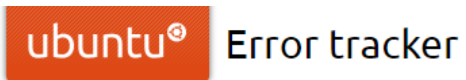
BENZENE: A Practical Root Cause Analysis System with an Under-Constrained State Mutation

Younggi Park, Hwiwon Lee, Jinho Jung,
Hyungjoon Koo, Huy Kang Kim

Software Crash?

Software Crash



Error tracker

Log in

We collect hundreds of thousands of error reports daily from millions of machines. This helps measure reliability of ...

Ubuntu 18.04 | Ubuntu 20.04 | Ubuntu 21.04 | Ubuntu 21.10 | Ubuntu 22.04 | Ubuntu 22.10 | Ubuntu 23.

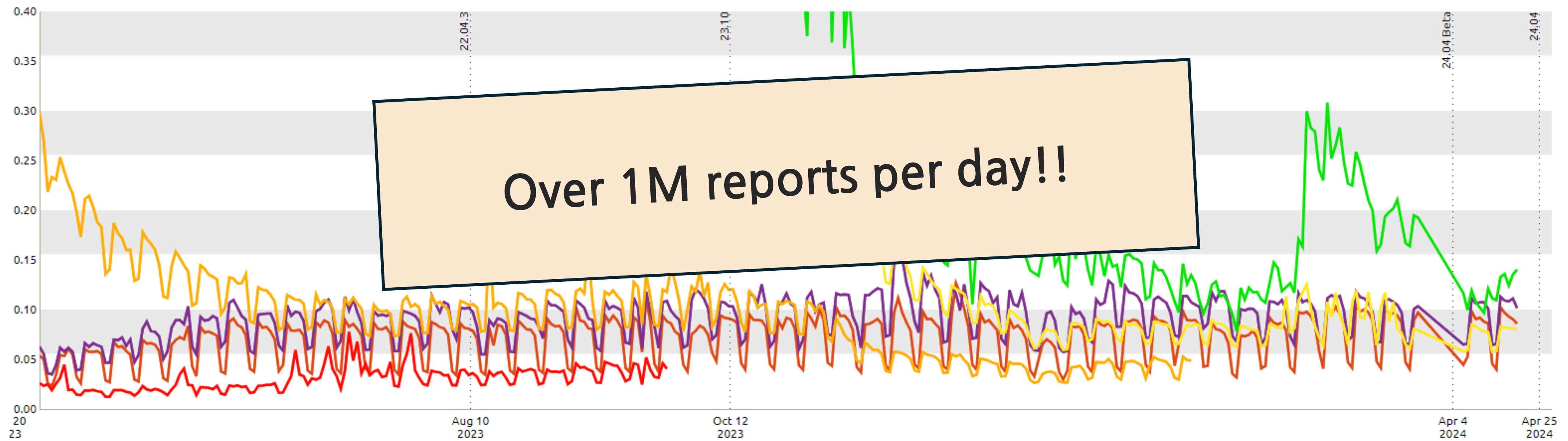
Ubuntu 18.04 (by 12.04 standards) | Ubuntu 20.04 (by 12.04 standards) | Ubuntu 21.04 (by 12.04 standards) | Ubuntu 21.10 (by 12.04 standards) | Ubuntu 22.04 (by 12.04 standards) | Ubuntu 22.

0.40 0.35 0.30 0.25 0.20 0.15 0.10 0.05 0.00

22.04.3 | 23.10 | 24.04 Beta | 24.04

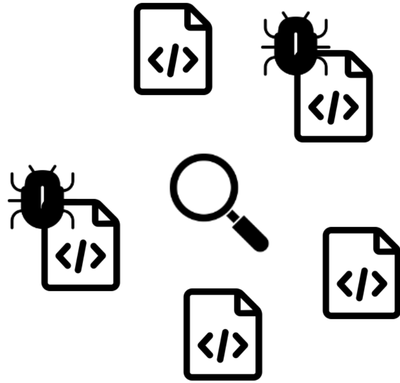
2023 | Aug 10 2023 | Oct 12 2023 | Apr 4 2024 | Apr 25 2024

Software Crash



Predicate-based Fault Localization

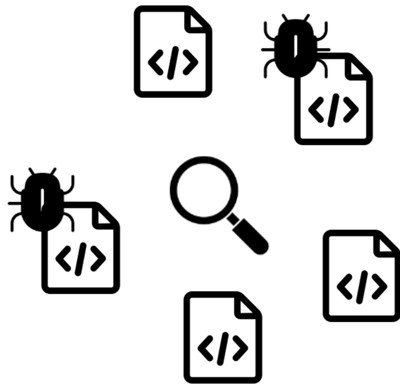
Predicate-based Fault Localization

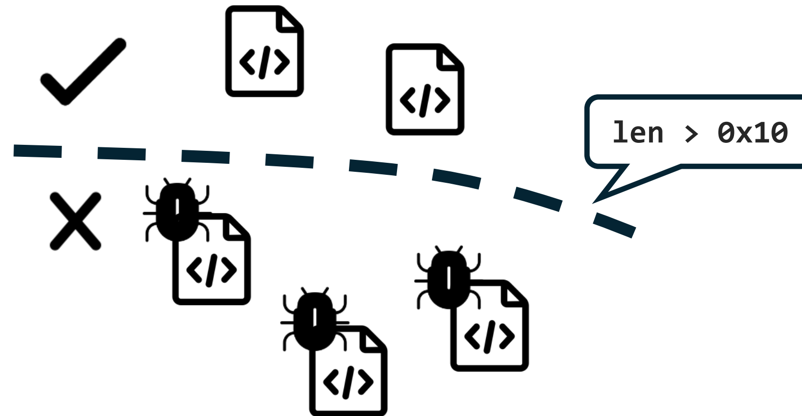


① Behavior Collection

Collect both crashing and non-crashing behaviors

Predicate-based Fault Localization





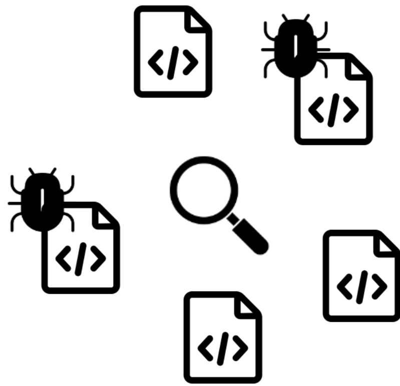
① Behavior Collection

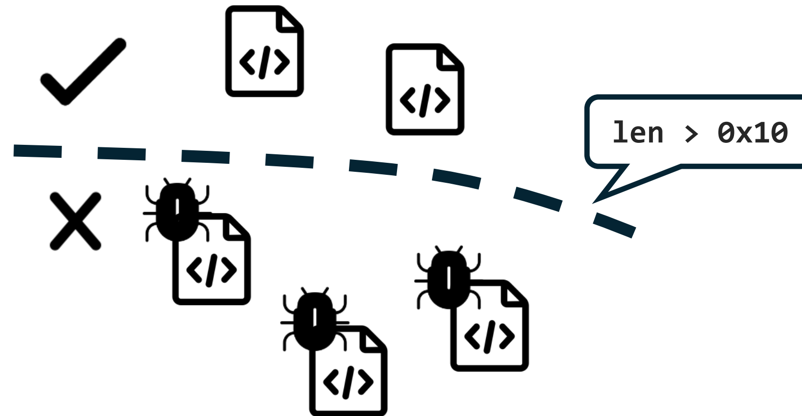
Collect both crashing and non-crashing behaviors

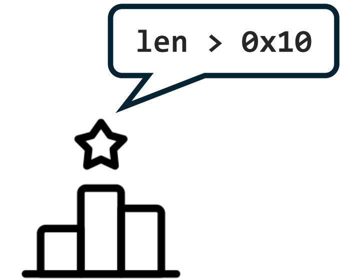
② Difference Observation

Extract predicates that statistically describe a crashing condition

Predicate-based Fault Localization







① Behavior Collection

Collect both crashing and non-crashing behaviors

② Difference Observation

Extract predicates that statistically describe a crashing condition

③ Predicate Ranking

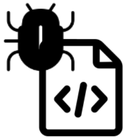
Rank the extracted predicates by their suspiciousness

BENZENE Overview

- We implement a root cause analysis system, BENZENE

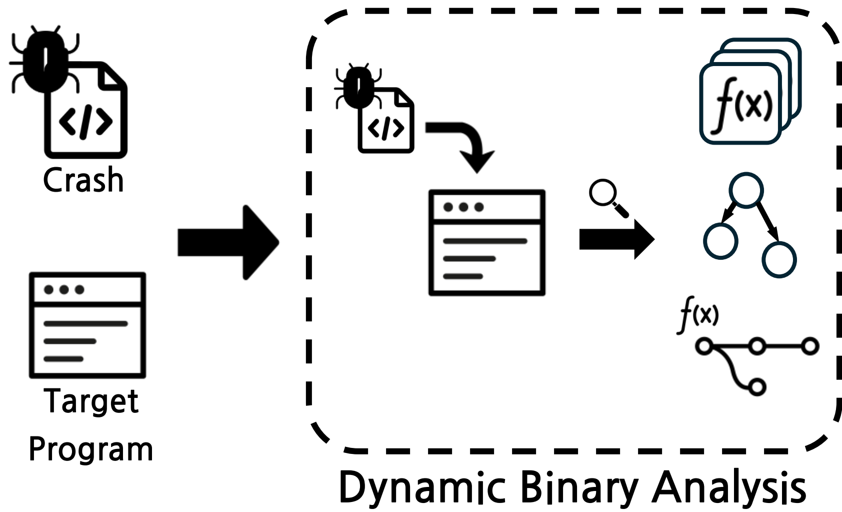
BENZENE Overview

- We implement a root cause analysis system, BENZENE



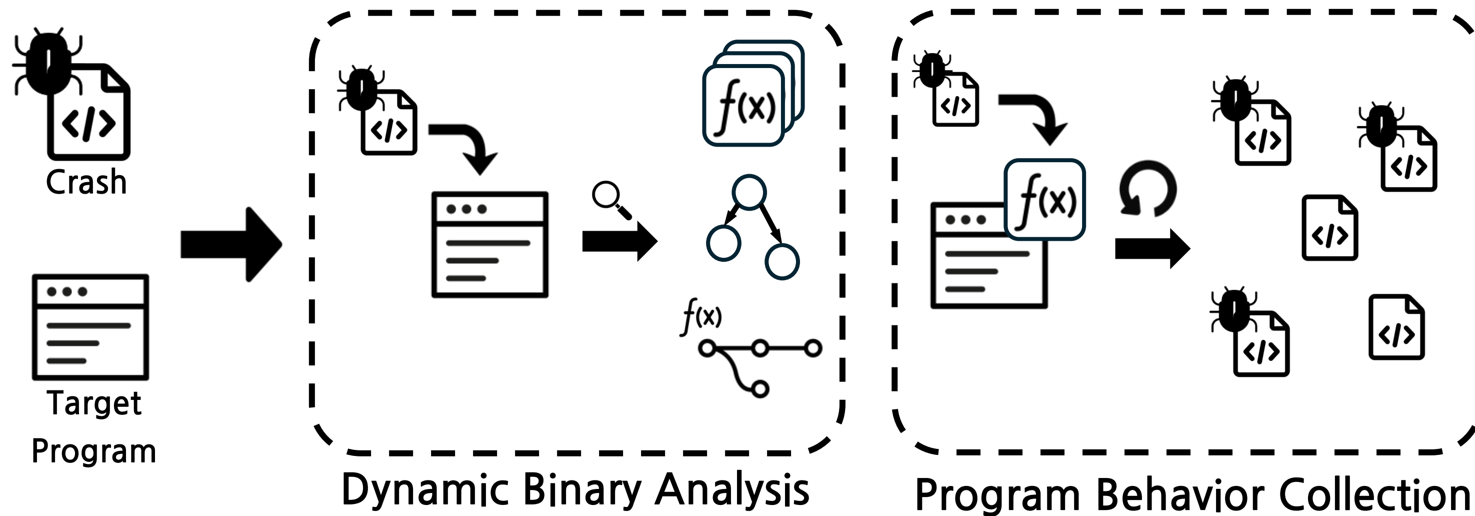
BENZENE Overview

- We implement a root cause analysis system, BENZENE



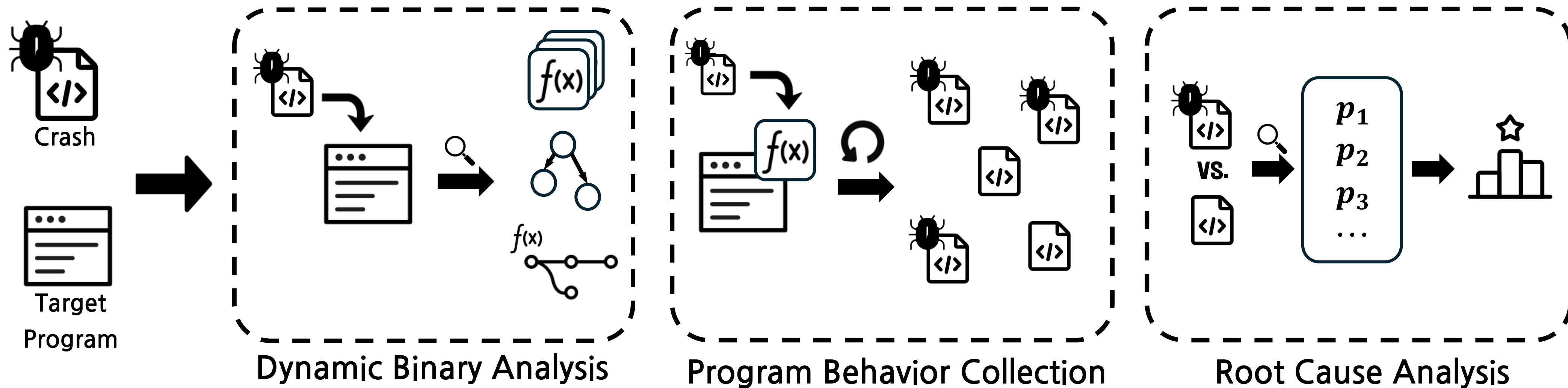
BENZENE Overview

- We implement a root cause analysis system, BENZENE



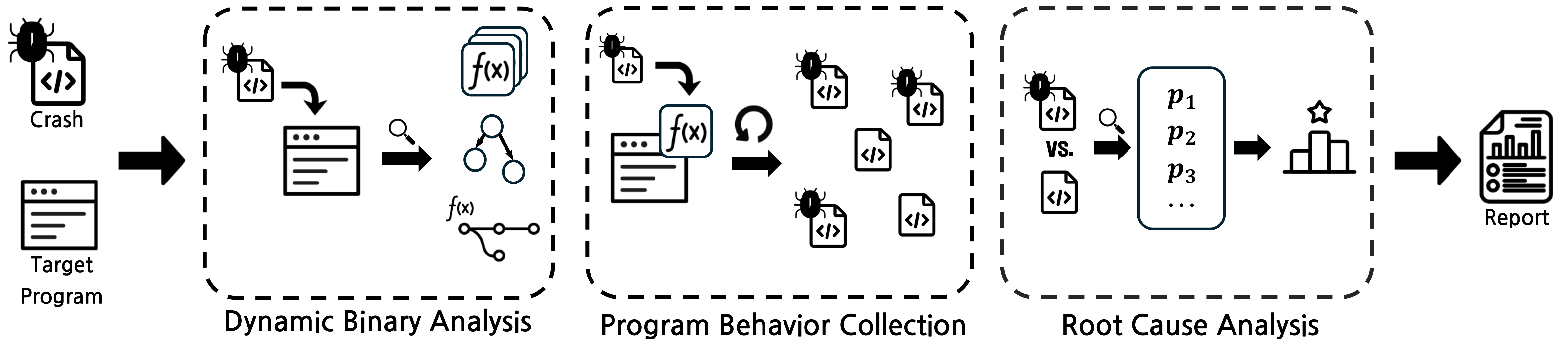
BENZENE Overview

- We implement a root cause analysis system, BENZENE



BENZENE Overview

- We implement a root cause analysis system, BENZENE



Motivating Example: PHP

- CVE-2019-6977: A heap buffer overflow due to the insufficient size allocation

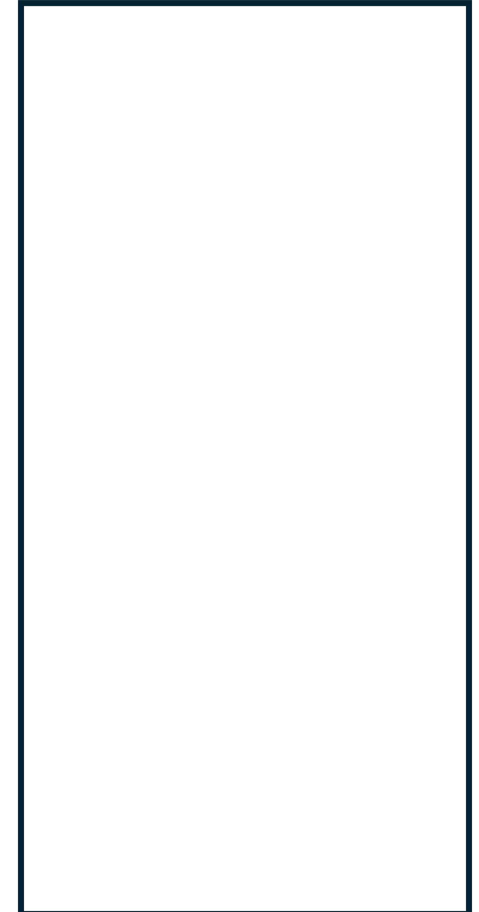
```
int gdImageColorMatch (...) {
          ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
          ...
}
```

※ For the explanation, the original source is simplified.

