Dependency-aware Residual Risk Analysis

Seongmin Lee (UCLA) and Marcel Böhme (MPI-SP)

UCLA

MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

Fundamental Question of Fuzzing

It doesn't tell you the

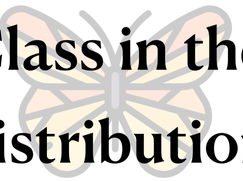
Absence of a bug.

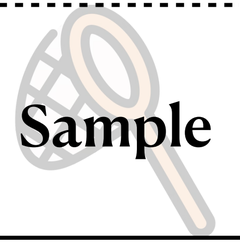
There can always be
unseen behavior.

Q. When can we stop fuzzing?

Prior work on Residual Risk Analysis

Ecology-based Statistical Software Testing [1]		
Fuzzing	$\Rightarrow$	Sampling Process
Program element/Bug	$\Rightarrow$	Class in the distribution
Test Execution	$\Rightarrow$	Sample





Residual risk :=

Probability of the next test execution finding a new bug

$$\leq$$

Discovery probability (DP) :=

Probability of the next test execution finding anything new (bug or coverage)

Stop the fuzzing if

Estimated DP $\leq \delta$ (**threshold**, e.g., 10^{-6})

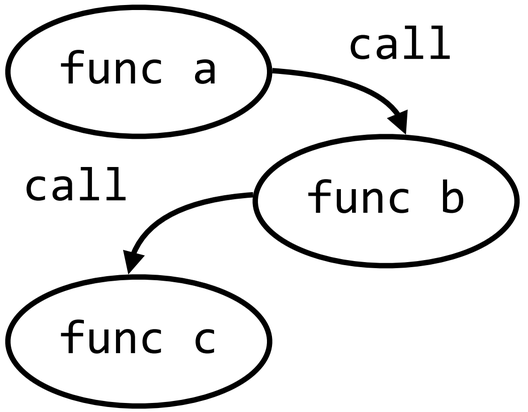
[1] Marcel Böhme. 2018. STADS: Software Testing as Species Discovery. TOSEM

Limitation of the Existing Residual Risk Analysis

- Their estimator assumes **independence** between program elements. ⇐ **NOT TRUE!**

```
...
if (pred)
  stmt;
...
```

'`stmt`' is executed only if '`if`' is executed.



There are long call-chains in the program.

Using a **dependency-agnostic** estimator led to considerable **overestimation**.

⇒ **Wrongly thinks** there's still a **high chance** of finding an unseen bug.

🙁 ⇒ **Waste of resources** on an already saturated testing campaign.

Q. How can we **accurately estimate**
the residual risk considering
dependencies between program elements?

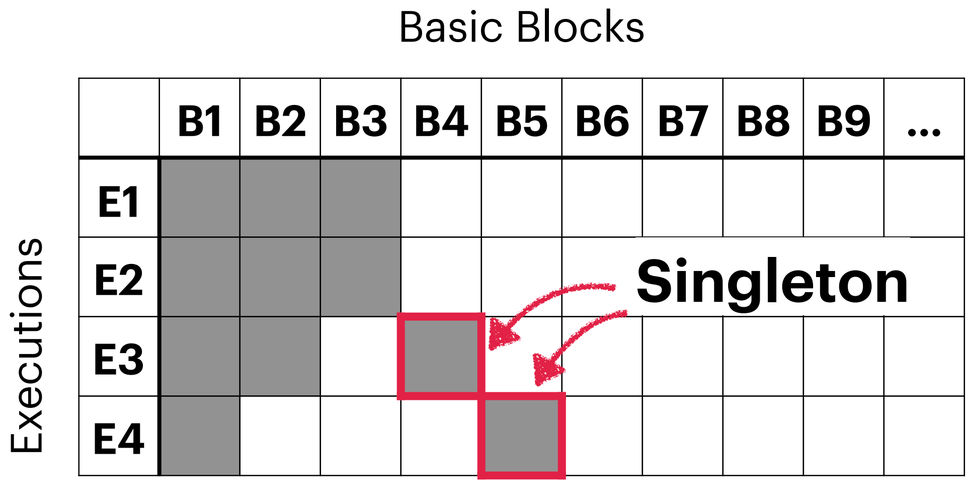
Existing estimators

(Dependency-agnostic)

— Good-Turing estimator [2] —

$$\hat{DP}_{Good} = \frac{\text{\# of basic blocks observed only once} \;:=\; \textbf{Singleton}}{\text{\# test execution}}$$

Coverage Matrix



$$\hat{DP}_{Good} = \frac{2}{4} = 0.5$$

[2] Good, Irving J. "The population frequencies of species and the estimation of population parameters." *Biometrika* 40.3-4 (1953): 237-264.

