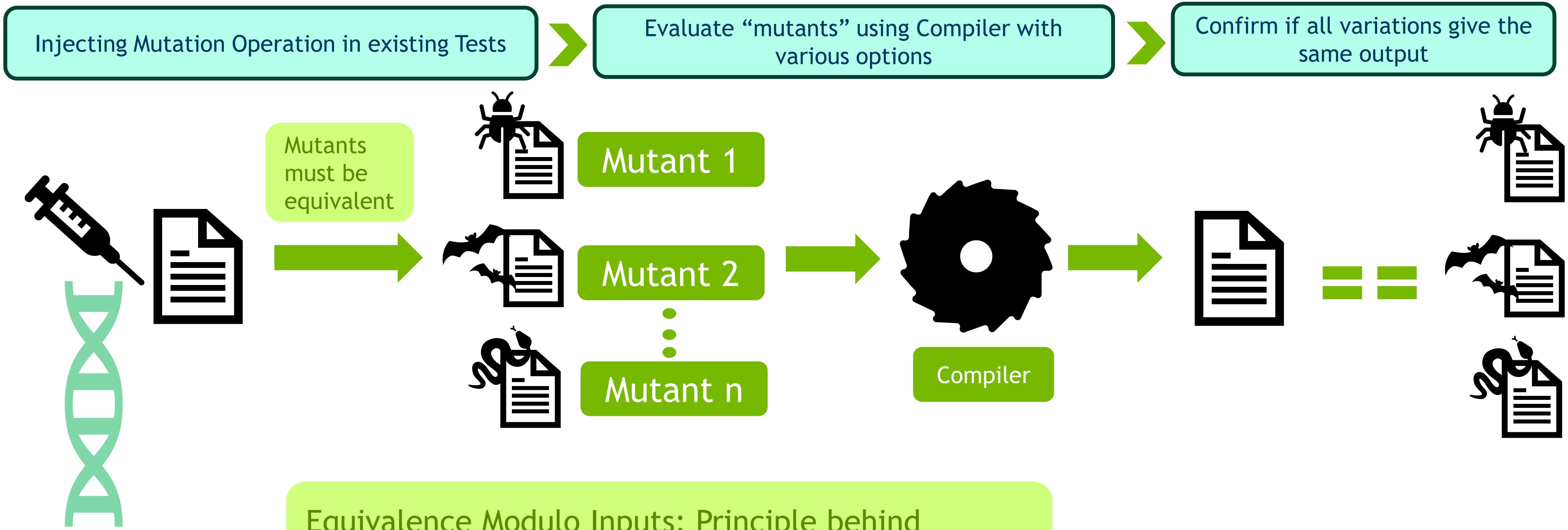
- Concept introduced by Richard Lipton (1971)
- Introduced to evaluate the quality of existing software tests



- Statement Deletion
- Statement Duplication
- Replacement of Boolean expression with true and false.
- Replacement of arithmetic operations, Boolean operations.
- Replacement of variables with others in the same scope.

MUTATION TESTING

Compiler Verification Perspective



Equivalence Modulo Inputs: Principle behind generating equivalent variants of a Program.

EQUIVALENCE MODULO INPUTS (EMI)

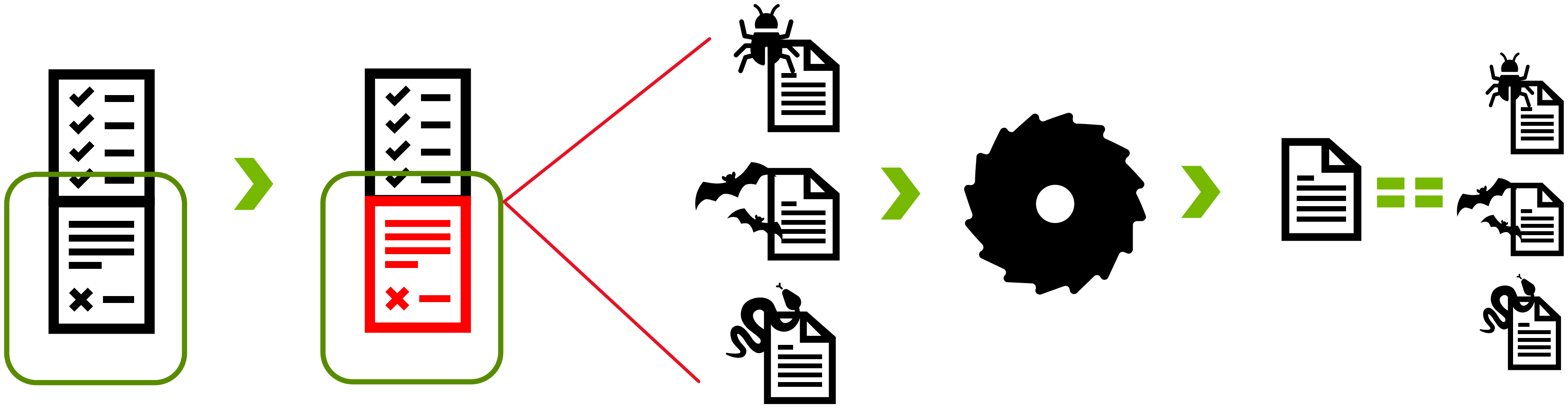
Yet another Compiler Verification Methodology

