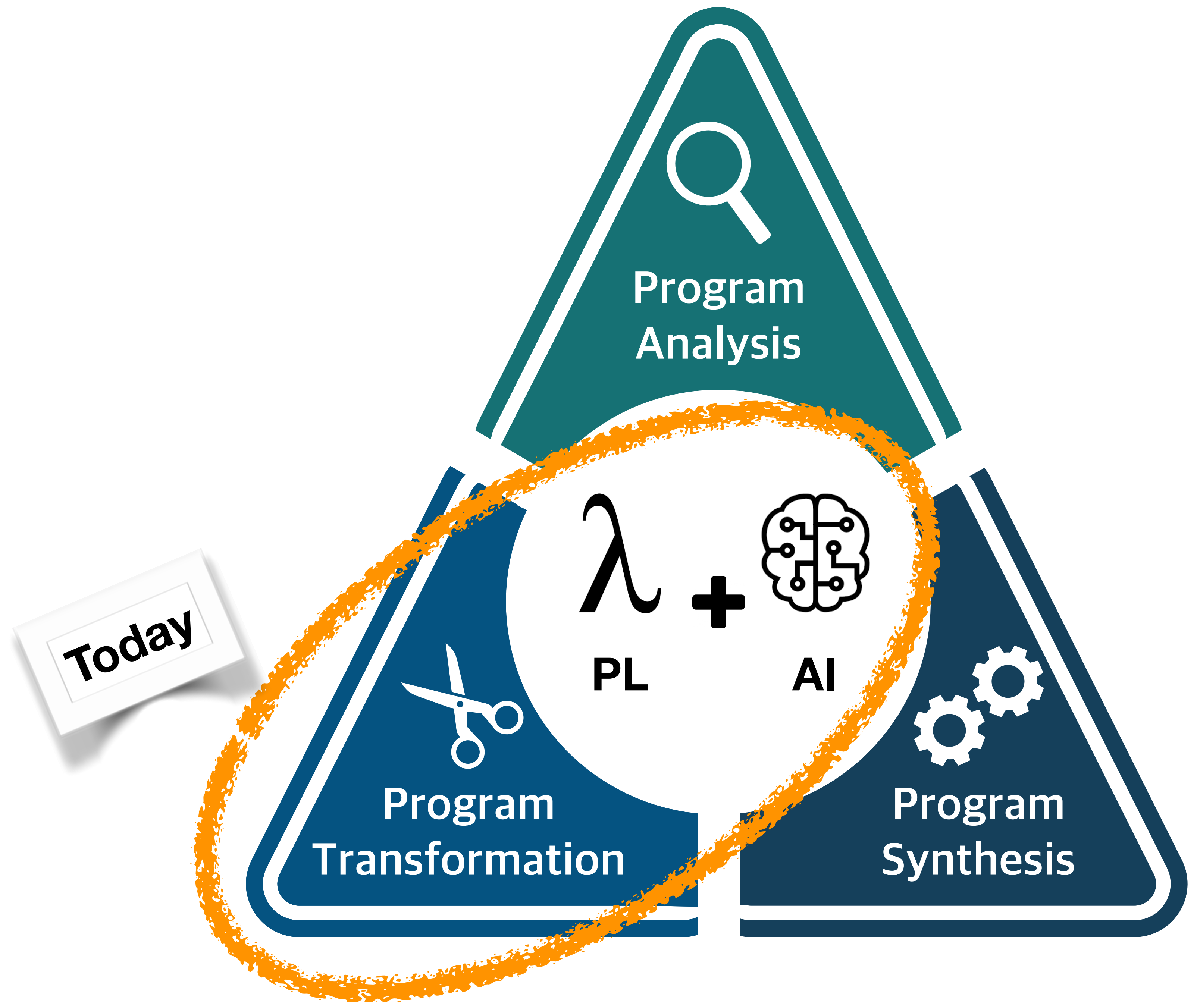
IS593: Language-based Security