Motivating Example: PHP

```
int gdImageColorMatch(...)
{
            ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
            ...
}
```

colorsTotal is 0x1

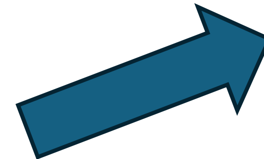


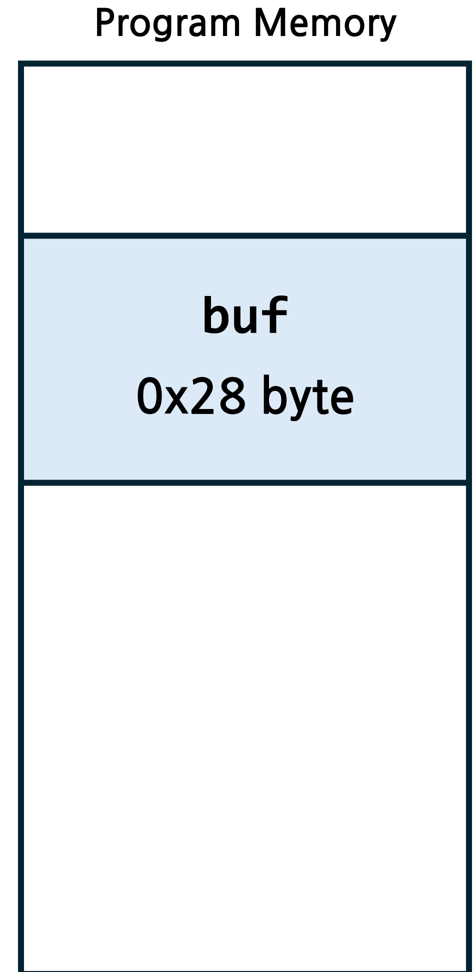
Motivating Example: PHP

```
int gdImageColorMatch(...)
{
        ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

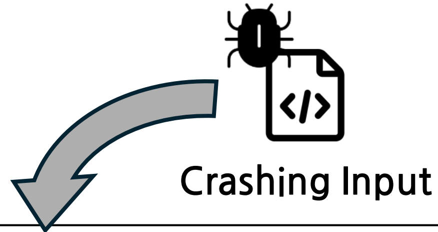
colorsTotal is 0x1

allocates
0x28 size memory



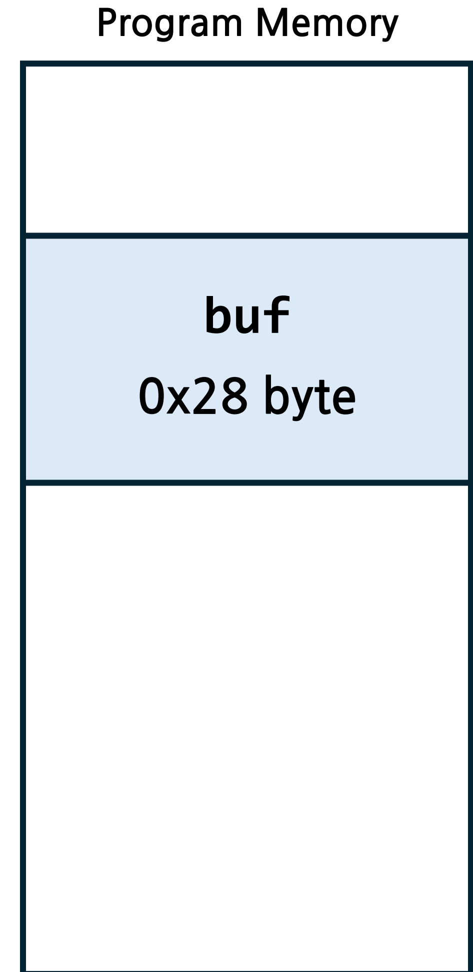


Motivating Example: PHP

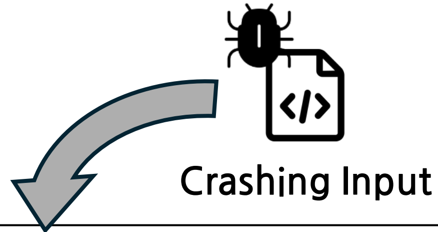



```
int gdImageColorMatch(...)
{
        ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

color is 0x80



Motivating Example: PHP

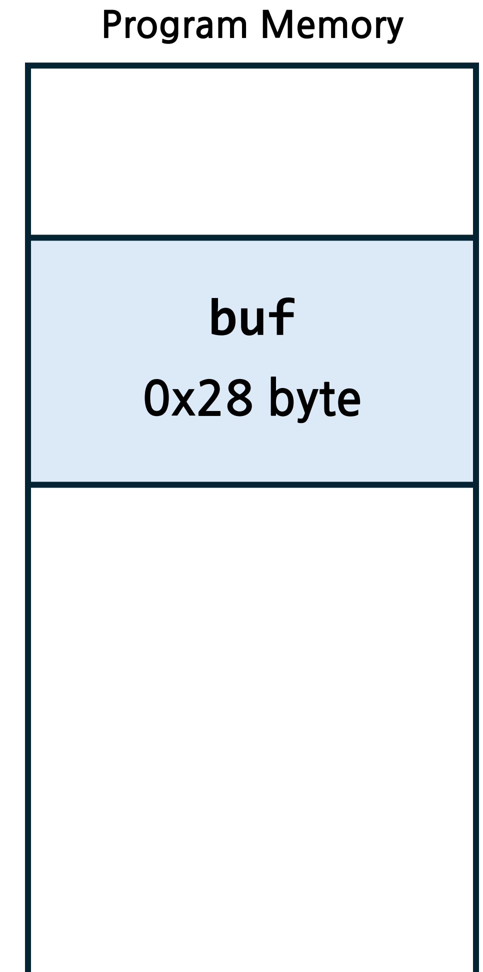



```
int gdImageColorMatch(...)
{
        ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

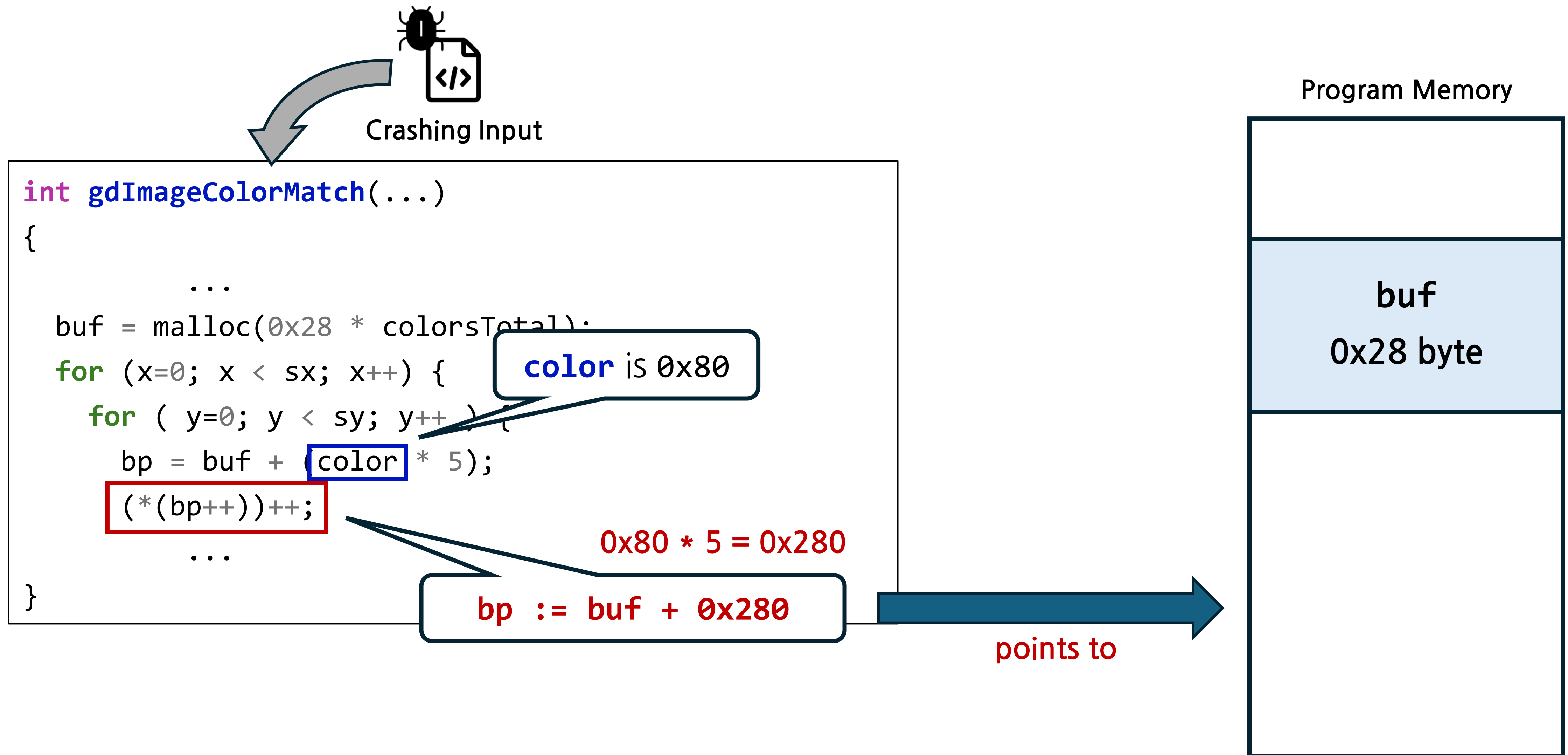
color is 0x80

0x80 * 5 = 0x280

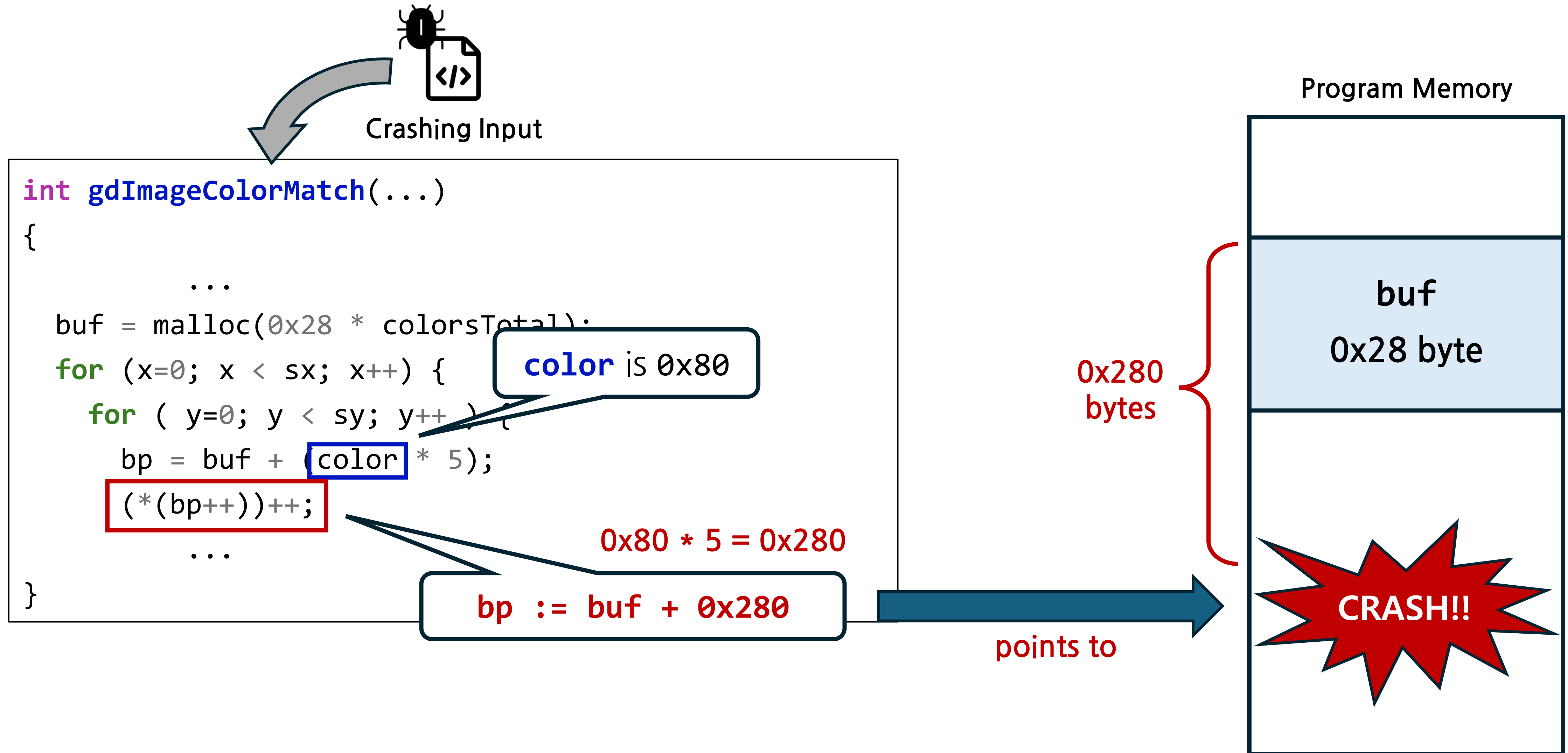
bp := buf + 0x280



Motivating Example: PHP



Motivating Example: PHP



Real-world Example: PHP

```
<?php
  $img1 = imagecreatetrucolor(0xfff, 0xfff);
  $img2 = imgcreate(0xfff, 0xfff);
  imagecolorallocate($img2, 0, 0, 0);
  imagesetpixel($img2, 0, 0, 0x80);
  imagecolormatch($img1, $img2);
?>
```

Crashing Input

