On the Surprising Efficiency and Exponential Cost of Fuzzing

Marcel Böhme
Software Security
MPI-SP & Monash

Keywords: Vulnerability Discovery, Automated Software Testing, Effectiveness, Efficiency, Scalability, Guarantees
whoami

• Fuzzing for Automatic Vulnerability Discovery
  • Making machines attack other machines.
  • Focus on scalability, efficiency, and effectiveness.

• Foundations of Software Security
  • Assurances in Software Security
  • Fundamental limitations of existing approaches
  • Drawing from multiple disciplines (information theory, biostatistics)
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;

    assert(crash != 1);
}
Whitebox Fuzzing

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if (crash == 1) abort();
}
```

Path Conditions

✓ $\phi_1 = (s0 \neq 'b')$
Whitebox Fuzzing

void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if(crash == 1) abort();
}

Path Conditions
✓ φ₁ = (s0 != 'b')
✓ φ₂ = (s0 == 'b') \ (s1 != 'a')
Whitebox Fuzzing

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!' )
                    crash = 1;
    if (crash == 1) abort();
}
```

**Path Conditions**

- $\phi_1 = (s0 \neq 'b')$
- $\phi_2 = (s0 == 'b') \land (s1 \neq 'a')$
- $\phi_3 = (s0 == 'b') \land (s1 == 'a') \land (s2 \neq 'd')$
- $\phi_4 = (s0 == 'b') \land (s1 == 'a') \land (s2 == 'd') \land (s3 \neq '!' )$
- $\phi_5 = (s0 == 'b') \land (s1 == 'a') \land (s2 == 'd') \land (s3 == '!' )$

• ✓ $\phi_1$
• ✓ $\phi_2$
• ✓ $\phi_3$
• ✓ $\phi_4$
• ✗ $\phi_5$
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if(crash == 1) abort();
}

Path Conditions
✓ $\varphi_1 = (s0 \neq 'b')$
✓ $\varphi_2 = (s0 == 'b') \land (s1 \neq 'a')$
✓ $\varphi_3 = (s0 == 'b') \land (s1 == 'a') \land (s2 \neq 'd')$
✓ $\varphi_4 = (s0 == 'b') \land (s1 == 'a') \land (s2 == 'd') \land (s3 \neq '!')$
✗ $\varphi_5 = (s0 == 'b') \land (s1 == 'a') \land (s2 == 'd') \land (s3 == '!')$

Whitebox Fuzzing: Most Effective!
void crashme (char s0, char s1, char s2, char s3) {
  int crash = 0;
  if (s0 == 'b')
    if (s1 == 'a')
      if (s2 == 'd')
        if (s3 == '!')
          crash = 1;
    
  if(crash == 1) abort();
}

It can prove the absence of assertion violation,
by enumerating all paths and modulo some assumptions.

Path Conditions
✓ $\phi_1 = (s0 \neq 'b')$
✓ $\phi_2 = (s0 == 'b') \land (s1 \neq 'a')$
✓ $\phi_3 = (s0 == 'b') \land (s1 == 'a') \land (s2 \neq 'd')$
✓ $\phi_4 = (s0 == 'b') \land (s1 == 'a') \land (s2 == 'd') \land (s3 \neq '!')$
✗ $\phi_5 = (s0 == 'b') \land (s1 == 'a') \land (s2 == 'd') \land (s3 == '!')$

Whitebox Fuzzing: Most Effective!
Whitebox Fuzzing: Quite Efficient!

void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    
    if(crash == 1) abort();
}

Path Conditions
✓ \( \varphi_1 = (s0 \neq 'b') \)
✓ \( \varphi_2 = (s0 == 'b') \land (s1 \neq 'a') \)
✓ \( \varphi_3 = (s0 == 'b') \land (s1 == 'a') \land (s2 \neq 'd') \)
✓ \( \varphi_4 = (s0 == 'b') \land (s1 == 'a') \land (s2 == 'd') \land (s3 != '!') \)
✗ \( \varphi_5 = (s0 == 'b') \land (s1 == 'a') \land (s2 == 'd') \land (s3 == '!') \)

Whitebox Fuzzing: Quite Efficient!
Whitebox Fuzzing: Quite Efficient!