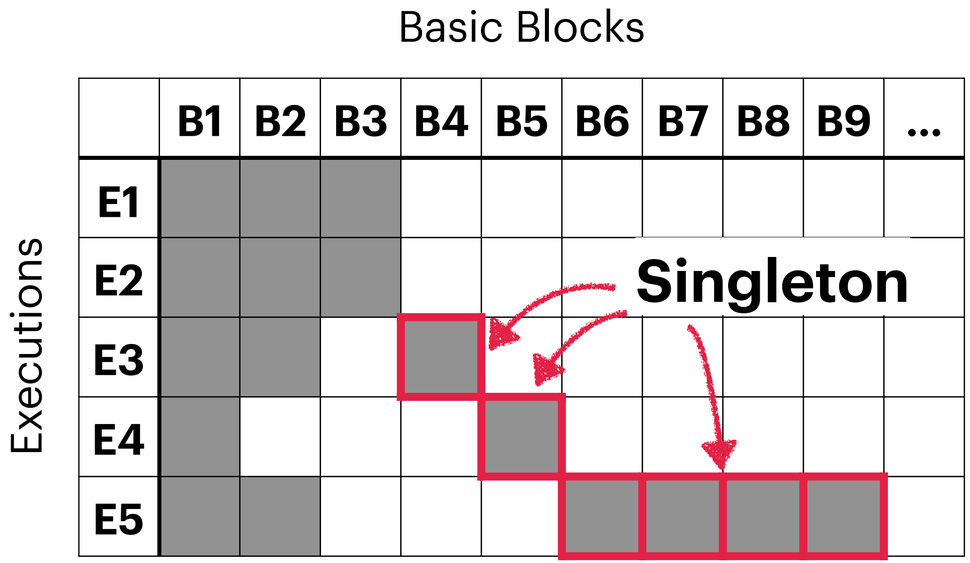
Existing estimators

(Dependency-agnostic)

— Good-Turing estimator [2] —

$$\hat{DP}_{Good} = \frac{\#\text{ of basic blocks observed only once } := \textbf{Singleton}}{\#\text{ test execution}}$$

Coverage Matrix



$$\hat{DP}_{Good} = \frac{6}{5} > 1$$

Problem:

A large number of singletons can appear together due to program dependencies.