```
int gdImageColorMatch(...)
{
          ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
          ...
}
```

developer patched here

buf is 0x28-size buffer

Out-of-Bound access, Crash Here!

Let’s locate

the root cause of this example

Root Cause Analysis Example

- colorsTotal : Observed values for colorsTotal

```
int gdImageColorMatch(...) {
          ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
          ...
}
```

Collected Behaviors

Behavior Sample	colorsTotal	Crash?

Root Cause Analysis Example

- **Crash?** (**Yes**/**No**) denotes whether the program has crashed for a given behavior

```
int gdImageColorMatch(...) {
          ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
          ...
}
```

Collected Behaviors

Behavior Sample	colorsTotal	Crash?

Root Cause Analysis Example

- First, we have the behavior of **the given input** that (obviously) crashes

```
int gdImageColorMatch(...) {
        ...                    0x1
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

Collected Behaviors

Behavior Sample	colorsTotal	Crash?
Crash	0x1	Yes

Root Cause Analysis Example

- Suppose behavior #1 is collected (`colorsTotal` is 0x400)

```
int gdImageColorMatch(...) {
          ...                 0x400
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
          ...
}
```

Collected Behaviors

Behavior Sample	colorsTotal	Crash?
Crash	0x1	Yes
#1	0x400	No

Root Cause Analysis Example

- Suppose behavior #1 is collected (`colorsTotal` is 0x400)

```
int gdImageColorMatch(...) {
        ...              0x400
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

Behavior #1 allocates sufficient memory

ed Behaviors

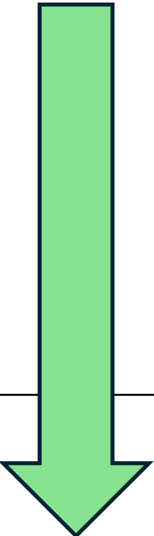
	orsTotal	Crash?
Crash	0x1	Yes
#1	0x400	No

Root Cause Analysis Example

- Suppose behavior #1 is collected (`colorsTotal` is 0x400)

```
int gdImageColorMatch(...) {
        ...                0x400
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

Behavior #1 allocates sufficient memory



Program Exit!!
(without crash)