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if (crash == 1) abort();
}
```

We only need 3 inputs to find the bug, on average, if we choose each path at random without replacement.

Choose a random path from the multivariate hypergeometric (i.e., enumerate). Choose some input that exercises that path (by constraint solving).

Path Conditions

- ✓ $\phi_1 = (s0 \neq 'b')$
- ✓ $\phi_2 = (s0 == 'b') \land (s1 \neq 'a')$
- ✓ $\phi_3 = (s0 == 'b') \land (s1 == 'a') \land (s2 \neq 'd')$
- ✓ $\phi_4 = (s0 == 'b') \land (s1 == 'a') \land (s2 == 'd') \land (s3 \neq '!')$
- ✗ $\phi_5 = (s0 == 'b') \land (s1 == 'a') \land (s2 == 'd') \land (s3 == '!')$
Blackbox Fuzzing: just random, really.

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if (crash == 1) abort();
}
```

For each parameter, choose 1 of 256 values uniformly at random.
Blackbox Fuzzing: just random, really.

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if (crash == 1) abort();
}
```

For each parameter, choose 1 of 256 values uniformly at random.

It can never prove the absence of assertion violation!

August 1969

NOTES ON STRUCTURED PROGRAMMING by prof.dr.Edsger W.Dijkstra

On the reliability of mechanisms.

Corollary of the first part of this section:

Program testing can be used to show the presence of bugs, but never to show their absence!
Blackbox Fuzzing: just random, really.

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;

    if (crash == 1) abort();
}
```

For each parameter, choose 1 of 256 values uniformly at random.

It can never prove the absence of assertion violation!

Well, that’s not entirely true. We can estimate a “residual risk”.

---

Blackbox Fuzzing: just random, really.

void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    
    if(crash == 1) abort();
}

- Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.
- Blackbox Fuzzer: Discovers the bug after \((1/256)^4\)^{-1} \approx 4 \text{ billion inputs}, in expectation.
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;

    if(crash == 1) abort();
}

- Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.
- Blackbox Fuzzer: Discovers the bug after \((1/256)^4\)^{-1} \approx 4 \text{ billion inputs}, in expectation.

So, whitebox fuzzing is better, right?
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;

    if(crash == 1) abort();
}

• Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.
• Blackbox Fuzzer: Discovers the bug after \((1/256)^4\)^{-1} \approx 4 billion inputs, in expectation.

So, whitebox fuzzing is better, right? **Wrong.** At least not always.
Partition Testing Does Not Inspire Confidence

Dick Hamlet, Member, IEEE, and Ross Taylor

This study was undertaken because partition testing did not live up to its intuitive value in two earlier studies. In their brief for random testing [3], Duran and Ntafos published a precise comparison between it and partition testing. Their surprising result is that the two methods are of almost equal value, under assumptions that seem to favor partition testing. Random testing has a decidedly spotty reputation, probably because it makes almost no use of special information about the program being tested. It is certainly counterintuitive that the best systematic method is little improvement over the worst. Hamlet [5] corroborates this result using a different sampling model. He shows random testing to be superior to partition testing, its superiority increasing with more partitions and with the program confidence required.
Partition Testing Does Not Inspire Confidence

Dick Hamlet, Member, IEEE, and Ross Taylor

This study was undertaken because partition testing did not live up to its intuitive value in two earlier studies. In their brief for random testing [3], Duran and Ntafos published a precise comparison between it and partition testing. Their surprising result is that the two methods are of almost equal value, under assumptions that seem to favor partition testing. Random testing has a decidedly spotty reputation, probably because it makes almost no use of special information about the program being tested. It is certainly counterintuitive that the best systematic method is little improvement over the worst. Hamlet [5] corroborates this result using a different sampling model. He shows random testing to be superior to partition testing, its superiority increasing with more partitions and with the program confidence required.
Blackbox Fuzzing: **Super fast!**

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    
    if(crash == 1) abort();
}
```

- Whitebox Fuzzer: Discovers the bug after **3 inputs**, in expectation.
- Blackbox Fuzzer: Discovers the bug after \(((1/256)^4)^{-1} \approx 4 \text{ billion inputs}\), in expectation.
Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.

Blackbox Fuzzer: Discovers the bug after \((1/256)^4\)\(^{-1} \approx 4\) billion inputs, in expectation.

If our whitebox fuzzer takes too long per input, our blackbox fuzzer outperforms!

» There is a maximum time per test input!