$\Rightarrow$ **significant overestimation**

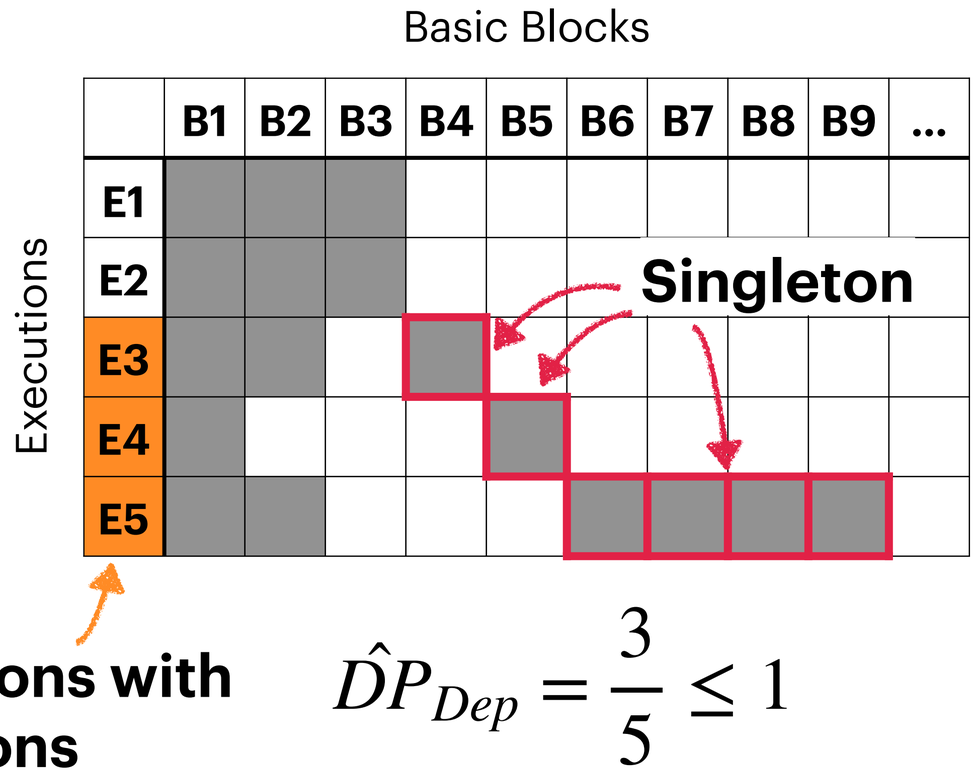
Existing estimators

— Dependency-aware estimator [3] —

$$\hat{DP}_{Dep} = \frac{\#\ \textbf{of executions with singleton}}{\#\ \text{test execution}}$$

(Many dependent singletons $\Rightarrow$ one execution)

Coverage Matrix



$$\hat{DP}_{Dep} = \frac{3}{5} \leq 1$$

[3] Ma, M-C., and Anne Chao. "Generalized sample coverage with an application to Chinese poems." *Statistica Sinica* (1993): 19-34.

Existing estimators

— Dependency-aware estimator [3] —

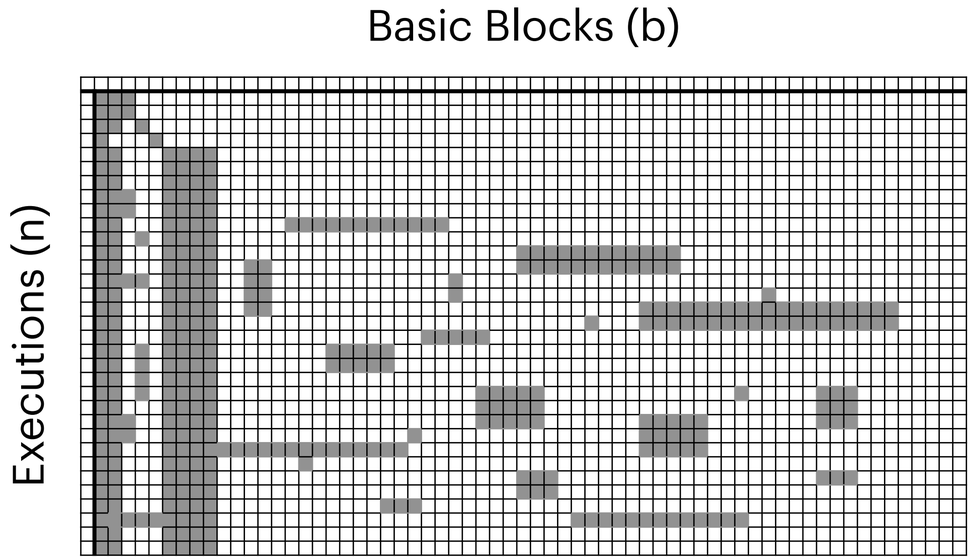
$$\hat{DP}_{Dep} = \frac{\#\text{ of executions with singleton}}{\#\text{ test execution}}$$

Problem:

Naïvely computing $\hat{DP}_{Dep}$ requires a **huge space** $O(n \times b)$, where

- n: # of test executions so far
- b: # of basic blocks in the program

Coverage Matrix (size: $n \times b$)



E.g., TCPDUMP & LIBPCAP, *24 hours of fuzzing*

Number of basic blocks (b)	*Number of exec. after 24 hours (n)*	*b X n*
14,355	467,938,080	~6 trillion

Our Solution: Two insights for efficient dependency-aware estimation

Insight 1:
No need to monitor all basic blocks

Optimization 1

Insight 2:
No need to record execution history

Optimization 2

Make a dependency-aware residual risk analysis feasible to apply at the scale of fuzzing.