...ed Behaviors

	...orsTotal	Crash?
Crash	0x1	Yes
#1	0x400	No

Root Cause Analysis Example

- Similarly, suppose we additionally collected 4 program behaviors…

```
int gdImageColorMatch(...) {
        ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

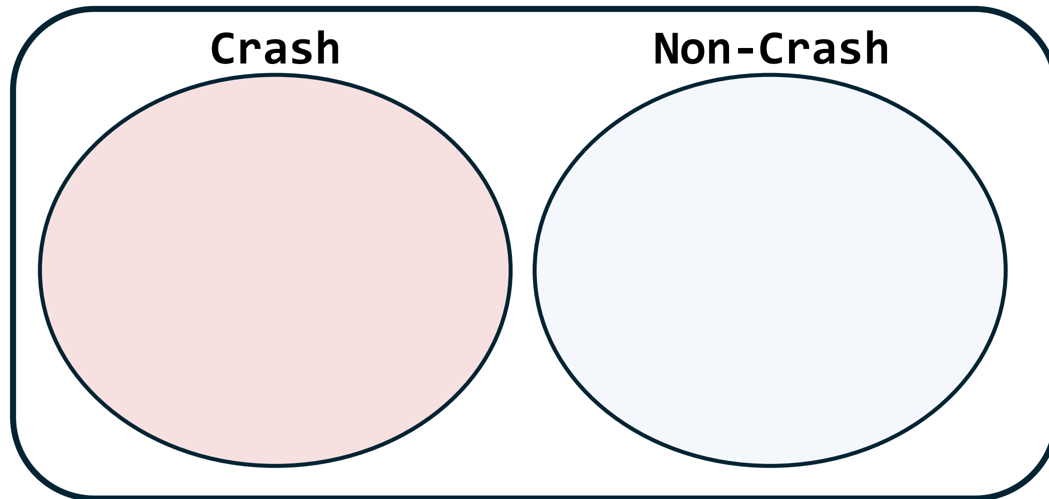
Four behaviors are collected!

Collected Behaviors

Behavior Sample	colorsTotal	Crash?
Crash	0x1	Yes
#1	0x400	No
#2	0x60	Yes
#3	0x80	No
#4	0x79	Yes
#5	0x20	Yes

Extracting Crashing Condition

- Observe a difference between crashing and non-crashing behaviors

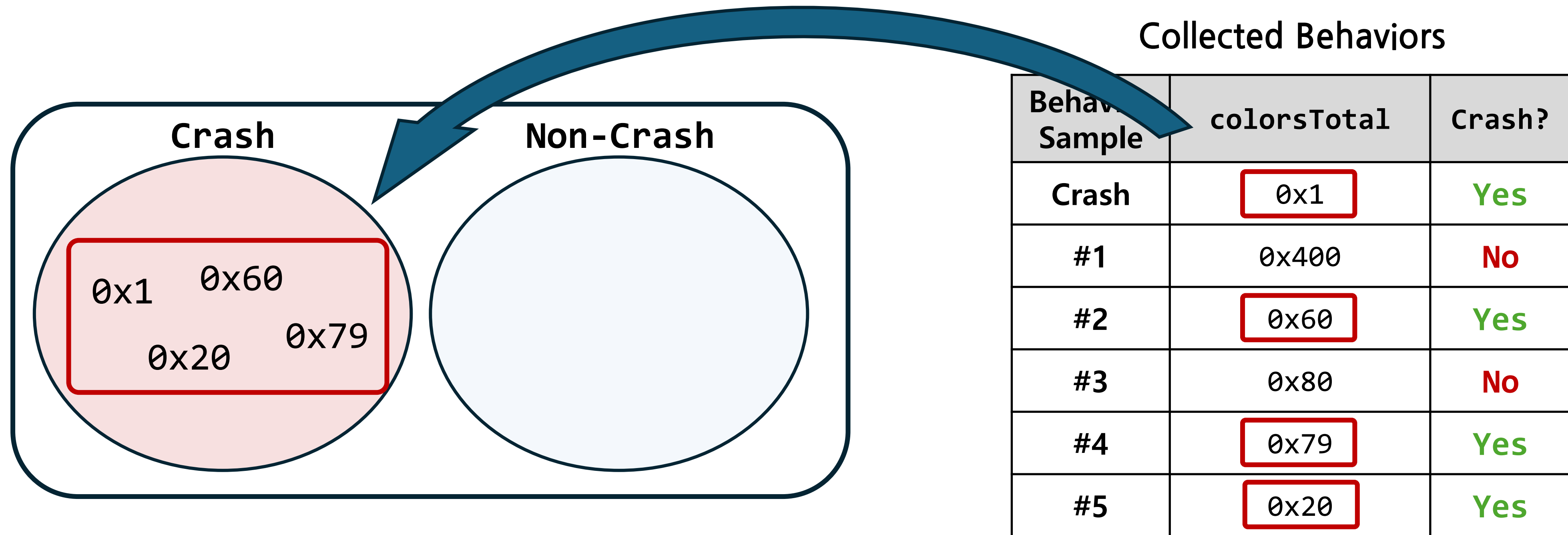


Collected Behaviors

Behavior Sample	colorsTotal	Crash?
Crash	0x1	Yes
#1	0x400	No
#2	0x60	Yes
#3	0x80	No
#4	0x79	Yes
#5	0x20	Yes

Extracting Crashing Condition

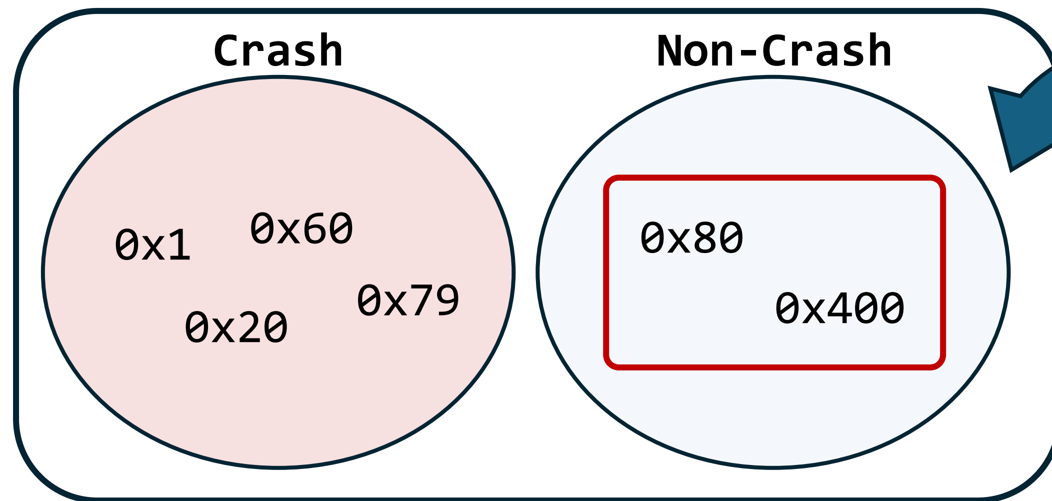
- Observe a difference between crashing and non-crashing behaviors



Beha[illegible] Sample	colorsTotal	Crash?
Crash	0x1	Yes
#1	0x400	No
#2	0x60	Yes
#3	0x80	No
#4	0x79	Yes
#5	0x20	Yes

Extracting Crashing Condition

- Observe a difference between crashing and non-crashing behaviors

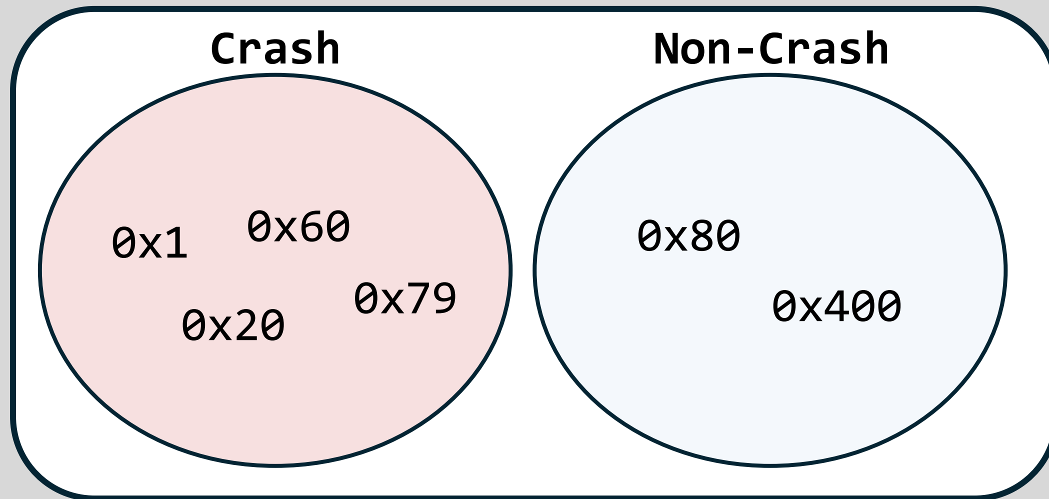


Collected Behaviors

Behavior Sample	colorsTotal	Crash?
Crash	0x1	Yes
#1	0x400	No
#2	0x60	Yes
#3	0x80	No
#4	0x79	Yes
#5	0x20	Yes

Extracting Crashing Condition

- Observe a difference between crashing and non-crashing behaviors

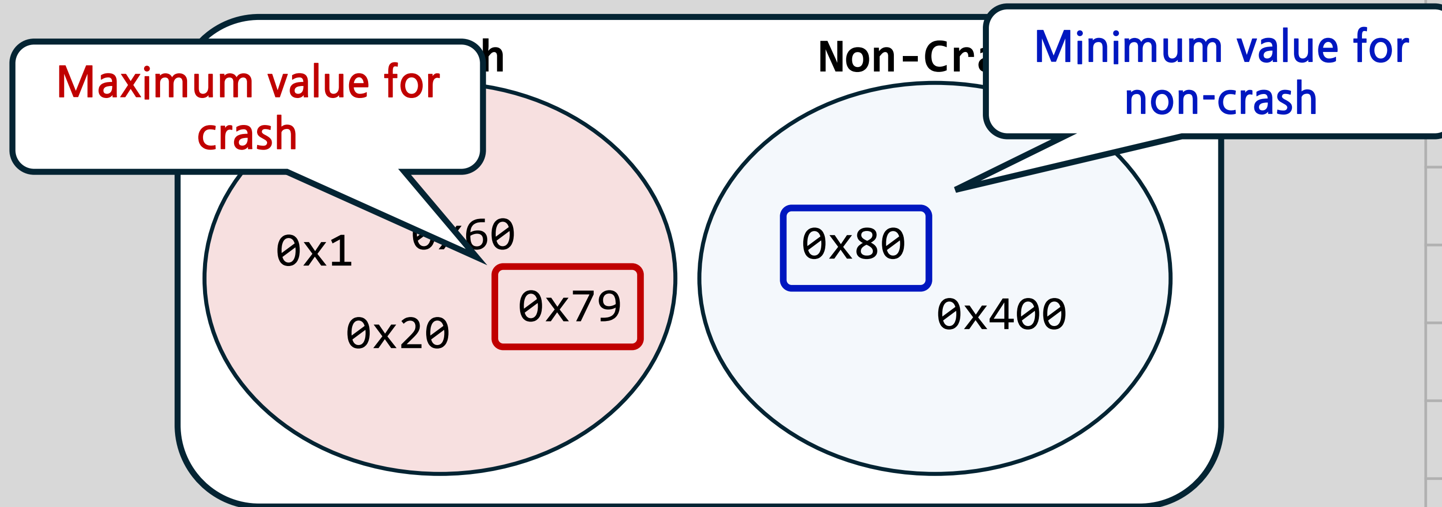


Collected Behaviors

Behavior Sample	colorsTotal	Crash?
Crash	0x1	Yes
#1	0x400	No
#2	0x60	Yes
#3	0x80	No
#4	0x79	Yes
#5	0x20	Yes

Extracting Crashing Condition

- Observe a difference between crashing and non-crashing behaviors



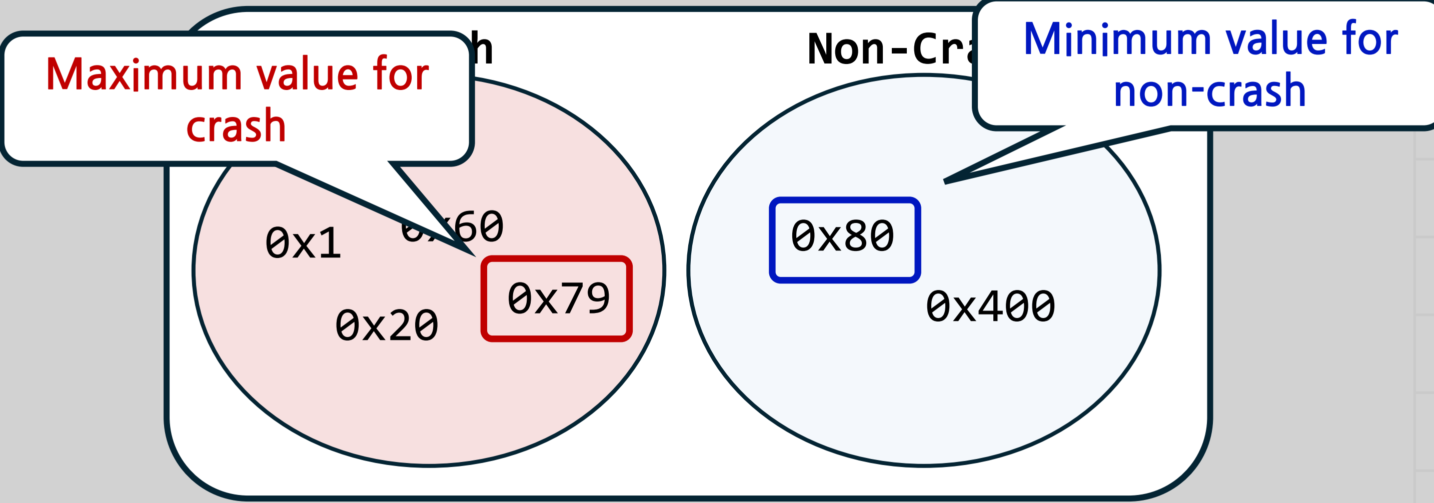
Collected Behaviors

Behavior Sample	colorsTotal	Crash?
Crash	0x1	Yes
#1	0x400	No
#2	0x60	Yes
#3	0x80	No
#4	0x79	Yes
#5	0x20	Yes

Extracting Crashing Condition

- Observe a difference between crashing and non-crashing behaviors

So, the crash is triggered when:
`[colorsTotal < 0x80]`

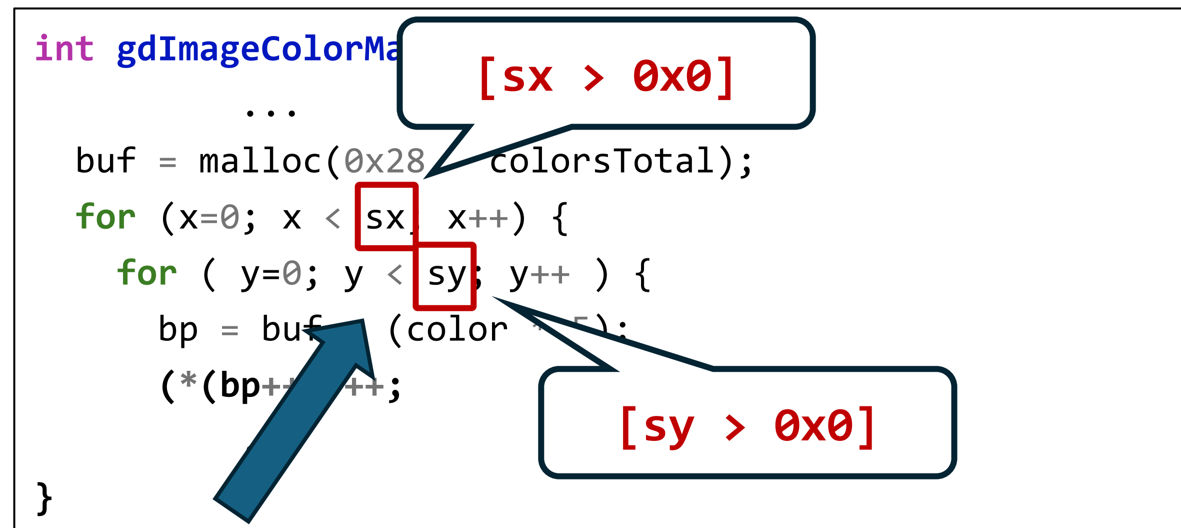


Collected Behavio[illegible]

Behavior Sample	colorsTota[illegible]	Crash?
Crash	[illegible]	Yes
#1	[illegible]	No
#2	[illegible]	Yes
#3	0x80	No
#4	0x79	Yes
#5	0x20	Yes

Extracting Crashing Condition

- Extract the predicates for the rest of the variables using the same method



extract predicates using the same method

Extracting Crashing Condition

Crashing condition that consists of three predicates

Crashing Condition

p_1 := `colorsTotal < 0x80`
p_2 := `sx > 0x0`
p_3 := `sy > 0x0`

Extracting Crashing Condition

Crashing condition that consists of three predicates

Crashing Condition

p_1 := colorsTotal < 0x80
p_2 := sx > 0x0
p_3 := sy > 0x0

pinpoints the root cause!

Essentials of root cause analysis?

“p_1 `:= colorsTotal < 0x80`”

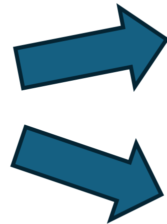
What if the behaviors were not enough

- Suppose non-crashing behavior #1 and #3 were not collected…

Collected Behaviors