```
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if (crash == 1) abort();
}
```

• Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.
• Blackbox Fuzzer: Discovers the bug after \((1/256)^4\)\(^{-1} \approx 4\) billion inputs, in expectation.

On my machine, this takes 6.3 seconds.
On 100 machines, it takes 63 milliseconds.

Blackbox Fuzzing: Super fast!
Blackbox Fuzzing: **Super fast!**

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;

    if(crash == 1) abort();
}
```

- Whitebox Fuzzer: Discovers the bug after **3 inputs**, in expectation.
- Blackbox Fuzzer: Discovers the bug after \((1/256)^4\)\(^{-1}\) \(\approx 4\) billion inputs, in expectation.
Blackbox Fuzzing: Super fast!

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if (crash == 1) abort();
}
```

- Whitebox Fuzzer: Discovers the bug after **3 inputs**, in expectation.
- Blackbox Fuzzer: Discovers the bug after \(\left(\frac{1}{256}\right)^4 \approx 4 \text{ billion inputs}\), in expectation. On my machine, this takes **6.3 seconds**. On 100 machines, it takes **63 milliseconds**.

If our whitebox fuzzer takes too long per input, our blackbox fuzzer outperforms!» There is a maximum time per test input!
• Our model: Error-based partitioning
  • EITHER all inputs in a partition do reveal a bug
    OR all inputs in a partition do not reveal a bug

The most effective testing technique
samples from error-based partitions!

— Weyuker and Jeng’91

[FSE’14] On the Efficiency of Automated Testing, M Böhme, S. Paul,
Bounds on Fuzzing Efficiency

Our model: Error-based partitioning

- EITHER all inputs in a partition do reveal a bug
  OR all inputs in a partition do not reveal a bug

- However, we have no a-priori knowledge whether a partition is error-revealing.
- Partitions are of arbitrary size and number.

The most effective testing technique samples from error-based partitions!

— Weyuker and Jeng’91

[FSE’14] On the Efficiency of Automated Testing, M Böhme, S. Paul,
Bounds on Fuzzing Efficiency

• Our model: Error-based partitioning
  • EITHER all inputs in a partition do reveal a bug
    OR all inputs in a partition do not reveal a bug.

• A testing technique samples the program’s input space and
  discovers a partition $D_i$ when $D_i$ is sampled for the first time.

• The discovery of $D_i$ shows whether or not $D_i$ reveals a bug.
  • Notice that we assume a test oracle.

[FSE’14] On the Efficiency of Automated Testing, M Böhme, S. Paul,
Bounds on Fuzzing Efficiency

• **Achieving confidence**: Whoever can show *first* that the program works correctly for x% of its inputs wins.

• A testing technique achieves a degree of confidence x when at least x% of the program inputs reside in discovered partitions.
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for \(x\%\) of its inputs wins.

[FSE'14] On the Efficiency of Automated Testing, M Böhme, S. Paul,
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.

- **Blackbox Fuzzing (R)**
  - Samples inputs randomly
  - Some partitions several times, others not at all
  - 1 time unit per input

- **Whitebox Fuzzing (S)**
  - Samples inputs systematically
  - Each partition exactly once!
  - Most effective!
  - c time units per input

[FSE'14] On the Efficiency of Automated Testing, M Böhme, S. Paul,
Bounds on Fuzzing Efficiency

- Achieving confidence: Whoever can show first that the program works correctly for x% of its inputs wins.
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for $x\%$ of its inputs wins.

![Graph showing expected time to achieve 100% confidence in whitebox fuzzing.](graph.png)

- Expected to achieve 100% in $c \cdot k$ time units.

- **Whitebox Fuzzing**
Bounds on Fuzzing Efficiency

• **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.
• **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.

![Graph showing bounds on fuzzing efficiency]
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for $x\%$ of its inputs wins.
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.