- Generate multiple variants of an input program (mutation)
  - Mutated tests must be valid programs.
  - Must not change the semantics of the original programs.
- Execute variants in optimization and no-optimization paths.
  - Both paths must result in the same behavior.
  - EMI variants are although equivalent but trigger different static data and control flow. This different control flow critically affects which optimizations are enabled.
  - Each variant demands the exact same output.
- Key Challenges:
  - How to create different and semantically equivalent variants.

GENERATING EMI VARIANTS

Approach

- Profile & Mutate : Prune portions of the program that are not executed to create variants.



Profile to find un-executed (dead) code

Prune dead code to create variants.

Compile on all optimization paths.

Confirm Results

Principle: Modifying dead code shouldn’t cause different behaviors.

GENERATING EMI VARIANTS

Detailed Approach

Input Source:
1. Csmith [most effective]
2. Open Source Projects
3. GCC/LLVM regression suites

LLVM based tool to prune AST of unexecuted statements based on a stochastic logic.

gcc, LLVM

C-Reduce to reduce test cases.

147 confirmed, unique bugs in GCC & LLVM. Most of them are miscompilations

Profiler: gcov

Profile to find un-executed (dead) code

Prune dead code to create variants.

Compile on all optimization paths.

Confirm Results

Principle: Modifying dead code shouldn't cause different behaviors.

EMI VARIANTS

Examples

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z,
      long l) {
  if (x.c != 10) abort();
  if (x.d != 20) abort();
  if (x.e != 30) abort();
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) abort();
  if (z.c != 12) abort();
  if (z.d != 22) abort();
  if (z.e != 32) abort();
  if (l != 123) abort();
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
}
```

Original

Deletion of certain abort() functions, caused “f” to be inlined by the Clang compiler and incompatibility between GVN and SRoA led to the miscompilation

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z,
      long l) {
  if (x.c != 10) /* deleted */;
  if (x.d != 20) abort();
  if (x.e != 30) /* deleted */;
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) /* deleted */;
  if (z.c != 12) abort();
  if (z.d != 22) /* deleted */;
  if (z.e != 32) abort();
  if (l != 123) /* deleted */;
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
}
```

EMI Variant

EMI VARIANTS

Examples

```
int a, b, c, d, e;
int main() {
  for (b = 4; b > -30; b--)
    for (; c;)
      for (;;) {
        b--
        e = a > 2147483647 - b;
        if (d) break;
      }
  return 0;
}
```

Mutation in Dead Code

Invariant

```
int a, b, c, d, e;
int main() {
  for (b = 4; b > -30; b--)
    for (; c;)
      for (;;) {
        e = a > 2147483647 - b;
        if (d) break;
      }
  return 0;
}
```

Triggers Loop Invariant Motion

Spurious Warning (in original context): Overflow when b == -1

```
int a, b, c, d, e;
int main() {
  for (b = 4; b > -30; b--)
    e = a > 2147483647 - b;
    for (; c;)
      for (;;) {
        if (d) break;
      }
  return 0;
}
```

Mis-compilation: Infinite Loop

EMI IMPLEMENTATIONS AND DRAWBACKS

ORION & ATHENA

- ORION – Randomly prunes in dead code regions.
- ATHENA – Inserts code segments in dead code regions.
- Drawbacks
  - Only limited to dead code regions.
  - Dead codes are susceptible to be optimized away by the compiler.
  - Mis-compilation bug in dead codes is not observable as the wrong code is not executed.