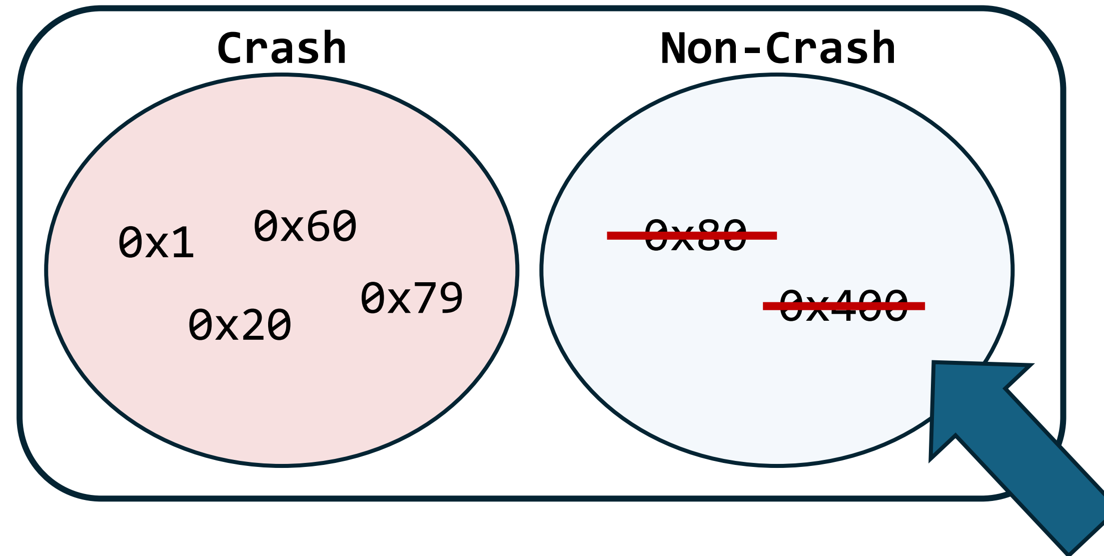
Behavior Sample	colorsTotal	Crash?
Crash	0x1	Yes
~~#1~~	~~0x400~~	~~No~~
#2	0x60	Yes
~~#3~~	~~0x80~~	~~No~~
#4	0x79	Yes
#5	0x20	Yes

Two behaviors not collected



What if the behaviors was not enough

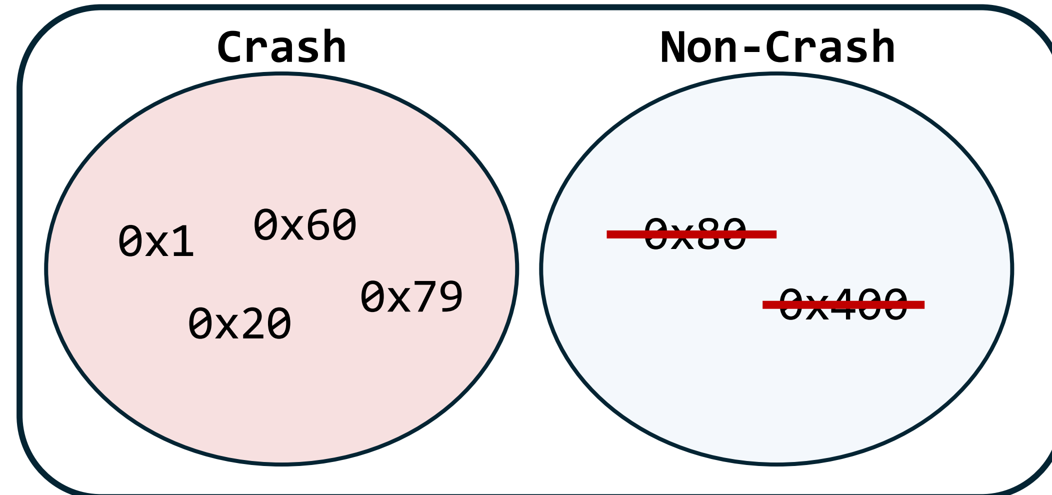
- Then, no behavioral difference exists for `colorsTotal`…



They are NOT included anymore

What if the behaviors was not enough

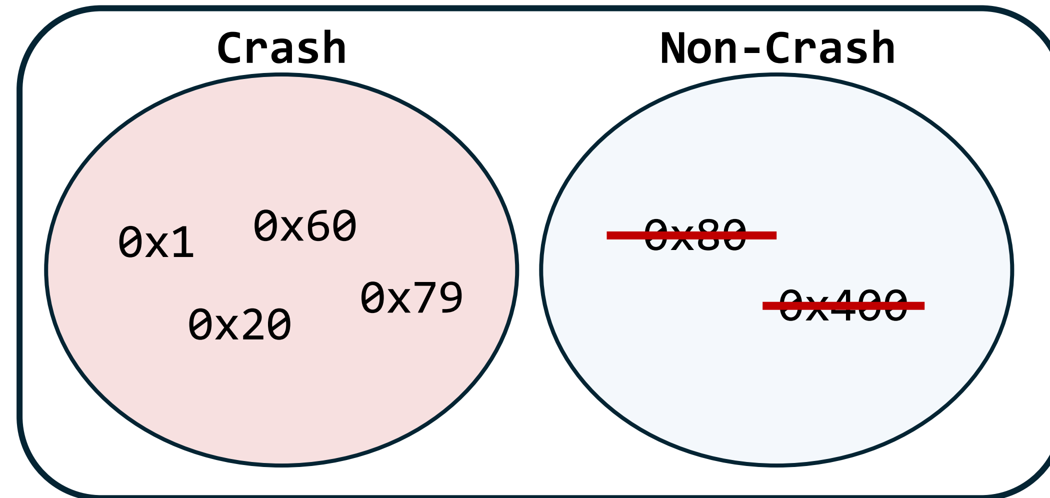
- Then, no behavioral difference exists for `colorsTotal`...

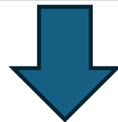


… It seems `colorsTotal` is irrelevant to the crash

What if the behaviors was not enough

- Our synthesized crashing condition <u>will not contain the root cause</u>



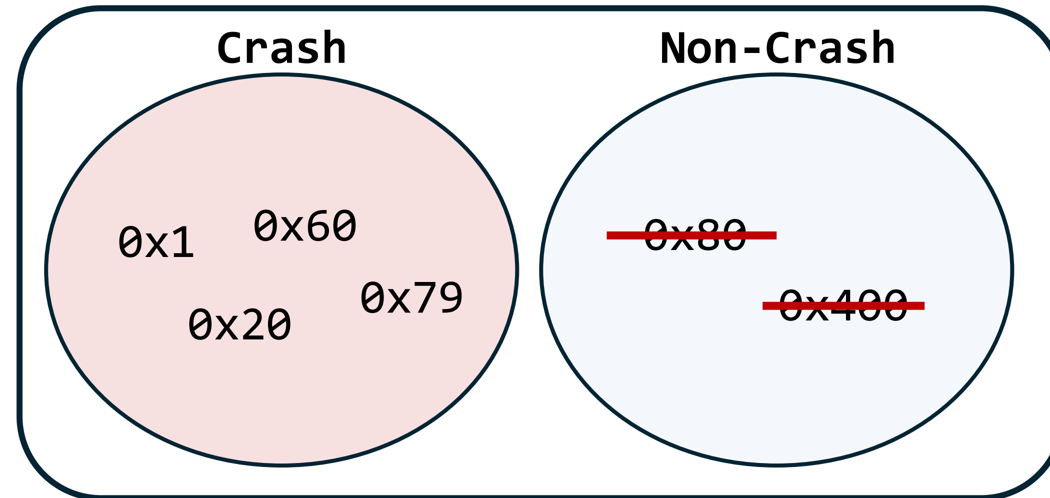


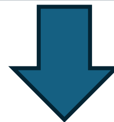
Crashing Condition

~~p_1 := colorsTotal < 0x80~~
p_2 := sx > 0x0
p_3 := sy > 0x0

What if the behaviors was not enough

- Our synthesized crashing condition <u>will not contain the root cause</u>





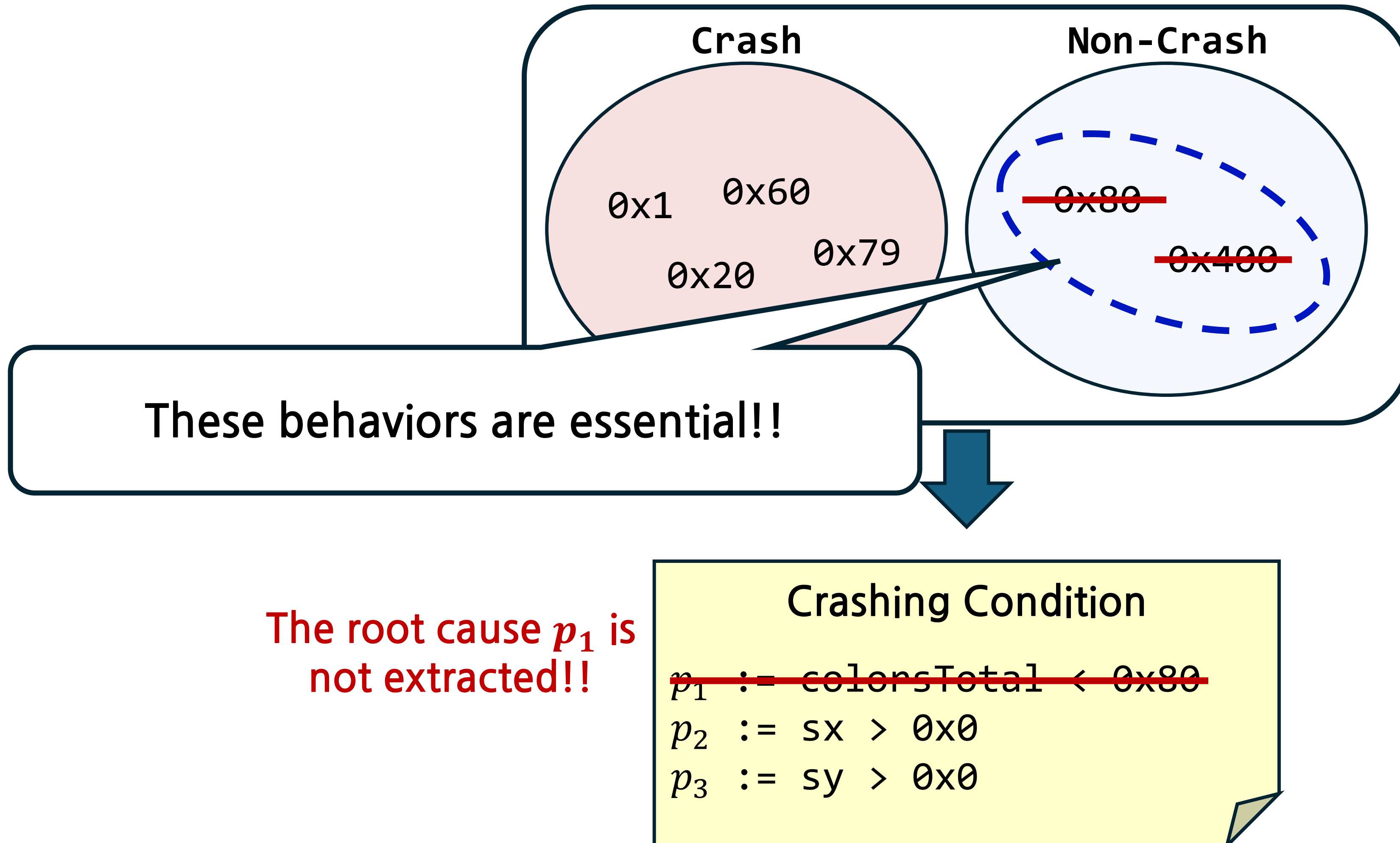
The root cause p_1 is not extracted!!

Crashing Condition

~~p_1 := colorsTotal < 0x80~~
p_2 := sx > 0x0
p_3 := sy > 0x0

What if the behaviors was not enough

- Our synthesized crashing condition will not contain the root cause



Behavior Dataset Requirement

- Behaviors with the following two conditions at the same time

Behavior Dataset Requirement

- Behaviors with the following <u>two conditions</u> at the same time

  ① Non-crashing behaviors

  - We should observe the behavioral differences comparing to crashing ones

Behavior Dataset Requirement

- Behaviors with the following two conditions at the same time

① Non-crashing behaviors

- We should observe the behavioral differences comparing to crashing ones

② Similar behaviors with the original crash

- If not, the observed differences would not be relevant to the root cause

Behavior Dataset Requirement

- Behaviors with the following two conditions at the same time

① Non-crashing behaviors

- We should observe the behavioral differences comparing to crashing ones

② Similar behaviors with the original crash

- If not, the observed differences would not be relevant to the root cause

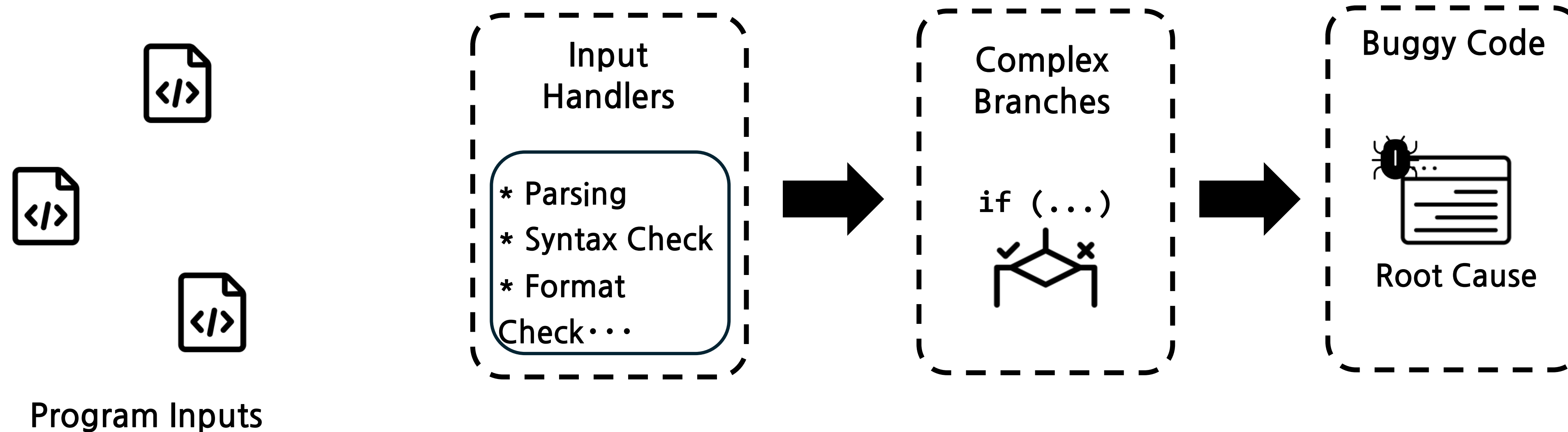
Crash-similar *AND* non-crashing behavior!

How do we get such key behaviors?

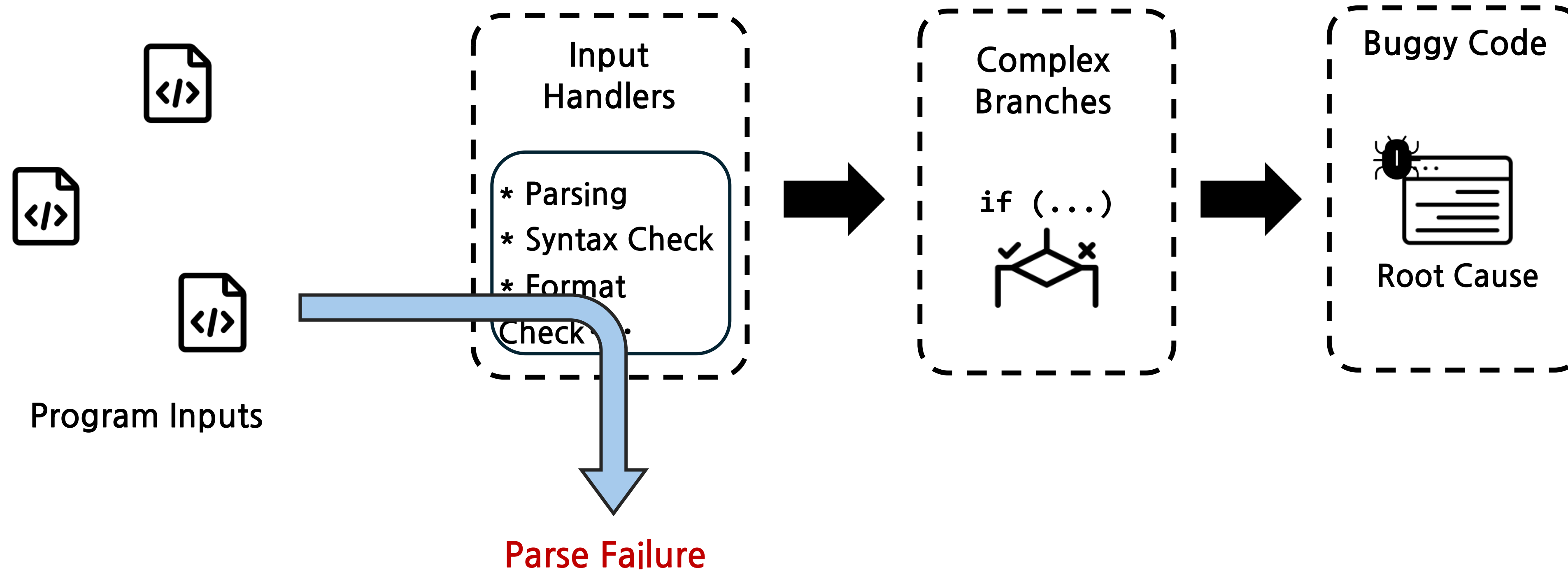
Fuzzing with a given crash

(Crash Exploration)

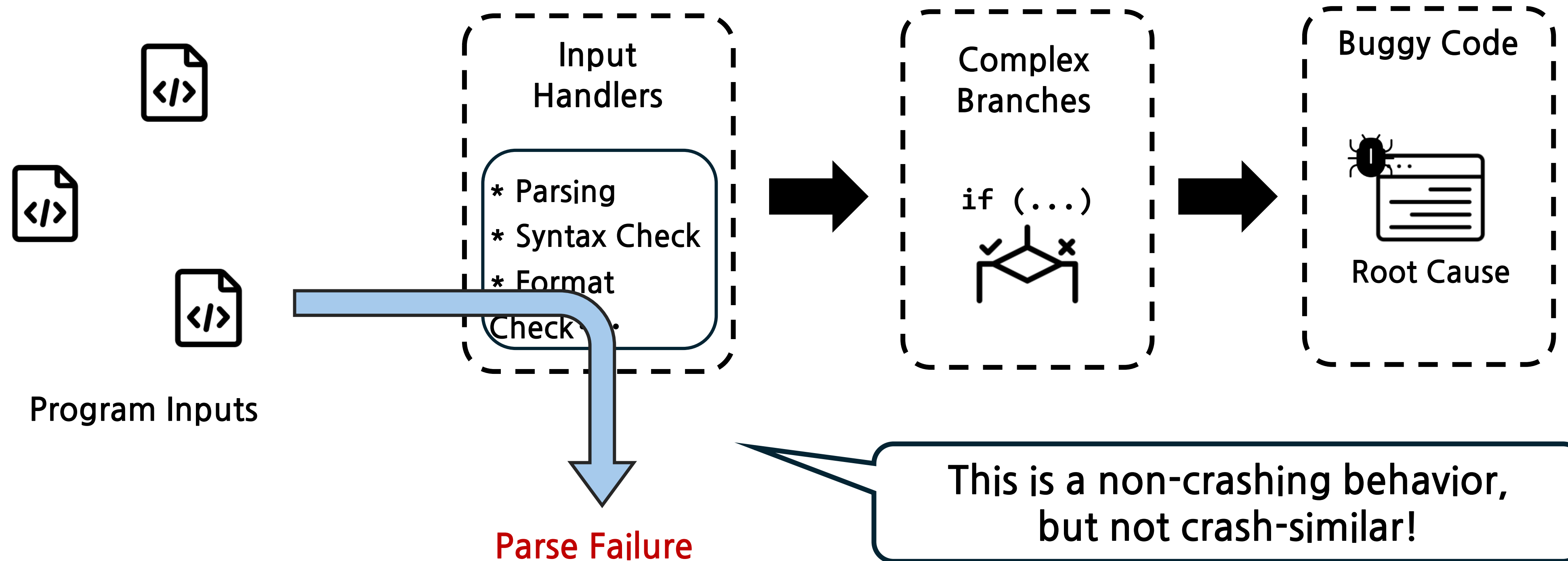
Finding *Proper* Behaviors is Not Easy



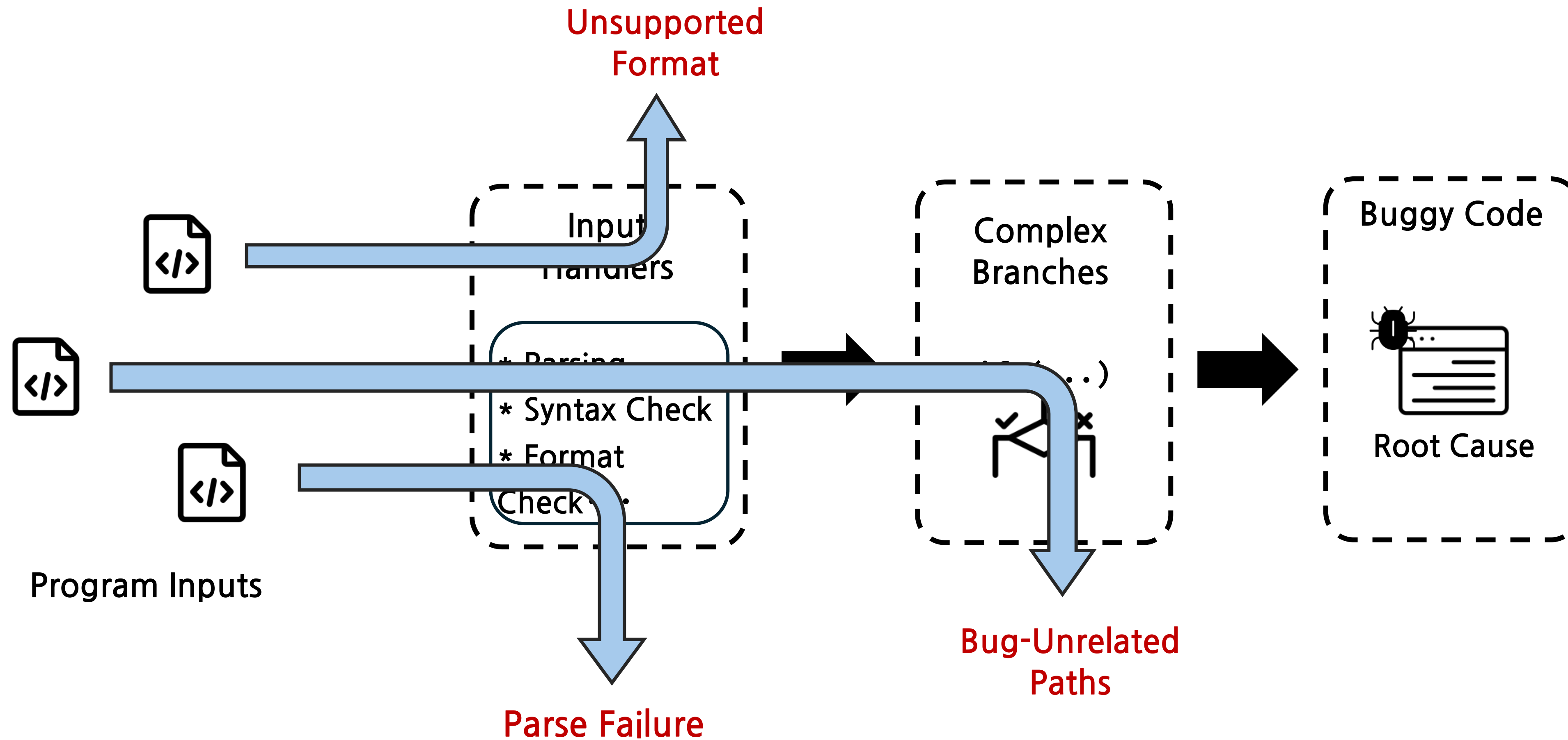
Finding *Proper* Behaviors is Not Easy



Finding *Proper* Behaviors is Not Easy

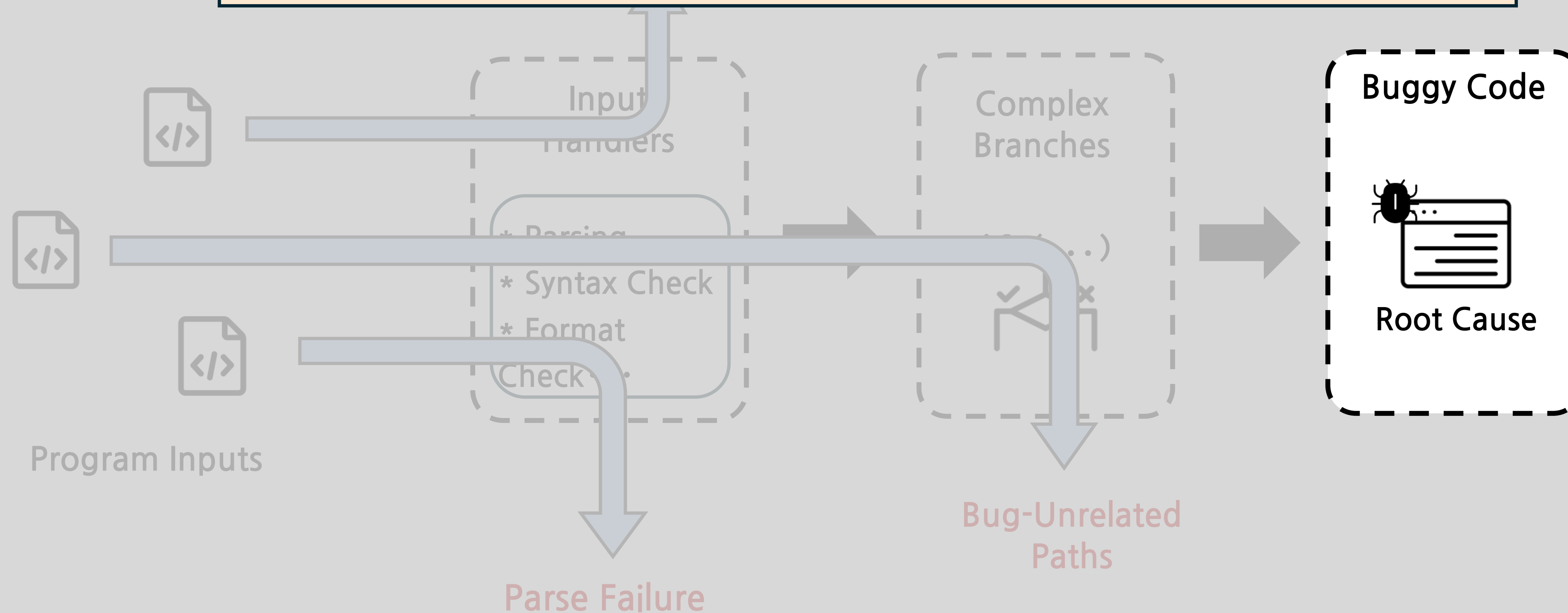


Finding *Proper* Behaviors is Not Easy



Finding *Proper* Behaviors is Not Easy

But, they exhibit *NO* behavioral differences associated with the root cause

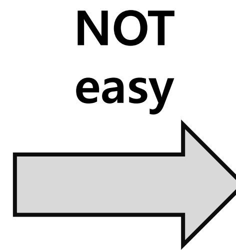


Root Cause Revealing Behavior

- CVE-2019-6977 requires…