![Graph showing increasing confidence achieved with time, with annotations: Increasing c of s, increases time to achieve same degree of confidence!](graph.png)
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.

Blackbox Fuzzing wins — if the average time to sample one test input exceeds c₀.
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.

- **Answer**: S is expected to lose if $c > \frac{1}{ex - ex^2}$ time units.

- **Example**:
  - R takes **1ms** to sample one test input
  - Establish correctness for x=90% of inputs
    - S₀ must take less than **4.1ms** to sample one test input
    - Otherwise, R is expected to achieve the 90%-degree of confidence first.

---

[FSE'14] On the Efficiency of Automated Testing, M Böhme, S. Paul,
Bounds on Fuzzing Efficiency

- **Achieving confidence:** Whoever can show first that the program works correctly for x% of its inputs wins.

- **Answer:** S is expected to lose if \( c > \frac{1}{ex - ex^2} \) time units.

- **Example:**
  - R takes 1ms to sample one test input
  - Establish correctness for \( x=99.9\% \) of inputs
    - \( S_0 \) must take less than 370ms to sample one test input
    - Otherwise, R is expected to achieve the 99.9%-degree of confidence first.

[FSE’14] On the Efficiency of Automated Testing, M Böhme, S. Paul,
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for x% of its inputs wins.

- **Answer**: $S$ is expected to lose if $C > \frac{1}{e^x - e^{2x}}$ time units.

- **Example**:
  - $R$ takes $1\text{ms}$ to sample one test input
  - Establish correctness for $x=99.9\%$ of inputs
    - $S_0$ must take less than $370\text{ms}$ to sample one test input
    - Otherwise, $R$ is expected to achieve the 99.9%-degree of confidence first.

[FSE'14] On the Efficiency of Automated Testing, M Böhme, S. Paul
Bounds on Fuzzing Efficiency

- **Achieving confidence**: Whoever can show first that the program works correctly for $x\%$ of its inputs wins.

- **Answer**: $S$ is expected to lose if $c > \frac{1}{e^x - e^x^2}$ time units.

- **Example**:
  - $R$ takes 1ms to sample one test input
  - Establish correctness for $x=99.9\%$ of inputs
    - $S_0$ must take less than 370ms to sample one test input
    - Otherwise, $R$ is expected to achieve the 99.9%-degree of confidence first.

---

[FSE'14] On the Efficiency of Automated Testing, M Böhme, S. Paul,
Bounds on Fuzzing Efficiency

- **Achieving confidence:** Whoever can show first that the program works correctly for x% of its inputs wins.

- **Answer:** $S$ is expected to lose if $c > \frac{1}{e^x - e^{2x}}$ time units.

- **Example:**
  - **R** takes $1\text{ms}$ to sample one test input
  - Establish correctness for $x=99.9\%$ of inputs
  - $S_0$ must take less than $370\text{ms}$ to sample one test input
  - Otherwise, **R** is expected to achieve the 99.9%-degree of confidence first.

---

[FSE’14] On the Efficiency of Automated Testing, M Böhme, S. Paul,
Bounds on Fuzzing Efficiency

• **Our insight:** Even the most effective fuzzing technique is less efficient than blackbox fuzzing if generating a test takes relatively too long.

• We shed light on a 40 year old riddle and demonstrate a fundamental limitation of whitebox fuzzing.
  (including grammar-based whitebox fuzzing)

[FSE'14] *On the Efficiency of Automated Testing*, M Böhme, S. Paul,
Blackbox Fuzzing: **Super fast!**

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if (crash == 1) abort();
}
```

- Whitebox Fuzzer: Discovers the bug after **3 inputs**, in expectation.
- Blackbox Fuzzer: Discovers the bug after **4 billion inputs**, in expectation.

So, if we have sufficiently many machines (to maximize execs/sec), blackbox fuzzers are the best we can get, right?
Blackbox Fuzzing: Super fast!

void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;

    if(crash == 1) abort();
}

- Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.
- Blackbox Fuzzer: Discovers the bug after 4 billion inputs, in expectation.