🎯 What we want to know: **Which execution covers singletons?**

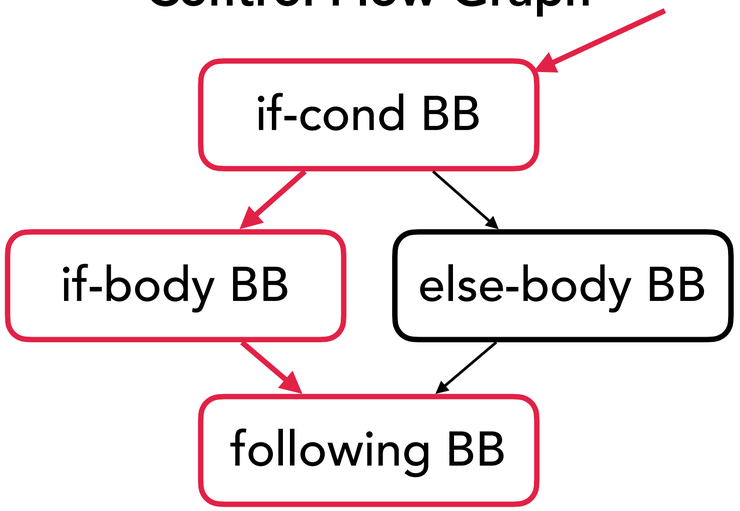
Insight 1: No need to monitor all basic blocks

Example.

```
if (p)
    stmt1;
else
    stmt2;
stmt3;
```

Code

Control Flow Graph



1. Execution $\mathbf{E}$ is the only execution covering if-cond BB.

* These basic blocks are singletons.

2. if-body BB already tells you $\mathbf{E}$ covers a singleton w/o help from if-cond BB.

⇒ No need to monitor if-cond BB.

If we use the **dominance relationship** in the **CFG**, we can **safely ignore** basic blocks for monitoring while still accurately determining which execution covers singletons.

🎯 What we want to know: **Which execution covers singletons?**

Insight 1: No need to monitor all basic blocks

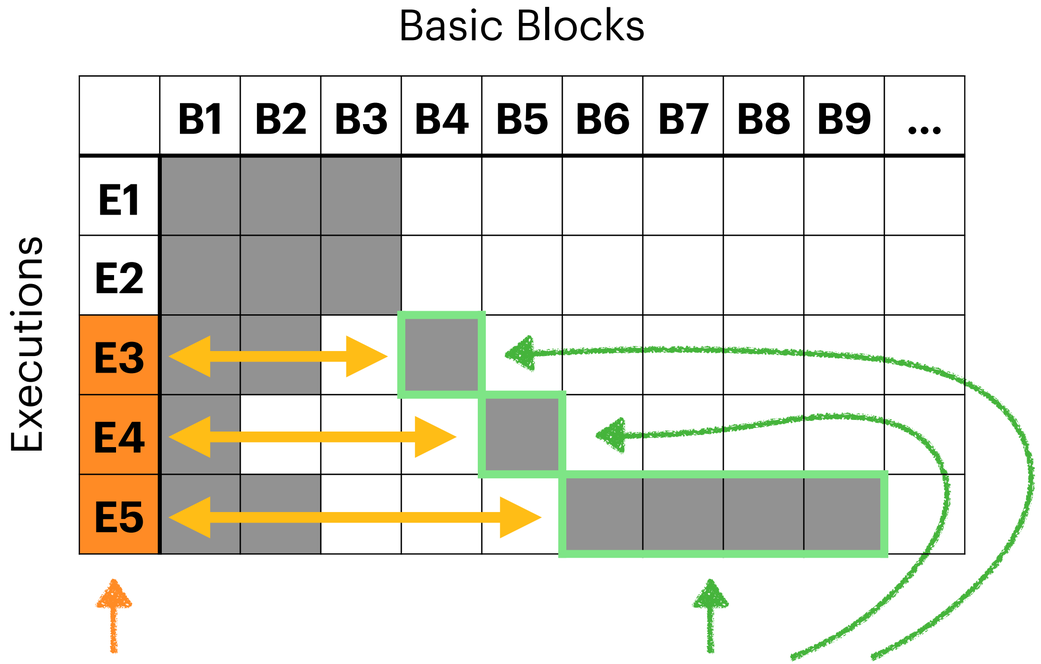
We formally define which basic blocks are **safely ignorable** from monitoring and prove that this is sound.

> THEOREM 1. If a basic block B dominates (or post-dominates) all of its successors (or predecessors, respectively) in the CFG G, then discarding B from monitoring does not change $\hat{DP}_{Dep}$.

In the experiment, **roughly half of the basic blocks need no monitoring**.