```
<?php
  $img1 = imagecreatetrucolor(0xfff, 0xfff);
  $img2 = imgcreate(0xfff, 0xfff);
  imagecolorallocate($img2, 0, 0, 0);

  imagesetpixel($img2, 0, 0, 0x80);
  imagecolormatch($img1, $img2);
?>
```




```
<?php
  $img1 = imagecreatetrucolor(0xfff, 0xfff);
  $img2 = imgcreate(0xfff, 0xfff);
  for ($i = 0; $i < 255; $i+=1) {
    imagecolorallocate($img2, 0, 0, 0);
  }
  imagesetpixel($img2, 0, 0, 0x80);
  imagecolormatch($img1, $img2);
?>
```

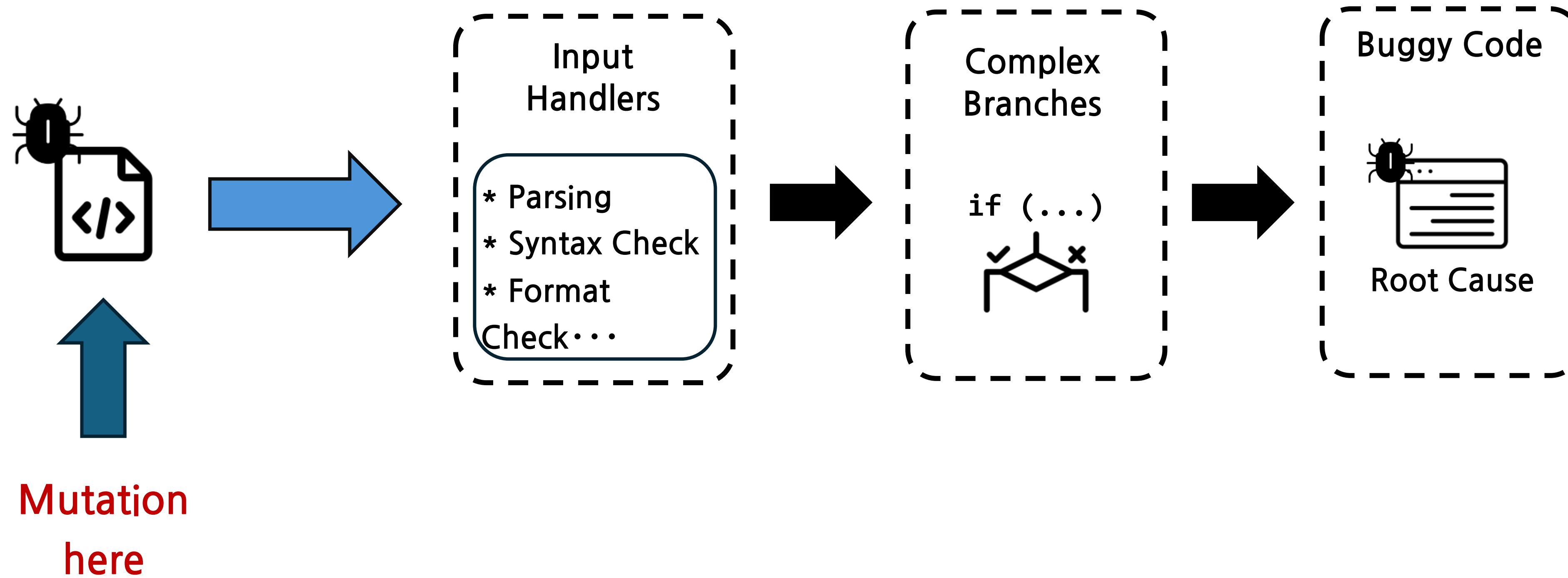
Our Idea:

Under-Constrained State Mutation

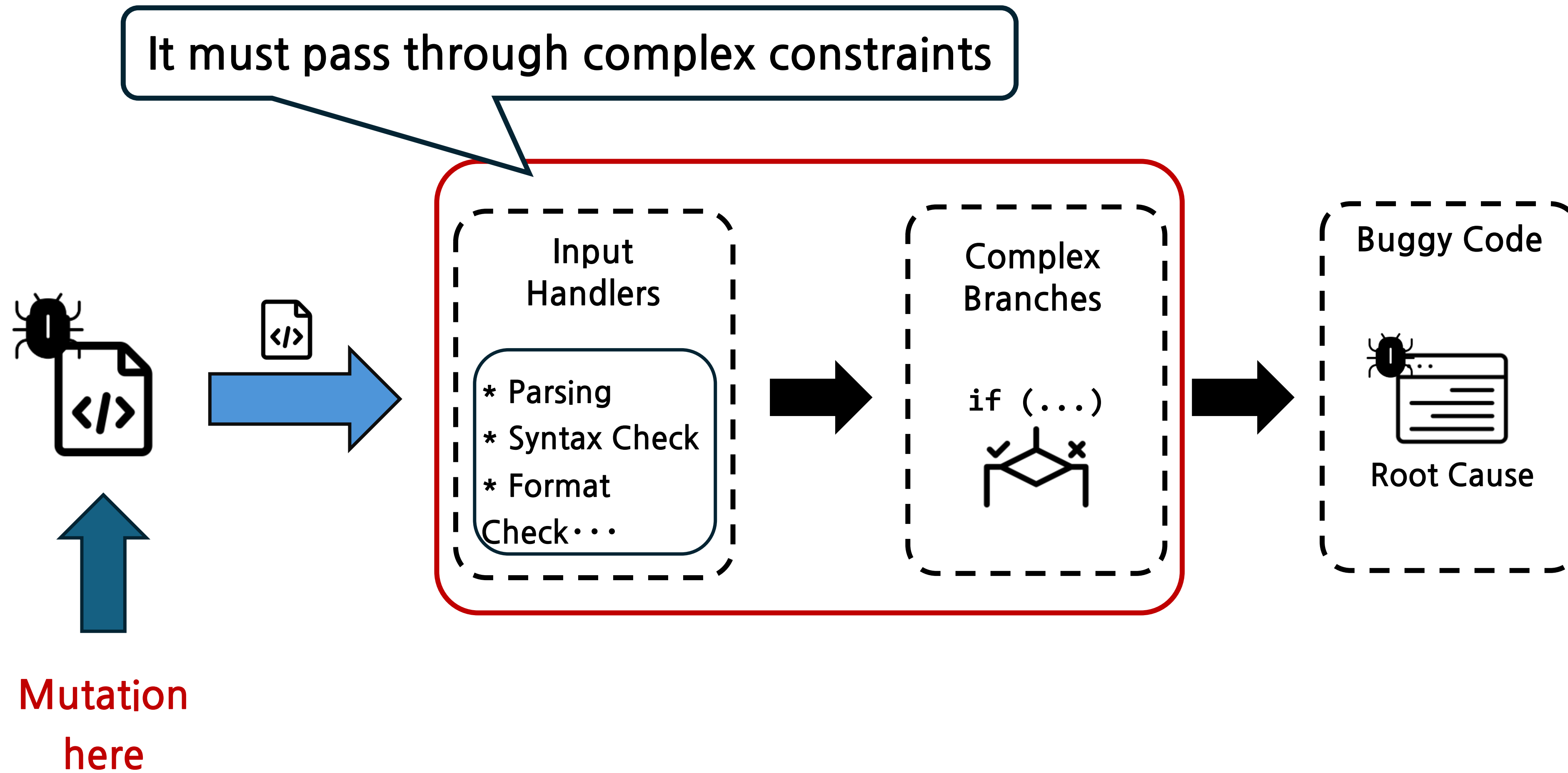
We forcefully mutate **a program state**

in the middle of execution!

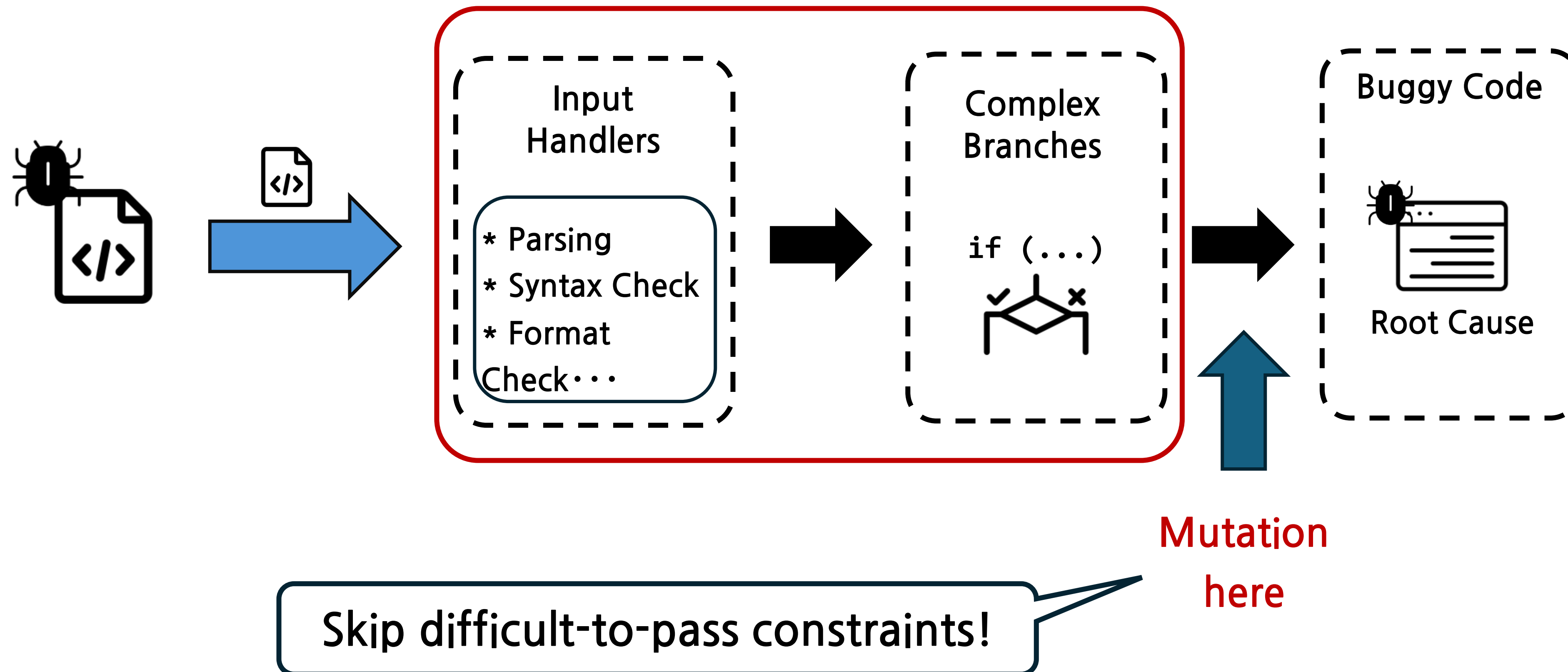
Intuition of State Mutation



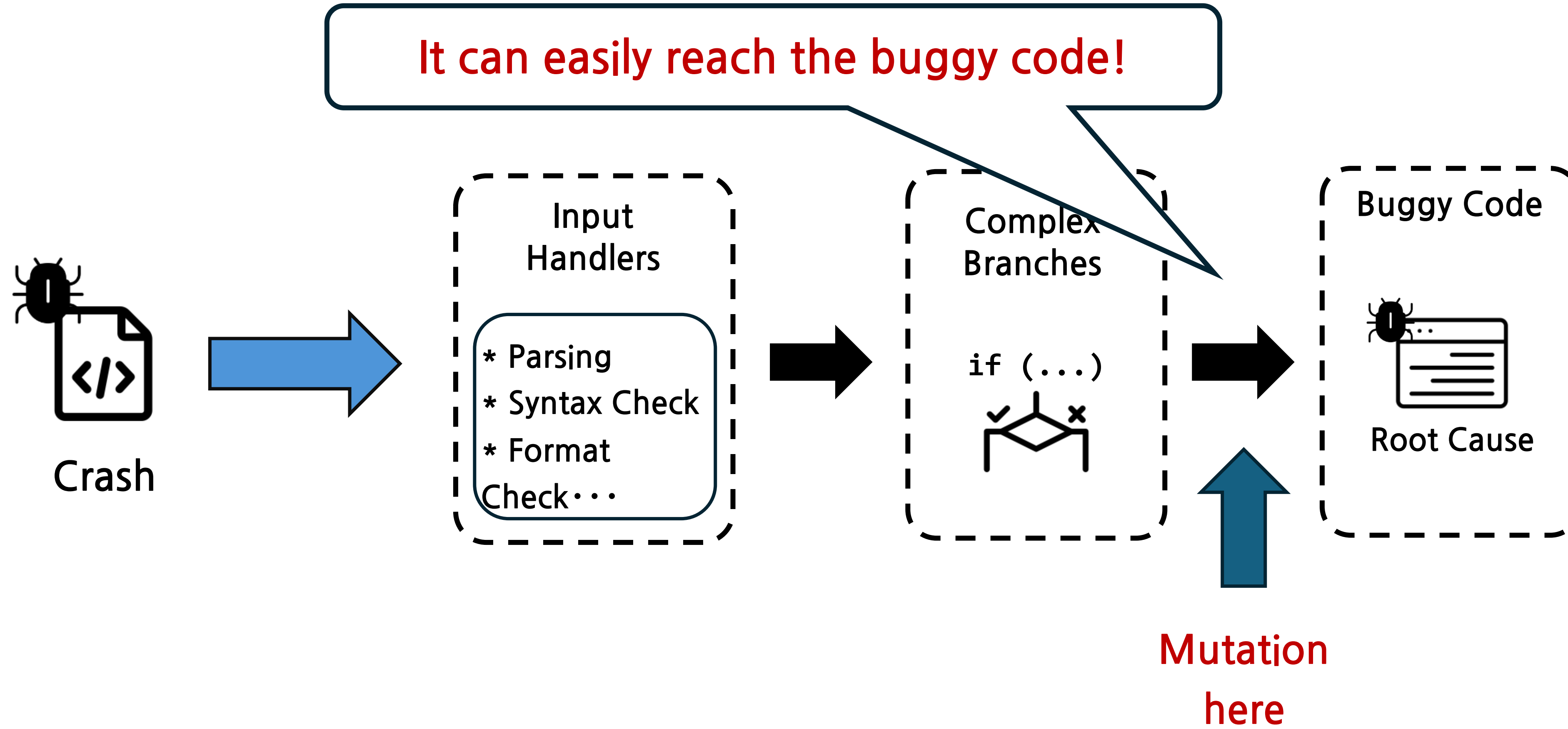
Intuition of State Mutation



Intuition of State Mutation



Intuition of State Mutation



Under-Constrained State Mutation

Source Code

```
int gdImageColorMatch(...)
{
        ...
  buf = emalloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

Under-Constrained State Mutation

Source Code

Machine Code

```c
int gdImageColorMatch(...)
{
        ...
  buf = emalloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

```
    ...
jle 0x555555800bc9
lea esi, [rax+rax*4]
xor edx, edx
    ...
```

Under-Constrained State Mutation

Source Code

```
int gdImageColorMatch(...)
{
        ...
  buf = emalloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

corresponds to

Machine Code

```
    ...
jle 0x555555800bc9
lea esi, [rax+rax*4]
xor edx, edx
    ...
```

Under-Constrained State Mutation

Source Code

```
int gdImageColorMatch(...)
{
        ...
  buf = emalloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

Machine Code

```
   ...
jle 0x555555800bc9
lea esi, [rax+rax*4]
xor edx, edx
   ...
```

① (marks `lea esi, [rax+rax*4]`) → suspend

```
RAX:  0x1
RBX:  0x555556332ff0
RCX:  0x0
      ...
```

Program State at ①

Under-Constrained State Mutation

Source Code

```
int gdImageColorMatch(...)
{
        ...
  buf = emalloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

State Mutation!!