So, if we have sufficiently many machines (to maximize execs/sec), blackbox fuzzers are the best we can get, right? Wrong.
Blackbox Fuzzing: Super fast!

void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if(crash == 1) abort();
}

• Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.
• Generational Blackbox Fuzzer: Discovers the bug after 4 billion inputs, in expectation.

So, if we have sufficiently many machines (to maximize execs/sec), blackbox fuzzers are the best we can get, right? Wrong.
Blackbox Fuzzing: Super fast!

void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;

    if(crash == 1) abort();
}

• Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.
• **Generational** Blackbox Fuzzer: Discovers the bug after 4 billion inputs, in expectation.
• **Mutational** Blackbox Fuzzer mutates a random character in a seed.

So, if we have sufficiently many machines (to maximize execs/sec), blackbox fuzzers are the best we can get, right? **Wrong.**
Blackbox Fuzzing: Super fast!

void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if(crash == 1) abort();
}

- Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.
- **Generational** Blackbox Fuzzer: Discovers the bug after 4 billion inputs, in expectation.
- **Mutational** Blackbox Fuzzer mutates a random character in a seed.
  - Started with the seed bad?
  - Discovers the bug after \(((4^{-1})^8)^{-1} \approx 1024\) inputs, in expectation.

So, if we have sufficiently many machines (to maximize execs/sec), blackbox fuzzers are the best we can get, right? Wrong.
Blackbox Fuzzing: **Super fast!**

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;

    if(crash == 1) abort();
}
```

- Whitebox Fuzzer: Discovers the bug after **3 inputs**, in expectation.
- **Generational** Blackbox Fuzzer: Discovers the bug after **4 billion inputs**, in expectation.
- **Mutational** Blackbox Fuzzer mutates a random character in a seed.
  - Started with the seed **bad**?
  - Discovers the bug after \((4^{-1})(2^{-8}))^{-1} \approx 1024\ inputs\), in expectation.

So, if we have sufficiently many machines (to maximize execs/sec), blackbox fuzzers are the best we can get, right? **Wrong.**
Blackbox Fuzzing: Super fast!

void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if(crash == 1) abort();
}

• Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.
• Generational Blackbox Fuzzer: Discovers the bug after 4 billion inputs, in expectation.
• Mutational Blackbox Fuzzer mutates a random character in a seed.
  • Started with the seed bad?
  • Discovers the bug after \((4^{-1})*(2^{-8})^{-1}\) \(\approx\) 1024 inputs, in expectation.

So, if we have sufficiently many machines (to maximize execs/sec), blackbox fuzzers are the best we can get, right? Wrong.
Blackbox Fuzzing: **Super fast!**

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if (crash == 1) abort();
}
```

- Whitebox Fuzzer: Discovers the bug after **3 inputs**, in expectation.
- **Generational** Blackbox Fuzzer: Discovers the bug after **4 billion inputs**, in expectation.
- **Mutational** Blackbox Fuzzer mutates a random character in a seed.
  - Started with the seed **bad**?
  - Discovers the bug after \((4^{-1})(2^{-8})^{-1} \approx 1024\) **inputs**, in expectation.

So, if we have sufficiently many machines (to maximize execs/sec), **blackbox fuzzers** are the best we can get, right? **Wrong.**
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if (crash == 1) abort();
}

• **Greybox Fuzzing**: Add generated inputs to the corpus which **increase coverage**!
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if (crash == 1) abort();
}

- **Greybox Fuzzing**: Add generated inputs to the corpus which *increase coverage*!