🎯 What we want to know: **The number of executions covering singletons**

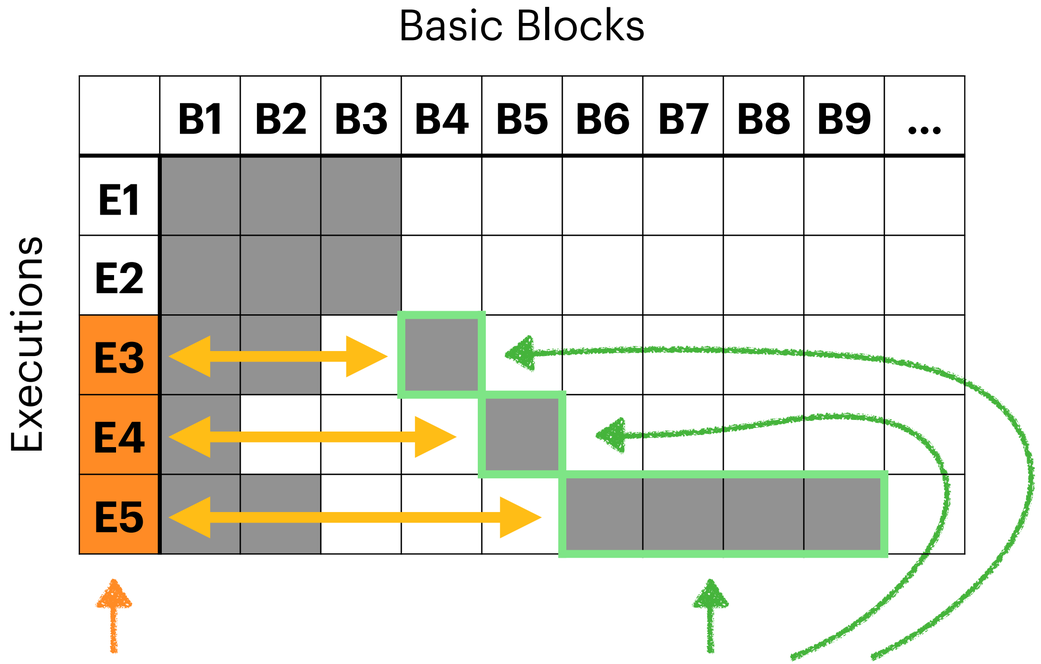
Insight 2: No need to record coverage history



Executions with singletons = **Singleton Clusters** := **set of singletons covered together in the same execution.**

🎯 What we want to know: **The number of executions covering singletons**

Insight 2: No need to record coverage history



$O(n \times b)$

Executions with singletons

=

Singleton Clusters := **set of singletons covered together in the same execution.**

$O(b)$

Space-Efficient Algorithm for Counting Singleton Clusters

- Key idea: keep **maintaining the current singleton clusters** after each test execution.

Algorithm 1: Online singleton cluster maintenance

Input: X^n: the stream of samples
Output: $V_1^{\equiv}$: singleton cluster set

$S_n \leftarrow \emptyset$; $V_1 \leftarrow \emptyset$; $V_1^{\equiv} \leftarrow \emptyset$
for $X_i \in X^n$ **do** // iterate over the stream of samples
  $B \leftarrow X_i \cap V_1$; // observed class set B
  $V_1 \leftarrow V_1 \setminus B$; // remove classes in B from V_1 and $V_1^{\equiv}$
  for $V_1^j \in V_1^{\equiv}$ **do**
    $V_1^j \leftarrow V_1^j \setminus B$
  $D \leftarrow X_i \setminus S_n$; // the newly observed classes D
  if $D \neq \emptyset$ **then** // Add D to S_n, V_1, and $V_1^{\equiv}$
    $S_n \leftarrow S_n \cup D$; $V_1 \leftarrow V_1 \cup D$; $V_1^{\equiv} \leftarrow V_1^{\equiv} \cup \{D\}$
return $V_1^{\equiv}$

Steps

- Update the existing singleton cluster by dropping basic blocks seen more than once
- Create a new singleton cluster with newly found basic blocks
- Space complexity: $O(b)$

Optimization 1.
Dropping safely ignorable basic blocks

Optimization 2.
Maintaining clusters of singletons

Implement

Dependency-aware estimator $\hat{DP}_{Dep}$ feasible to apply at scale in fuzzing.

Evaluation

RQ1. Efficiency of Optimization

How much does **dropping safely ignorable basic blocks reduce the cost** of dependency-aware estimation?

Metrics:

1. Amount of ignorable basic blocks
2. Compile time speed-up
3. Fuzzing speed-up

dependency aware | dependency agnostic

RQ2. Accuracy of $\hat{DP}_{Dep}$ vs $\hat{DP}_{Good}$

How much **more accurate** is the dependency-aware estimation in residual risk analysis?