LIVE CODE EMI MUTATION

Overview

- Instead of inserting mutations in dead code, create EMI mutations in Live Code.
- Key Challenge: Inserting mutations that still preserve original semantics yet cause a side-effect so that compiler preserves it.
- Approach

Instrument source program to find out values of live variables.

Synthesize Live Code EMI (based on variable values)

Inject synthesized EMIs into Source Program (Live Code Mutation)

Test on all optimizations of the compiler.

Read More: Finding Compiler Bugs via Live Code Mutation – Chengnian Sun, Vu Le, Zhendong Su

LIVE CODE EMI MUTATION

Generating variants

- Always False Conditional Block (FCB)
  - Non-empty if/while block with the conditional predicate always evaluating to false.
  - The conditional predicate is generated by considering the value of the “live” variables at a specific program point.
  - Body is a “random” code segment generated using variables in the context (as much as possible).

```
int a;

int main () {
  int b = -1, d, e = 0, f[2] = { 0 };
  unsigned short c = b;

  for (; e < 3; e++)
    for (d = 0; d < 2; d++)
      /* a=0, b=-1, c=65535, d={0,1}, e={0,1,2}, f[0]=0 */
       if (a < 0) // Inserted code highlighted in gray.
        for (d = 0; d < 2; d++)
          if (f[c])
            break;
  return 0;
}
```

LIVE CODE EMI MUTATION

Generating variants

- Always True Guard
  - An “if” statement is inserted at a specific location which is always true thereby generating a structural difference but not affecting the original control flow.

- Always True Conditional Block (TCB)
  - A non-empty if block that has the conditional predicate as “true” always and introduces a side-effect in its body.
  - The side-effects should be reversed after the block exits to avoid any change in original semantics

Create Backup Variable

Update program variable

```
int backup_v = v;
if ((synthesized true predicate)) {
  v = (synthesized valid expression);
  if/while((synthesized false predicate)) {
    print v;
  }
}
v = backup_v;
```

Use updated value to avoid it being optimized away

Restore Original Variable

DETAILED APPROACH

Hermes

INSTRUMENT

- LLVM based tool instruments statements that generate Execution Profile at specific program points.
- Not all variables at all points are required. Selection can be based on a stochastic logic.

EXECUTION PROFILE

- Has a schema that stores the program's state.
- Values of variables at selected program points are recorded.

SYNTHESIZE

- FCB – Based on the program state variables, a false predicate is generated.
  - Randomly choose a logical operator.
  - Compute target truth values of the children such that the predicate evaluates to the expected truth value.
- TCB
  - Live variables are used to create expressions that are merged together with various operators to form a valid single expression.

DETAILED APPROACH

Synthesizing a Predicate

Algorithm 2: Synthesize a predicate

```
function SynPred (Env env, Bool expected, Int depth):
    if depth = 0 then
        return SynAtom(env, expected)
    switch Random(4) do
        /* synthesize a negation */
        case 1 do return SynNeg(env, expected, depth)
        /* synthesize a conjunctive predicate */
        case 2 do return SynCon(env, expected, depth)
        /* synthesize a disjunctive predicate */
        case 3 do return SynDis(env, expected, depth)
        /* synthesize an atomic predicate */
        case 4 do return SynAtom(env, expected)

function SynNeg (Env env, Bool expected, Int depth):
    return Expr('!', SynPred(env, !expected, depth − 1))

function SynCon (Env env, Bool expected, Int depth):
    if expected then left ← true, right ← true
    else if FlipCoin() then left ← true, right ← false
    else left ← false, right ← FlipCoin()
    left_pred ← SynPred(env, left, depth − 1)
    right_pred ← SynPred(env, right, depth − 1)
    return Expr('&&', left_pred, right_pred)

function SynDis (Env env, Bool expected, Int depth):
    if ! expected then left ← false, right ← false
    else if FlipCoin() then left ← false, right ← true
    else left ← true, right ← FlipCoin()
    left_pred ← SynPred(env, left, depth − 1)
    right_pred ← SynPred(env, right, depth − 1)
    return Expr('||', left_pred, right_pred)

function SynAtom (Env env, Bool expected):
    /* Rule 1: randomly pick one variable v and
       construct a relational predicate with expected
       truth value over v and a constant */
    /* Rule 2: randomly pick two variables v1 and v2,
       and construct a relational predicate with
       expected truth value over v1 and v2 */
```

depth: 2 expected: true

Case 2

left: true right: true

left_pred: !a < 10

right_pred: !b != 1

Expr: !