---

**Seed corpus**

<table>
<thead>
<tr>
<th>“Interesting” Input</th>
<th>Expected #inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>**** ba**</td>
<td>((1/2 \times 4^{-1} \times 2^{-8})^{-1})</td>
</tr>
<tr>
<td>b***</td>
<td>((1/3 \times 4^{-1} \times 2^{-8})^{-1})</td>
</tr>
<tr>
<td>bad*</td>
<td>((1/4 \times 4^{-1} \times 2^{-8})^{-1})</td>
</tr>
<tr>
<td>Total: 10240</td>
<td></td>
</tr>
</tbody>
</table>
Greybox Fuzzing: “Enumerate”

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;

    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;

    if (crash == 1) abort();
}
```

- **Greybox Fuzzing**: Add generated inputs to the corpus which increase coverage!
Greybox Fuzzing: “Enumerate”

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if(crash == 1) abort();
}
```

- **Greybox Fuzzing**: Add generated inputs to the corpus which **increase coverage**!

- **Greybox Fuzzing** started only with **** in the seed corpus discovers the bug after 10k inputs (in 150 microseconds)!

---

[CCS’16] Coverage-based Greybox Fuzzing as Markov Chain
M Böhme, V.T. Pham, A. Roychoudhury (extended in IEEE TSE journal)
Greybox Fuzzing: "Enumerate"

- **Greybox Fuzzing**: Add generated inputs to the corpus which increase coverage!

- Greybox Fuzzing started only with **** in the seed corpus discovers the bug after 10k inputs (in 150 microseconds)!

- **Boosted** Greybox Fuzzing started with **** in the seed corpus discovers the bug after 4k inputs (in 55 microseconds)!

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| **** | b*** | $(1 \times 4^{-1} \times 2^{-8})^{-1}$  
      |      | = 1024             |
| **** | ba**  | $(1 \times 4^{-1} \times 2^{-8})^{-1}$  
       |      | = 1024             |
| **** | bad*  | $(1 \times 4^{-1} \times 2^{-8})^{-1}$  
      |      | = 1024             |
| **** | bad!  | $(1 \times 4^{-1} \times 2^{-8})^{-1}$  
      |      | = 1024             |
|     |     | **Total: 4096**    |

[CCS’16] Coverage-based Greybox Fuzzing as Markov Chain  
M Böhme, V.T. Pham, A. Roychoudhury (extended in IEEE TSE journal)
More Machines!

Awesome! We have a really efficient fuzzers. Let's throw more machines at the problem!

- Blackbox Fuzzer: Discovers the bug after $(1/256)^4\cdot 1 \approx 4 \text{ billion inputs}$, in expectation.
  - On my machine, this takes 6.3 seconds.
  - On 100 machines, it takes 63 milliseconds.
More Machines!

X times more machines means X times more bugs, right?

• Blackbox Fuzzer: Discovers the bug after \((1/256)^4\)^{-1} \approx 4 \text{ billion inputs}, in expectation.
  On my machine, this takes \textbf{6.3 seconds}.
  On 100 machines, it takes \textbf{63 milliseconds}.
More Machines!

X times more machines means
X times more bugs, right?

Wrong.

- Blackbox Fuzzer: Discovers the bug after $((1/256)^4)^{-1} \approx 4 \text{ billion inputs}$, in expectation.
  On my machine, this takes 6.3 seconds.
  On 100 machines, it takes 63 milliseconds.
Scalability

• **300+ OSS** projects (OSSFuzz & Fuzzer-Test-Suite)
• **6 measures** of code coverage (4) and bug finding effectiveness (2)
• **4+ CPU years** worth of fuzzing campaigns
• **2 fuzzers** (AFL and LibFuzzer)

**Open Science and Reproducibility**
• Reproduce our results under other circumstances.
• Inspect our data and simulation @ Kaggle (Jupyter Notebook).
• Modify parameters in our simulation and analysis.

[FSE’20] Fuzzing: On the Exponential Cost of Vulnerability Discovery. M. Böhme, Brandon Falk (Microsoft)
Scalability

- #Machines
  - An abstraction of the \textit{#inputs the fuzzer can generate per minute}.
  - Example: Twice the #machines can generate twice #inputs per minute.
  - We assume \textbf{no synchronisation overhead}. For greybox fuzzers, new seeds immediately available to all fuzzers.
  - We use this definition for \textbf{data scaling}.

\[\text{[FSE'20] Fuzzing: On the Exponential Cost of Vulnerability Discovery.}
\text{M. Böhme, Brandon Falk (Microsoft)}\]
Figure 1: Each vuln. discovery requires exponentially more machines (left). Yet, exponentially more machines allow to find the same vulnerabilities exponentially faster (right).