Compares

[Estimated discovery probability]

vs

[Residual risk]

Result 1. Efficiency of Optimization

Subject	#(BB)	#(BB_{drop})	Ratio
Sqlite3	58,253	32,849	0.56
Freetype2	46,051	26,553	0.58
Libxml2	93,858	50,573	0.54
Libjpeg	36,840	21,788	0.59
Zlib	1,775	986	0.56
Libpcap	14,355	7,815	0.54
Jsoncpp	8,780	5,631	0.64
Libpng	10,463	6,077	0.58
Avg.			0.57

Dropping safely ignorable basic blocks

significantly reduces (54-64%) the number of basic blocks to monitor.

Result 1. Efficiency of Optimization

Subject	#(BB)	#(BB_{drop})	Ratio	$\Delta(T_{compile})$
Sqlite3	58,253	32,849	0.56	-82.6s (-29%)
Freetype2	46,051	26,553	0.58	-7.6s (-9%)
Libxml2	93,858	50,573	0.54	-5.7s (-11%)
Libjpeg	36,840	21,788	0.59	-9.9s (-18%)
Zlib	1,775	986	0.56	-0.6s (-13%)
Libpcap	14,355	7,815	0.54	-8.9s (-15%)
Jsoncpp	8,780	5,631	0.64	-3.5s (-9%)
Libpng	10,463	6,077	0.58	-2.7s (-10%)
Avg.			0.57	-15.2s (-14%)

The compilation time was also reduced by 14%.

- Smaller number of basic blocks to monitor

  $\Rightarrow$ less coverage instrumentation load

* *Note: this compilation time includes the dominance analysis*

Result 1. Efficiency of Optimization

Subject	Ratio	$\mu(eps)$	$\mu(eps_{rm})$	$\mu\left(\frac{eps_{rm}}{eps}\right)$
Sqlite3	0.56	21.93	31.43	1.43
Freetype2	0.58	51.06	104.05	2.04
Libxml2	0.54	18.82	50.17	2.67
Libjpeg	0.59	290.40	1061.58	3.66
Zlib	0.56	7043.66	7683.91	1.09
Libpcap	0.54	5415.95	5664.24	1.05
Jsoncpp	0.64	2686.26	3370.08	1.25
Libpng	0.58	2210.39	3140.78	1.42
Avg.	0.57			1.83

The fuzzing itself became

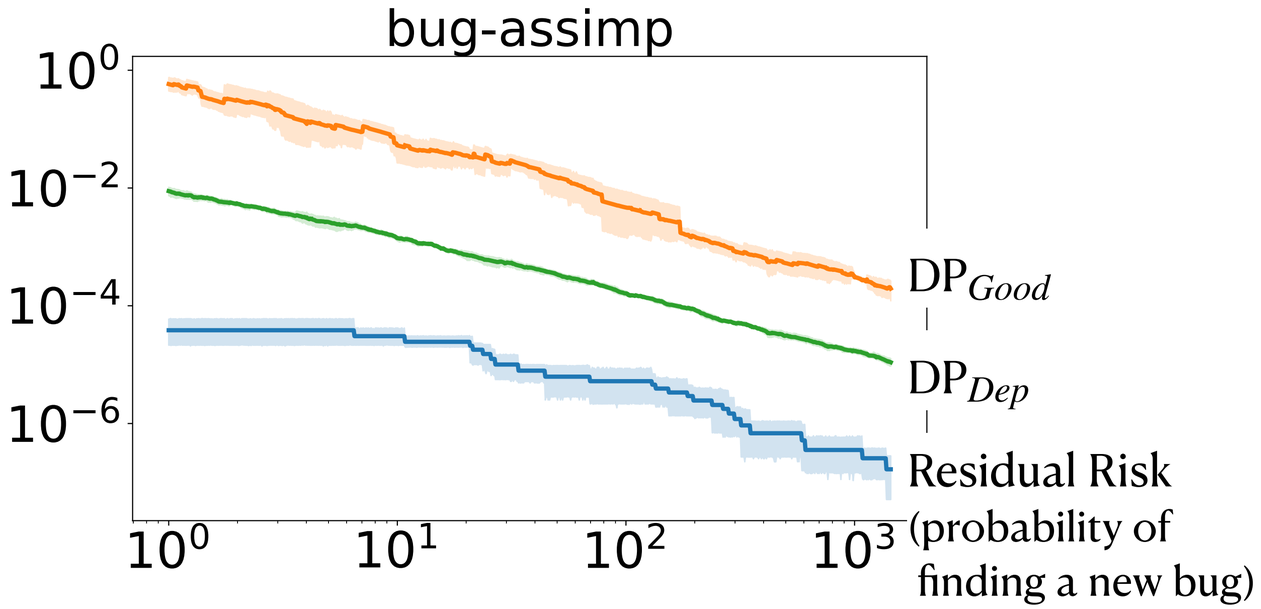
83% faster.