0x1 ⇨ 0xff

Machine Code

```
   ...
jle 0x555555800bc9
lea esi, [rax+rax*4]
xor edx, edx
   ...
```

① lea esi, [rax+rax*4]

suspend

```
RAX: 0xff
RBX: 0x555556332ff0
RCX: 0x0
      ...
```

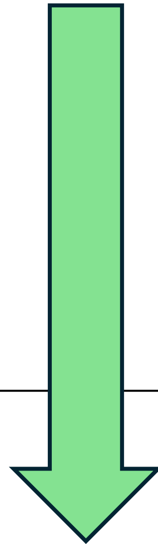
Program State at ①

Under-Constrained State Mutation

Source Code

```
int gdImageColorMatch(...)
{
        ...
  buf = emalloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
        ...
}
```

execute



Program Exit!!
(without a crash)

Machine Code

```
   ...
jle 0x555555800bc9
lea esi, [rax+rax*4]
xor edx, edx
   ...
```

① lea esi, [rax+rax*4]

suspend

resume

```
RAX: 0xff
RBX: 0x555556332ff0
RCX: 0x0
      ...
```

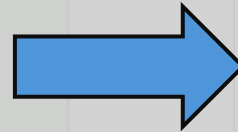
Program State at ①

Under-Constrained State Mutation

State mutation can obtain
the root cause revealing behavior

```
<?php
  $img1 = imagecreatetrucolor(0xfff, 0xfff);
  $img2 = imgcreate(0xfff, 0xfff);
  imagecolorallocate($img2, 0, 0, 0);

  imagesetpixel($img2, 0, 0, 0x80);
  imagecolormatch($img1, $img2);
?>
```



```
<?php
  $img1 = imagecreatetrucolor(0xfff, 0xfff);
  $img2 = imgcreate(0xfff, 0xfff);
  for ($i = 0; $i < 255; $i+=1) {
    imagecolorallocate($img2, 0, 0, 0);
  }
  imagesetpixel($img2, 0, 0, 0x80);
  imagecolormatch($img1, $img2);
?>
```

Is the discovered behavior valid?

Validity Problem

- Typical Fuzzing (e.g., AFL, libfuzzer, …)
    - It is for the bug (vulnerability) discovery
    - It must *validate* the reachability of the discovered behaviors

Validity Problem

- Typical Fuzzing (e.g., AFL, libfuzzer, …)
  - It is for the bug (vulnerability) discovery
  - It must *validate* the <u>reachability</u> of the discovered behaviors

Validity Problem

- Typical Fuzzing (e.g., AFL, libfuzzer, …)
  - It is for the bug (vulnerability) discovery
  - It must *validate* the reachability of the discovered behaviors

- Crash Exploration (for root cause analysis)
  - The bug (i.e., crash) is already given
  - What we need is to extract the crashing condition

Validity Problem

- Typical Fuzzing (e.g., AFL, libfuzzer, …)
  - It is for the bug (vulnerability) discovery
  - It must *validate* the reachability of the discovered behaviors

- Crash Exploration (for root cause analysis)
  - The bug (i.e., crash) is already given
  - What we need is to extract the crashing condition

Which state should we mutate?

There are tons of states even in a single (crashing) execution

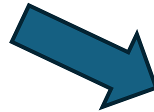
Design Choices for a State Mutation

- State mutation in a function granularity
  - Similar to LibFuzzer and in-memory fuzzing techniques
  - Speed acceleration using `fork()`

Design Choices for a State Mutation

- State mutation in a function granularity
  - Similar to LibFuzzer and in-memory fuzzing techniques
  - Speed acceleration using `fork()`

Target Function!!

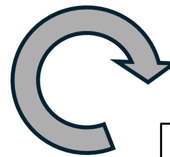


```
int gdImageColorMatch(...) {
          ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
          ...
}
```

Design Choices for a State Mutation

- State mutation in a function granularity
  - Similar to LibFuzzer and in-memory fuzzing techniques
  - Speed acceleration using `fork()`

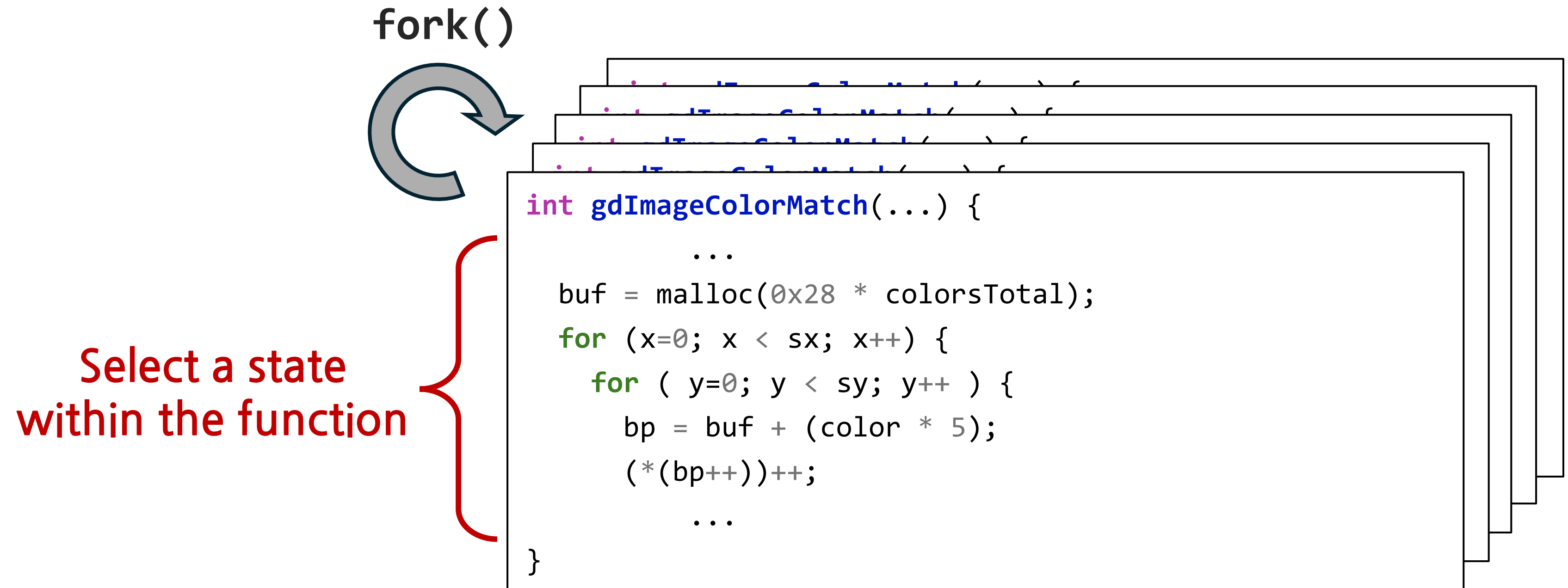
`fork()`



```
int gdImageColorMatch(...) {
          ...
  buf = malloc(0x28 * colorsTotal);
  for (x=0; x < sx; x++) {
    for ( y=0; y < sy; y++ ) {
      bp = buf + (color * 5);
      (*(bp++))++;
          ...
}
```

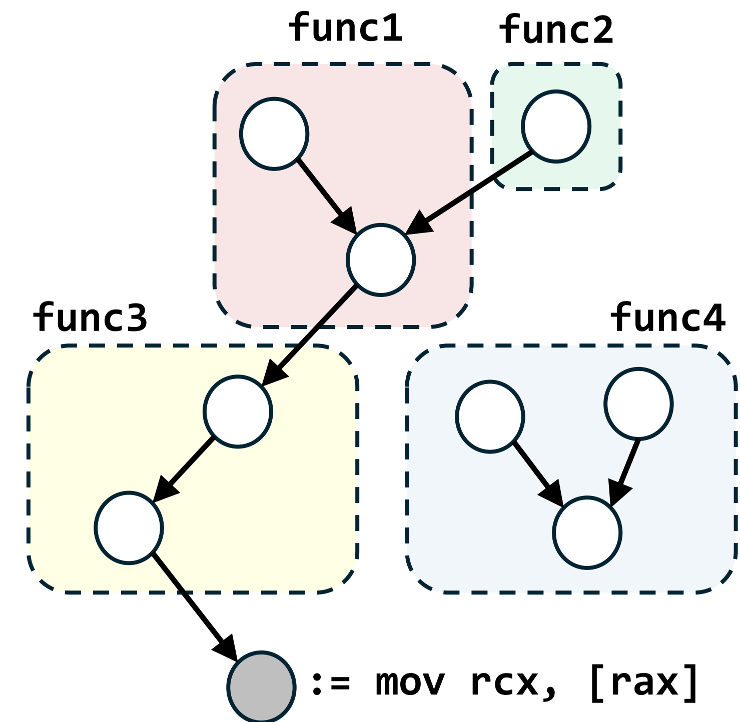
Design Choices for a State Mutation

- State mutation in a function granularity
  - Similar to LibFuzzer and in-memory fuzzing techniques
  - Speed acceleration using `fork()`



Design Choices for a State Mutation

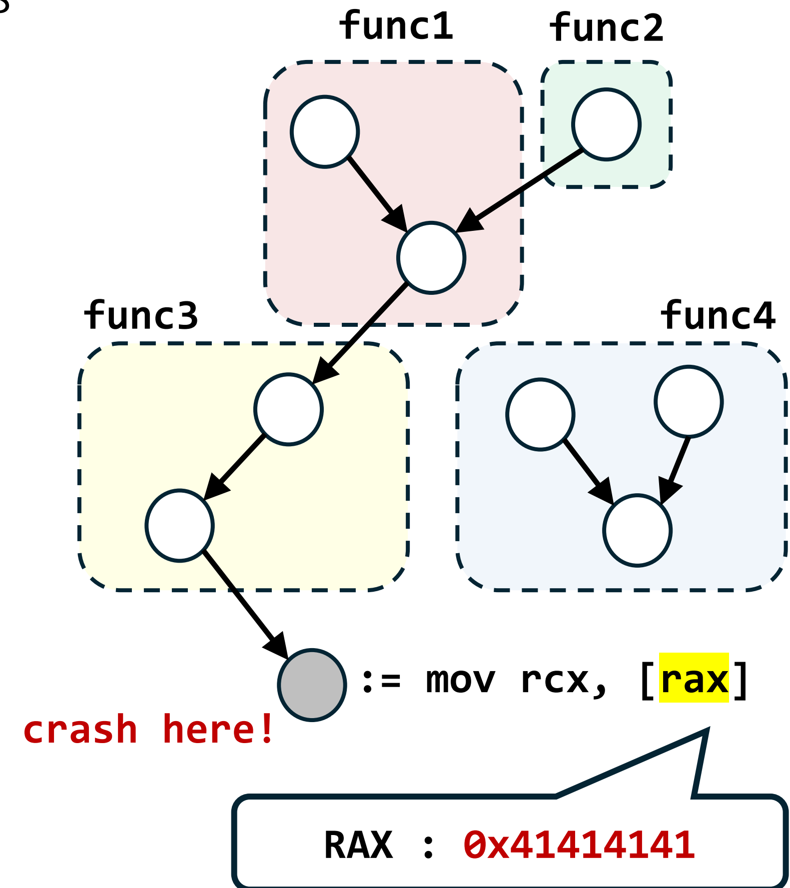
- State mutation in a function granularity
  - Similar to LibFuzzer and in-memory fuzzing techniques
  - Speed acceleration using `fork()`
- Target function extraction
  - Focusing on crash-related functions
  - Following data-flow & derefence edges



Design Choices for a State Mutation

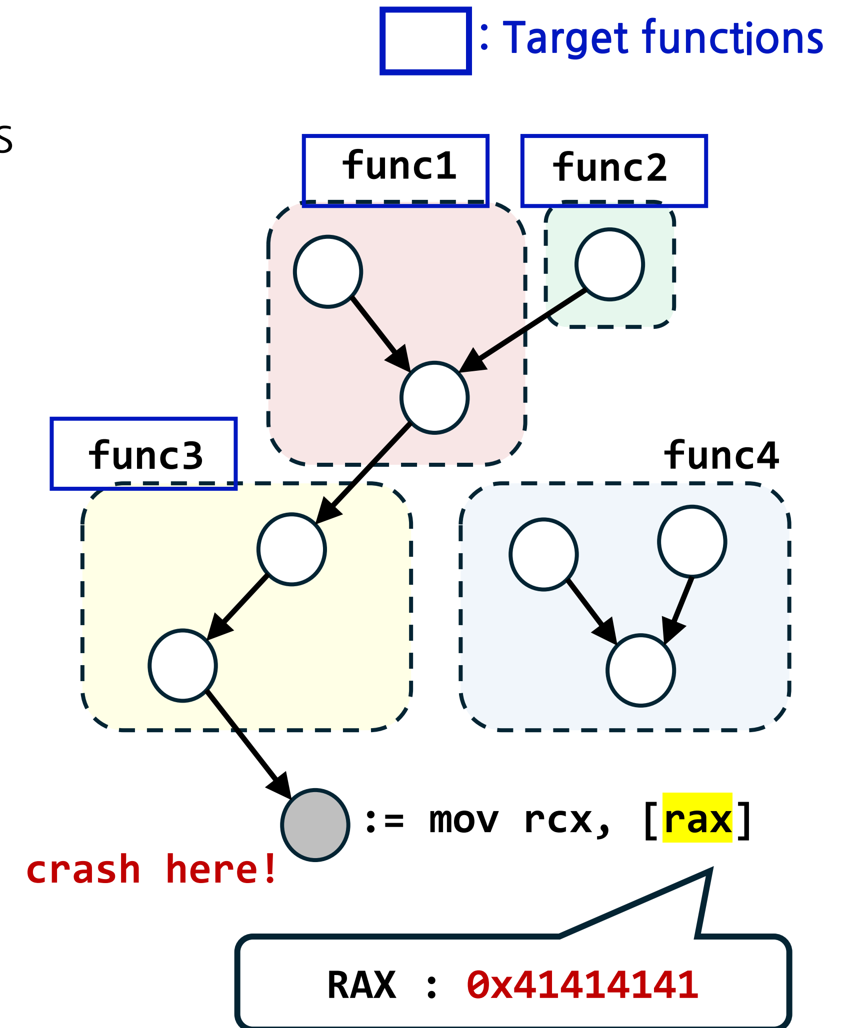
- State mutation in a function granularity
  - Similar to LibFuzzer and in-memory fuzzing techniques
  - Speed acceleration using `fork()`

- Target function extraction
  - Focusing on crash-related functions
  - Following data-flow & derefence edges