Fuzzer Test Suite (45min campaigns)
Number of Additional Vulns Discovered

Figure 2: (#Crashes, #Features, #Edges @ OSS-Fuzz). Average number of additional species discovered when fuzzing all 263 programs in OSS-Fuzz simultaneously with LIBFUZZER for 45 minutes as a function of available machines (4 reps).
Number of Additional Vulns Discovered

(b) #Features and #Edges @ FTS. Average number of additional number of features / edges covered when fuzzing these 12 programs in FTS with LibFuzzer for 45 minutes, as the number of available machines increases (20 repetitions).
Given the same non-deterministic fuzzer, discovering linearly more new species within the same time budget, requires exponentially more inputs per minute.
Figure 6: Probability that the vulnerability has been discovered in twenty seconds given the available number of machines (solid line). Average number of machines required to find the vulnerability in twenty seconds (dashed line).
Probability to Discover Given Vulnerability

Figure 6: Probability that the vulnerability has been discovered in twenty seconds given the available number of machines (solid line). Average number of machines required to find the vulnerability in twenty seconds (dashed line).
Probability to Discover Given Vulnerability

**Figure 8:** Probability $Q_{\exp}(x) = S(2^x n)$ to discover a given species within a given time budget as the number of machines increases exponentially (solid line).
Probability to Discover Given Vulnerability

![Graph showing the probability of discovering a vulnerability given a certain number of machines. The graph is divided into two parts: one showing the discovery probability on a linear scale, and the other showing the same data on a logarithmic scale. The inflection point is marked on both graphs.](Image)
Between origin and inflection point, an exponential curve grows slower than discovery probability!
Probability to Discover Given Vulnerability

• **What does that mean** for a non-deterministic fuzzer?
What does that mean for a non-deterministic fuzzer?

• The probability of exposing a specific known vulnerability,

• the probability of reaching a specific program statement,

• the probability of violating a specific program assertion, etc.

• within a given time budget increases approximately linearly with the number of available machines — up to a certain limit.
What does that mean for a non-deterministic fuzzer?

- The probability of exposing all (known) vulnerabilities,
- the probability of reaching all program statements,
- the probability of violating all program assertions, etc.
- within a given time budget increases approximately linearly with the number of available machines — up to a certain limit.
Explaining to Exponential Cost

Figure 10. Number of additional species discovered in a fixed time budget as the number of machines increases (5 random samples of $\{q_i\}_{i=1}^S$ each.)
Explaining to Exponential Cost

Figure 10. Number of additional species discovered in a fixed time budget as the number of machines increases (5 random samples of $\{q_i\}^S_{i=1}$ each.
Explaining to Exponential Cost

Figure 10. Number of additional species discovered in a fixed time budget as the number of machines increases (5 random samples of $\{q_i\}_{i=1}^S$ each.
Figure 10. Number of additional species discovered in a fixed time budget as the number of machines increases (5 random samples of $\{q_i\}_{i=1}^{S}$ each.

Explaining to Exponential Cost
Explaining to Exponential Cost

Figure 10. Number of additional species discovered in a fixed time budget as the number of machines increases (5 random samples of $\{q_i\}_{i=1}^S$ each.)
Intuitively, each new vulnerability requires some more resources (time or machines) than the previous vulnerability.
On the Cost of Vulnerability Discovery

A constant rate of vulnerability discovery requires exponential amount of resources.

*This is a fundamental limitation of fuzzing!
Whitebox Fuzzing: Most Effective!

```c
void crashme (char s0, char s1, char s2, char s3) {
  int crash = 0;
  if (s0 == 'b')
    if (s1 == 'a')
      if (s2 == 'd')
        if (s3 == '!')
          crash = 1;
  if (crash == 1) abort();
}
```

It can prove the absence of assertion violation, by enumerating all paths and modulo some assumptions.