- Smaller number of basic blocks to monitor

  + less computation needed for estimation

  $\Rightarrow$ the execution time became shorter

Result 2. Accuracy of Dependency-aware Estimator



Relative Error

Subject	$\|C\|$	RE^r_{Good}	RE^r_{Dep}
Assimp	5.6	1090.6	53.4
File	4.4	124.3	38.5
Harfbuzz	4.2	3557.3	347.7
Libxml2	6.0	1190.2	107.6

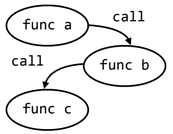
The dependency-aware estimator gives a **significantly tighter upper-bound of the residual risk** compared to the dependency-agnostic estimator.

Limitation of Existing Estimator

- They assume **independence** between program elements. ⇐ **NOT TRUE!**

```
...
if (pred)
  stmt;
...
```

'stmt' is executed only if 'if' is executed.



There are long call-chains in the program.

Dependencies led to considerable **overestimation** of the DP.

⇒ **False underconfidence** in the testing result

⇒ **Waste of resources** on an already saturated (unable to find anything new) testing campaign

Existing estimators

— Dependency-aware estimator [3] —

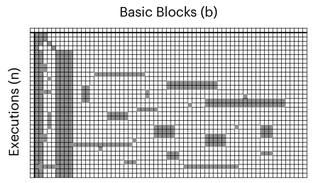
$$\hat{DP}_{Dep} = \frac{\#\text{ of executions with singleton}}{\#\text{ test execution}}$$

Problem:

Naïvely computing $\hat{DP}_{Dep}$ requires a **huge space** $O(n \times b)$, where

- n: # of test executions so far
- b: # of basic blocks in the program

Coverage Matrix (size: $n \times b$)



E.g., TCPDUMP & LIBPCAP, *24 hours of fuzzing*

Number of basic blocks (b)	Number of exec. after 24 hours (n)	b X n
14,355	467,938,080	~6 trillion

Dr. Seongmin Lee

UCLA

🏠 https://nimgnoeseel.github.io/

Dr. Marcel Böhme

MPI-SP

🏠 https://mpi-softsec.github.io/

🎯 What we want to know: **Which execution covers singletons?**

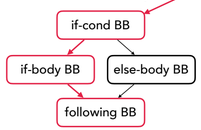
Insight 1: No need to monitor all basic blocks

Example.

```
if (p)
    stmt1;
else
    stmt2;
stmt3;
```

Code

Control Flow Graph



1. Execution $\mathbf{E}$ is the only execution covering if-cond BB.

* These basic blocks are singletons.

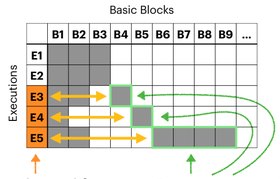
2. if-body BB already tells you $\mathbf{E}$ covers a singleton w/o help from if-cond BB.

⇒ No need to monitor if-body BB

If we use the **dominance relationship** in the **CFG**, we can **safely ignore** basic blocks for monitoring while still accurately determining which execution covers singletons.

🎯 What we want to know: **The number of executions covering singletons**

Insight 2: No need to record execution history



$O(n \times b)$ **Executions with singletons** = **Singleton Clusters** := set of singletons covered together in the same execution. $O(b)$

Our Solution: Two insights for efficient dependency-aware estimation

Insight 1:
No need to monitor all basic blocks

Optimization 1.
Dropping safely ignorable basic blocks

$b \Rightarrow \approx 0.5b$

Insight 2:
No need to record execution history

Optimization 2.
Maintaining clusters of singletons

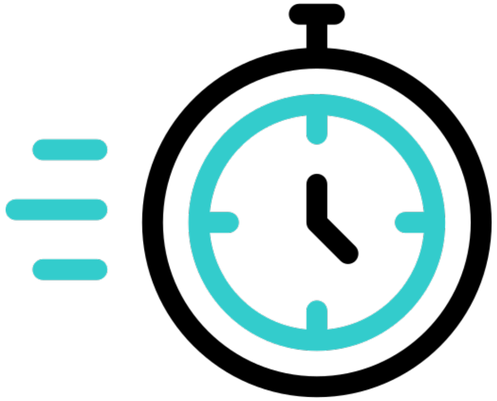
$O(b \cdot n) \Rightarrow O(b)$

Make a dependency-aware estimator feasible to apply at scale in software fuzz testing.