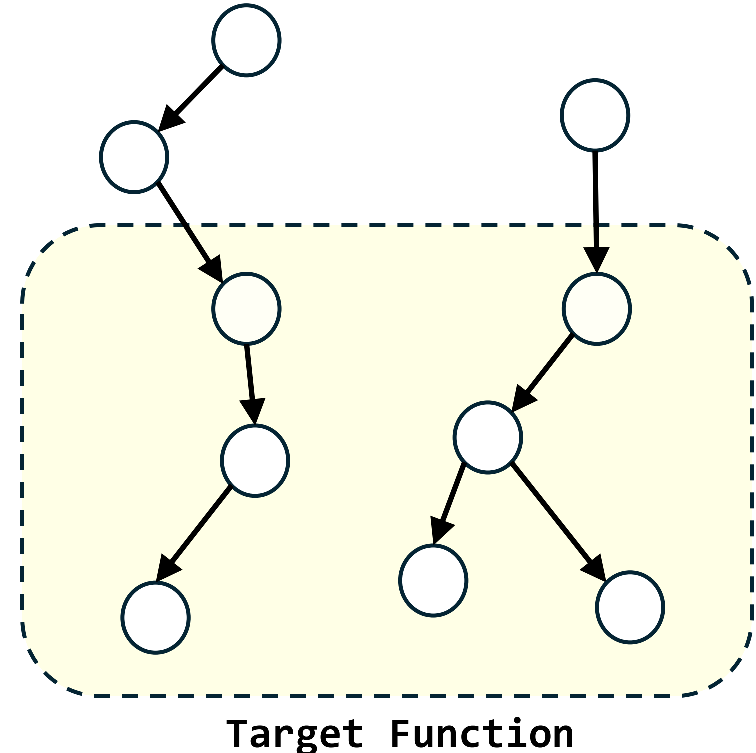
Design Choices for a State Mutation

- State mutation in a function granularity
  - Similar to LibFuzzer and in-memory fuzzing techniques
  - Speed acceleration using `fork()`
- Target function extraction
  - Focusing on crash-related functions
  - Following data-flow & derefence edges



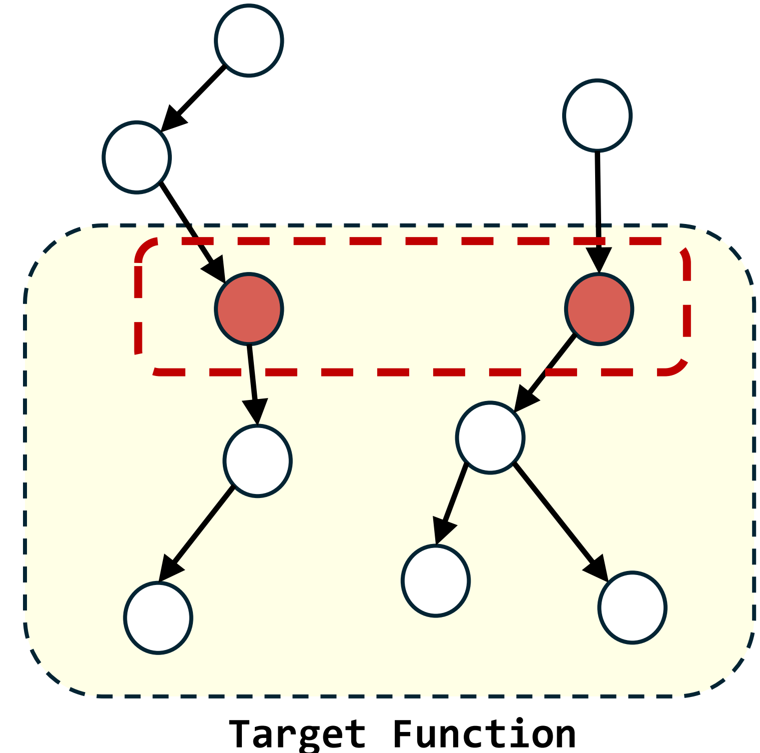
Design Choices for a State Mutation

- State mutation in a function granularity
  - Similar to LibFuzzer and in-memory fuzzing techniques
  - Speed acceleration using `fork()`
- Target function extraction
  - Focusing on crash-related functions
  - Following data-flow & derefence edges
- Focusing on instructions that use the values from outside of a function
  - Parameters, global variables, ...



Design Choices for a State Mutation

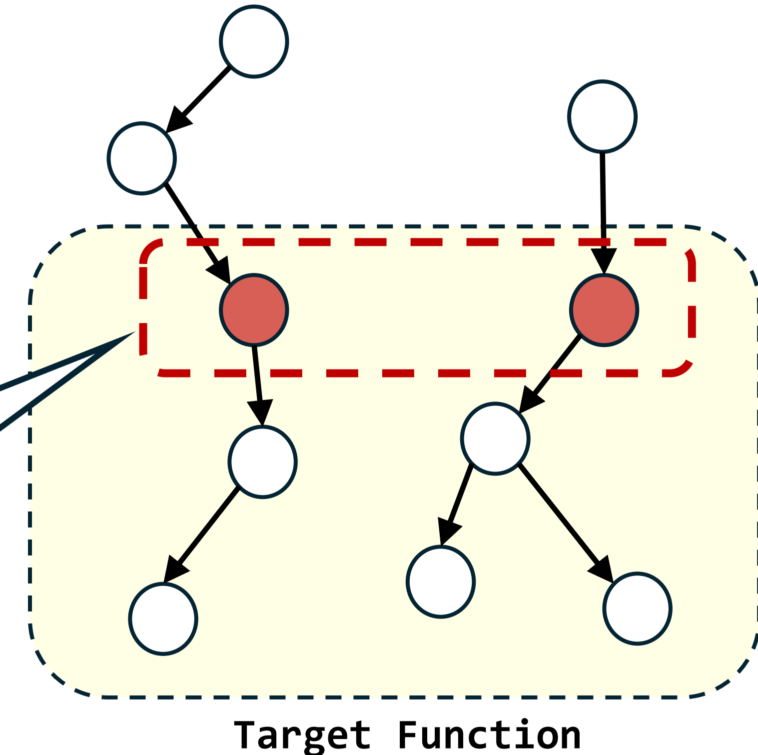
- State mutation in a function granularity
  - Similar to LibFuzzer and in-memory fuzzing techniques
  - Speed acceleration using `fork()`
- Target function extraction
  - Focusing on crash-related functions
  - Following data-flow & derefence edges
- Focusing on instructions that use the values from outside of a function
  - Parameters, global variables, …



Design Choices for a State Mutation

- State mutation in a function granularity
  - Similar to LibFuzzer and in-memory fuzzing techniques
  - Speed acceleration using `fork()`
- Target function extraction
  - Focusing on crash-related functions
  - Following dat

Function behaviors are (solely) decided by entry nodes!!



- Focusing on instructions that use <u>the values from outside of a function</u>
  - Parameters, global variables, …

Evaluation

Evaluation

- We collect 60 bugs from real-world applications
  - Targets include PHP, Python, SQLite, PDF reader, …

Evaluation

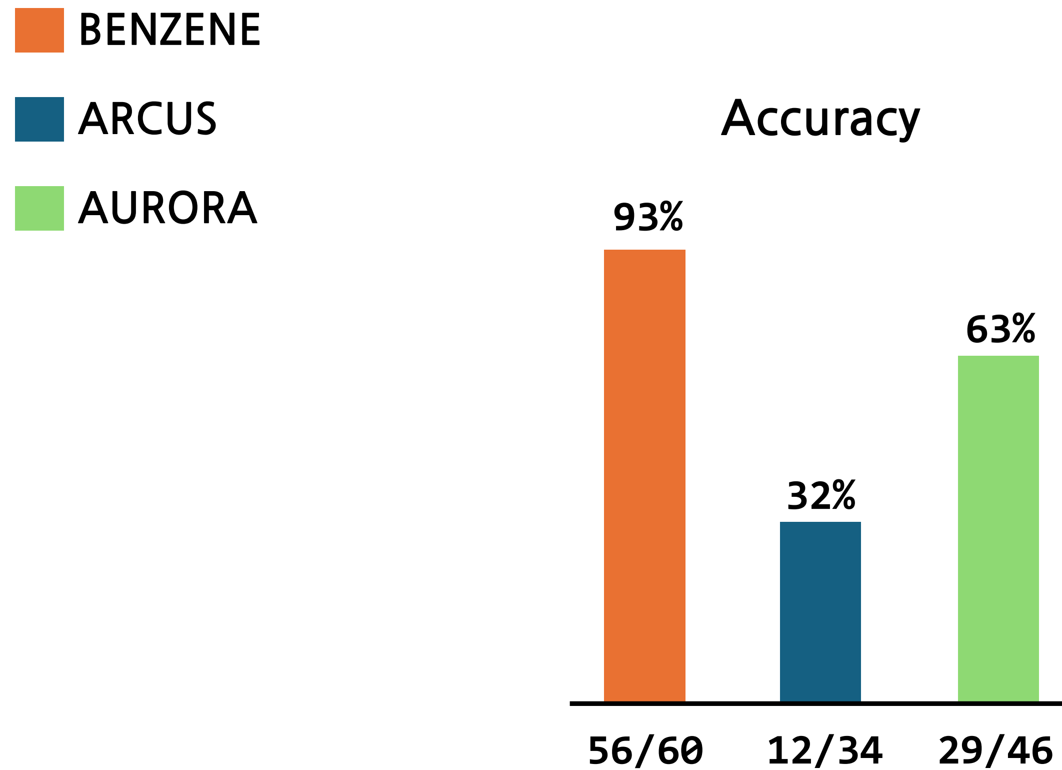
- We collect 60 bugs from real-world applications
  - Targets include PHP, Python, SQLite, PDF reader, …
- Our dataset spans 11 bug classes
  - heap overflow, integer overflow, UAF, …

Evaluation

- In our evaluation, BENZENE outperforms the SOTA tools

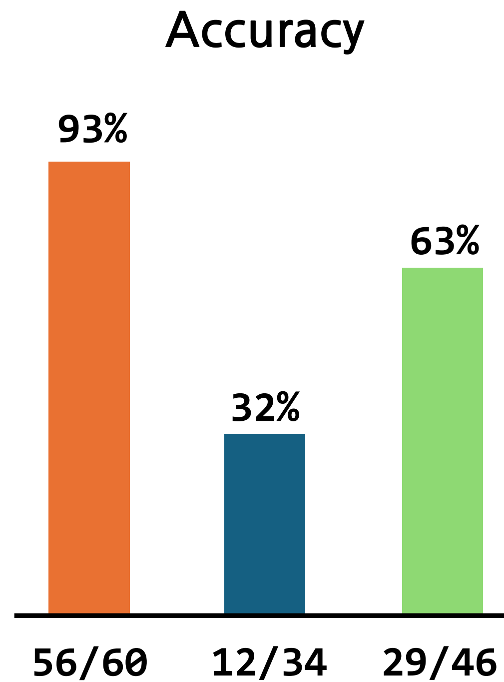
Evaluation

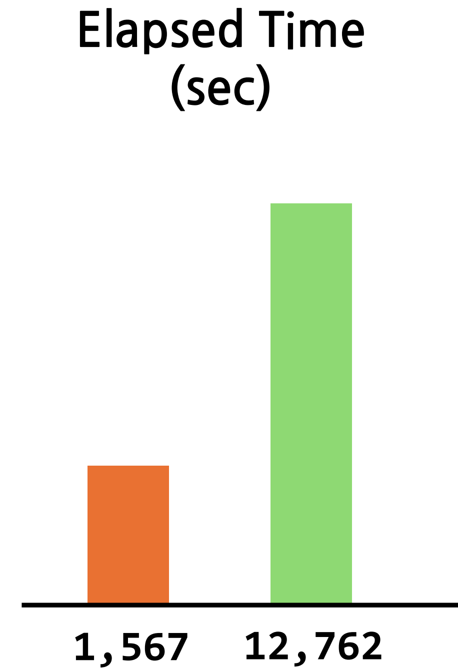
- In our evaluation, BENZENE outperforms the SOTA tools



Evaluation

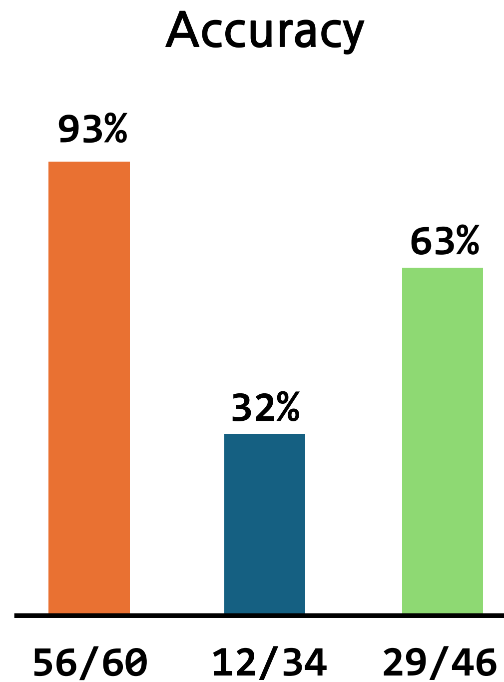
- In our evaluation, BENZENE outperforms the SOTA tools

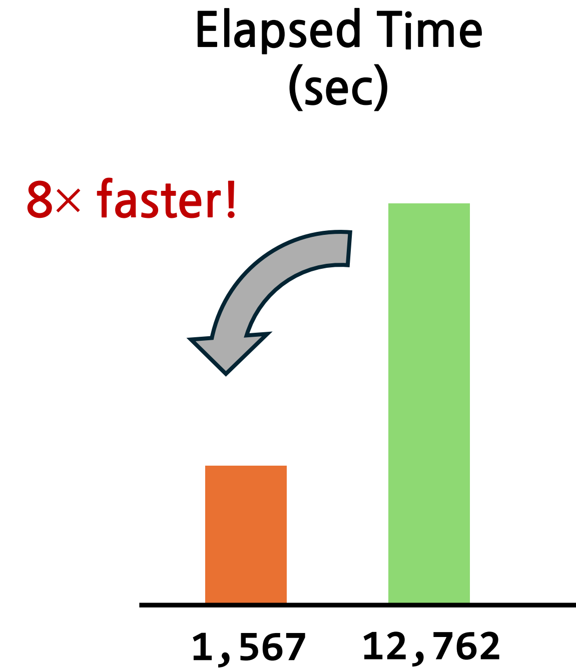




Evaluation

- In our evaluation, BENZENE outperforms the SOTA tools





Conclusion

- Finding crash-similar but non-crashing behaviors for RCA is difficult

Conclusion

- Finding crash-similar but non-crashing behaviors for RCA is difficult
- Under-constrained state mutation can efficiently discover the desired behaviors

Conclusion

- Finding crash-similar but non-crashing behaviors for RCA is difficult
- Under-constrained state mutation can efficiently discover the desired behaviors
- We introduce BENZENE, a root cause analysis system based on the under-constrained state mutation

Conclusion

- Finding crash-similar but non-crashing behaviors for RCA is difficult
- Under-constrained state mutation can efficiently discover the desired behaviors
- We introduce BENZENE, a root cause analysis system based on the under-constrained state mutation
- We evaluate BENZENE on 60 real-world bugs, successfully locating 93.3% root causes

Things Not Covered in This Talk

- Automatic predicate extraction
- Justification of validity problem for non-crashing behaviors
- Crashing behavior handling
- Detailed state mutation strategies
- Similarity-based ranking algorithm

Things Not Covered in This Talk

- Automatic predicate extraction
- Justification of validity problem for non-crashing behaviors
- Crashing behavior handling
- Detailed state mutation strategies
- Similarity-based ra

If interested,
we encourage you to read our paper!!

Thank you!

- grill66@korea.ac.kr
- https://github.com/zer0fall/BENZENE