Path Conditions

✓ φ₁ = (s0 != 'b')
✓ φ₂ = (s0 == 'b') /\ (s1 != 'a')
✓ φ₃ = (s0 == 'b') /\ (s1 == 'a') /\ (s2 != 'd')
✓ φ₄ = (s0 == 'b') /\ (s1 == 'a') /\ (s2 == 'd') /\ (s3 != '!')
✗ φ₅ = (s0 == 'b') /\ (s1 == 'a') /\ (s2 == 'd') /\ (s3 == '!')

Blackbox Fuzzing: Super fast!

```c
void crashme (char s0, char s1, char s2, char s3) {
  int crash = 0;
  if (s0 == 'b')
    if (s1 == 'a')
      if (s2 == 'd')
        if (s3 == '!')
          crash = 1;
  if (crash == 1) abort();
}
```

If our whitebox fuzzer takes too long per input, our blackbox fuzzer outperforms!

» There is a maximum time per test input!

• Whitebox Fuzzer: Discovers the bug after 3 inputs, in expectation.
• Blackbox Fuzzer: Discovers the bug after \((1/256)^4 \cdot 2^{-8}\) ≈ 4 billion inputs, in expectation.

On my machine, this takes 6.3 seconds.
On 100 machines, it takes 63 milliseconds.

Greybox Fuzzing: “Enumerate”

• Greybox Fuzzing: Add generated inputs to the corpus which increase coverage!

• Greybox Fuzzing started only with **** in the seed corpus discovers the bug after 10k inputs (in 150 microseconds)!

• Boosted Greybox Fuzzing started with **** in the seed corpus discovers the bug after 4k inputs (in 55 microseconds)!

<table>
<thead>
<tr>
<th></th>
<th>b***</th>
<th>b****</th>
<th>b*****</th>
<th>ba**</th>
<th>bad*</th>
<th>ba***</th>
<th>ba**</th>
<th>bad*</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1\cdot4^{-1}\cdot2^{-9})^{-1}</td>
<td>(1\cdot4^{-1}\cdot2^{-9})^{-1}</td>
<td>(1\cdot4^{-1}\cdot2^{-9})^{-1}</td>
<td>(1\cdot4^{-1}\cdot2^{-9})^{-1}</td>
<td>(1\cdot4^{-1}\cdot2^{-9})^{-1}</td>
<td>(1\cdot4^{-1}\cdot2^{-9})^{-1}</td>
<td>(1\cdot4^{-1}\cdot2^{-9})^{-1}</td>
<td>(1\cdot4^{-1}\cdot2^{-9})^{-1}</td>
<td>4096</td>
</tr>
</tbody>
</table>

Total: 4096

Exponential Cost of Vulnerability Discovery

![Figure 10. Number of additional species discovered in a fixed time budget as the number of machines increases (5 random samples of \(q_i\) for each.)](image-url)
Whitebox Fuzzing: Most Effective!

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if(crash == 1) abort();
}
```

It can prove the absence of assertion violation, by enumerating all paths and modulo some assumptions.

Blackbox Fuzzing: Super fast!

```c
void crashme (char s0, char s1, char s2, char s3) {
    int crash = 0;
    if (s0 == 'b')
        if (s1 == 'a')
            if (s2 == 'd')
                if (s3 == '!')
                    crash = 1;
    if(crash == 1) abort();
}
```

If our whitebox fuzzer takes too long per input, our blackbox fuzzer outperforms!

> There is a maximum time per test input!

Greybox Fuzzing: "Enumerate"

- Greybox Fuzzing: Add generated inputs to the corpus which increase coverage!
- Greybox Fuzzing started only with **** in the seed corpus discovers the bug after 10k inputs (in 150 microseconds)!
- Boosted Greybox Fuzzing started with **** in the seed corpus discovers the bug after 4k inputs (in 55 microseconds)!

If you want to take a deeper dive:

* Read our interactive text book: The Fuzzing Book
* Read our IEEE Software article: “Fuzzing: Challenges and Reflections”
* Apply for PhD / PostDoc in my group at MPI-SP, Bochum, Germany.

Web: [https://mboehme.github.com](https://mboehme.github.com)  Twitter: [@mboehme](https://twitter.com/mboehme)