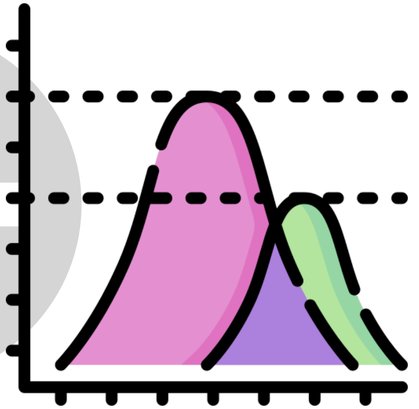
➡ **83% Faster Fuzzing**

➡ **>10× less bias in estimating bug-finding probability.**

Next Direction



$$\frac{\{P\}S\{Q\}\ ,\ \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$



Sampling-based low-overhead coverage approximation

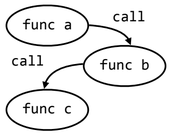
Unified verification with formal + statistical guarantee

Limitation of Existing Estimator

- They assume **independence** between program elements. ⇐ **NOT TRUE!**

```
...
if (pred)
  stmt;
...
```

'stmt' is executed only if 'if' is executed.



There are long call-chains in the program.

Dependencies led to considerable **overestimation** of the DP.

⇒ **False underconfidence** in the testing result

⇒ **Waste of resources** on an already saturated (unable to find anything new) testing campaign

Existing estimators

— Dependency-aware estimator [3] —

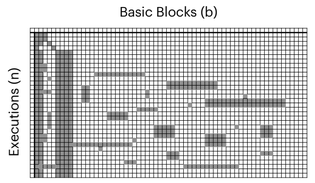
$$\hat{DP}_{Dep} = \frac{\#\text{ of executions with singleton}}{\#\text{ test execution}}$$

Problem:

Naïvely computing $\hat{DP}_{Dep}$ requires a **huge space** $O(n \times b)$, where

- n: # of test executions so far
- b: # of basic blocks in the program

Coverage Matrix (size: $n \times b$)



E.g., TCPDUMP & LIBPCAP, *24 hours of fuzzing*

Number of basic blocks (b)	Number of exec. after 24 hours (n)	b X n
14,355	467,938,080	~6 trillion

Dr. Seongmin Lee

UCLA

🏠 https://nimgnoeseel.github.io/

Dr. Marcel Böhme

MPI-SP

🏠 https://mpi-softsec.github.io/

🎯 What we want to know: **Which execution covers singletons?**

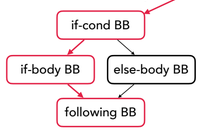
Insight 1: No need to monitor all basic blocks

Example.

```
if (p)
    stmt1;
else
    stmt2;
stmt3;
```

Code

Control Flow Graph



1. Execution $\mathbf{E}$ is the only execution covering if-cond BB.

* These basic blocks are singletons.

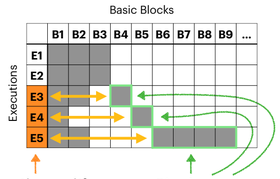
2. if-body BB already tells you $\mathbf{E}$ covers a singleton w/o help from if-cond BB.

⇒ No need to monitor if-body BB

If we use the **dominance relationship** in the **CFG**, we can **safely ignore** basic blocks for monitoring while still accurately determining which execution covers singletons.

🎯 What we want to know: **The number of executions covering singletons**

Insight 2: No need to record execution history



$O(n \times b)$ **Executions with singletons** = **Singleton Clusters** := set of singletons covered together in the same execution. $O(b)$

Our Solution: Two insights for efficient dependency-aware estimation

Insight 1:
No need to monitor all basic blocks

Optimization 1.
Dropping safely ignorable basic blocks

$b \Rightarrow \approx 0.5b$

Insight 2:
No need to record execution history

Optimization 2.
Maintaining clusters of singletons

$O(b \cdot n) \Rightarrow O(b)$

Make a dependency-aware estimator feasible to apply at scale in software fuzz testing.

➡ **83% Faster Fuzzing**

➡ **>10× less bias in estimating bug-finding probability.**