a < 10

Expr: !a<10 && ! b!=1

depth: Used to notify the number of predicate sub-expressions that must be generated. Greater value means more complex expressions.

env: Execution Profile – List of live variables and their values.

a:10	b:1	c:-1

NVIDIA

DETAILED APPROACH

Synthesizing Valid Expressions

Algorithm 3: Synthesize valid expressions

```
function SynExpr (Env env):
    worklist ← Sample(env, Random(|dom(env)|))
    while |worklist| > 1 do
        if FlipCoin() then                          // unary expression
            v ← Sample(worklist, 1)
            uop_list ← a shuffled list of unary operators
            foreach uop ∈ uop_list do
                if IsUndefined(env, v, uop) then
                    continue
                worklist ← (worklist \ {v})
                worklist ← worklist ∪ {Expr(uop, v)}
                break
        else                                        // binary expression
            {u, v} ← Sample(worklist, 2)
            bop_list ← a shuffled list binary operators
            foreach bop ∈ bop_list do
                if IsUndefined(env, u, v, bop) then
                    continue
                worklist ← (worklist \ {u, v})
                worklist ← worklist ∪ {Expr(bop, u, v)}
                break
    return the only expression in worklist
```

env: Execution Profile – List of live variables and their values.

a:10 | b:1 | c:-1

Bop list – List of binary operators.

& | \| | + | ...

worklist: a:10 | b:1

{u,v}: a:10 | b:1

a & b

Key Challenge: isUndefined function needs to take care of subtle undefined behaviors considering the types and the operations that can be performed on them. – solved by emulation

NVIDIA.

EVALUATION

Bug Examples

```
int a;
int b;
short c;

int main () {
  int j;
  int d = 1;

  for (; c >= 0; c++) {
    a = d;
    d = 0;

    if (b) {
      printf ("%d", 0);
      if (j) {
        printf ("%d", 0);
      }
    }
  }

  printf ("%d\n", d);
  return 0;
}
```

Always False Conditional Block

Evaluating an un-initialized variable.

The mis-compiled binary using GCC prints 1 instead of 0.

Undefined behavior is pulled out due to optimization

Root Cause

Loop Un-switching
Moving a conditional out of the loop to improve Parallelization

```
int
main ()
{
  int j, d = 1;
  if (j)
    {
      for (; c >= 0; c++)
      {
        a = d;
        d = 0;
        if (b)
          printf ("%d", 0);
        printf ("%d", 0);
      }
    }
    else
    {
      d = 0;
      for (; c >= 0; c++)
      {
        a = d;
        if (b)
          printf ("%d", 0);
      }
    }
  }
}
```

EVALUATION

Bug Examples

```c
int a = 2, b = 1, c = 1;

int fn1 () {
  int d;
  for (; a; a--) {
    for (d = 0; d < 4; d++) {
      int k;
      if (c < 1)
        if (k) c = 0;
      if (b) continue;
      return 0;
    }
    b = !1;
  }
  return 0;
}

int main () {
  fn1 ();
  if (a != 1)
    __builtin_abort ();
  return 0;
}
```

Always False Conditional Block

Evaluating an un-initialized variable inside an FCB

Evaluating an un-initialized variable no longer in FCB

The mis-compiled binary using GCC aborts instead of terminating normally.

Root Cause

If-Combine
Combining nested ifs to a single if statement with multiple conditional expressions using conjunction.

```c
int fn1 ()
{
  int d;
  for (; a; a--) {
    for (d = 0; d < 4; d++) {
      int k;
      if (c < 1 && k)
        c = 0;
      if (b) continue;
        return 0;
    }
    b = !1;
  }
return 0;
}
```

CONCLUSIONS

Live Code Mutation is a realistic and practical way of finding bugs

	GCC	LLVM	Total
Fixed	85	47	132
Not-yet-fixed	10	26	36
Duplicate	28	20	48
Invalid	1	0	1
Total	124	93	217

1.7 seconds to profile and synthesize a test

400

Generates 400 programs/hour (single core)

SUMMARY

Compiler Verification – Techniques & Technologies

- Use cases of Compilers are touching human lives in a far greater way than before. Compiler Verification is increasingly becoming more and more important.
- We have barely scratched the surface of the problems in Verification.
  - Several ideas are still at research phase.
- Problems in verification are at the cusp of various technology domains which are increasingly becoming stronger thanks to AI/ML, HPC etc.
- It’s an interesting space to watch out for!

Questions?