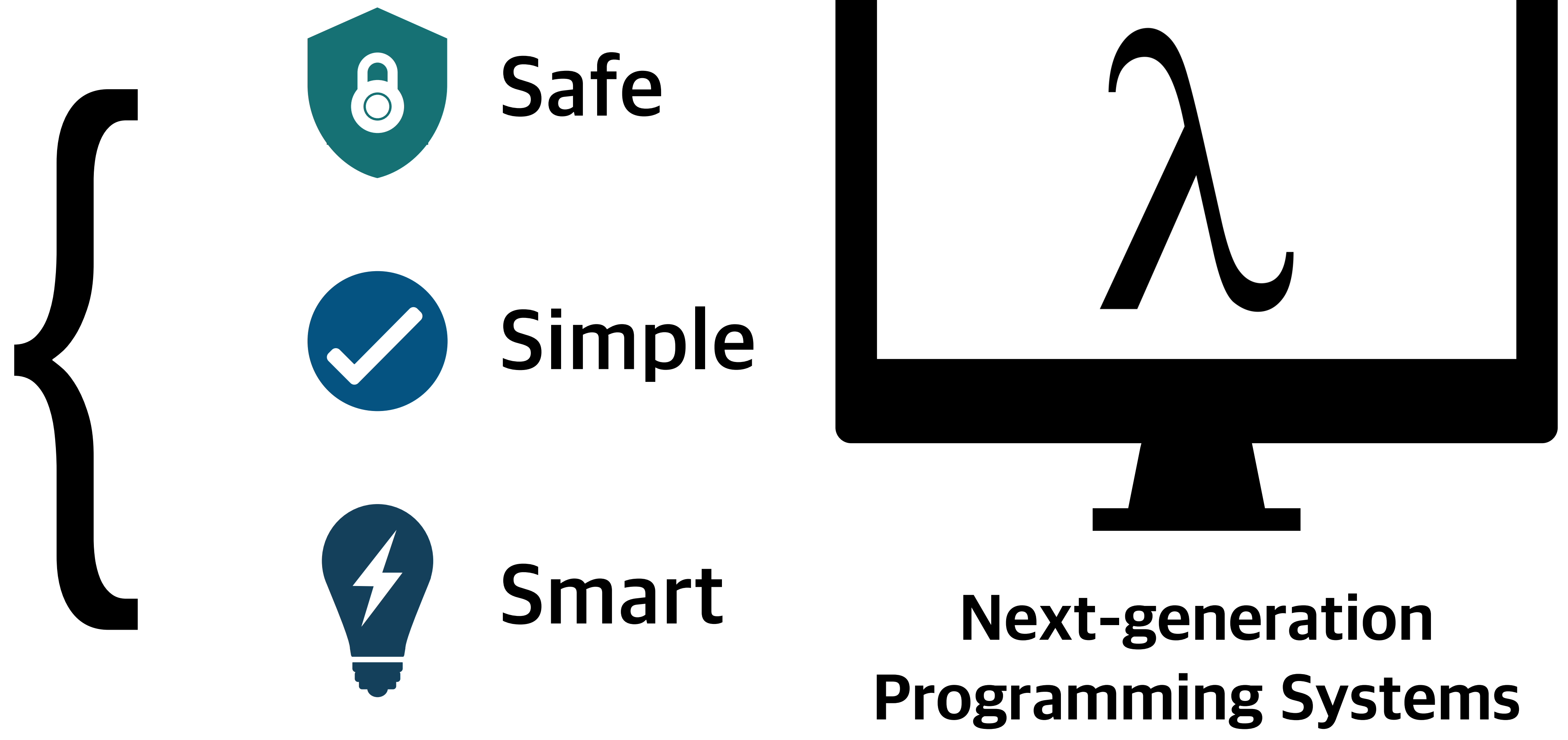
13. Program Debloating

Kihong Heo

KAIST

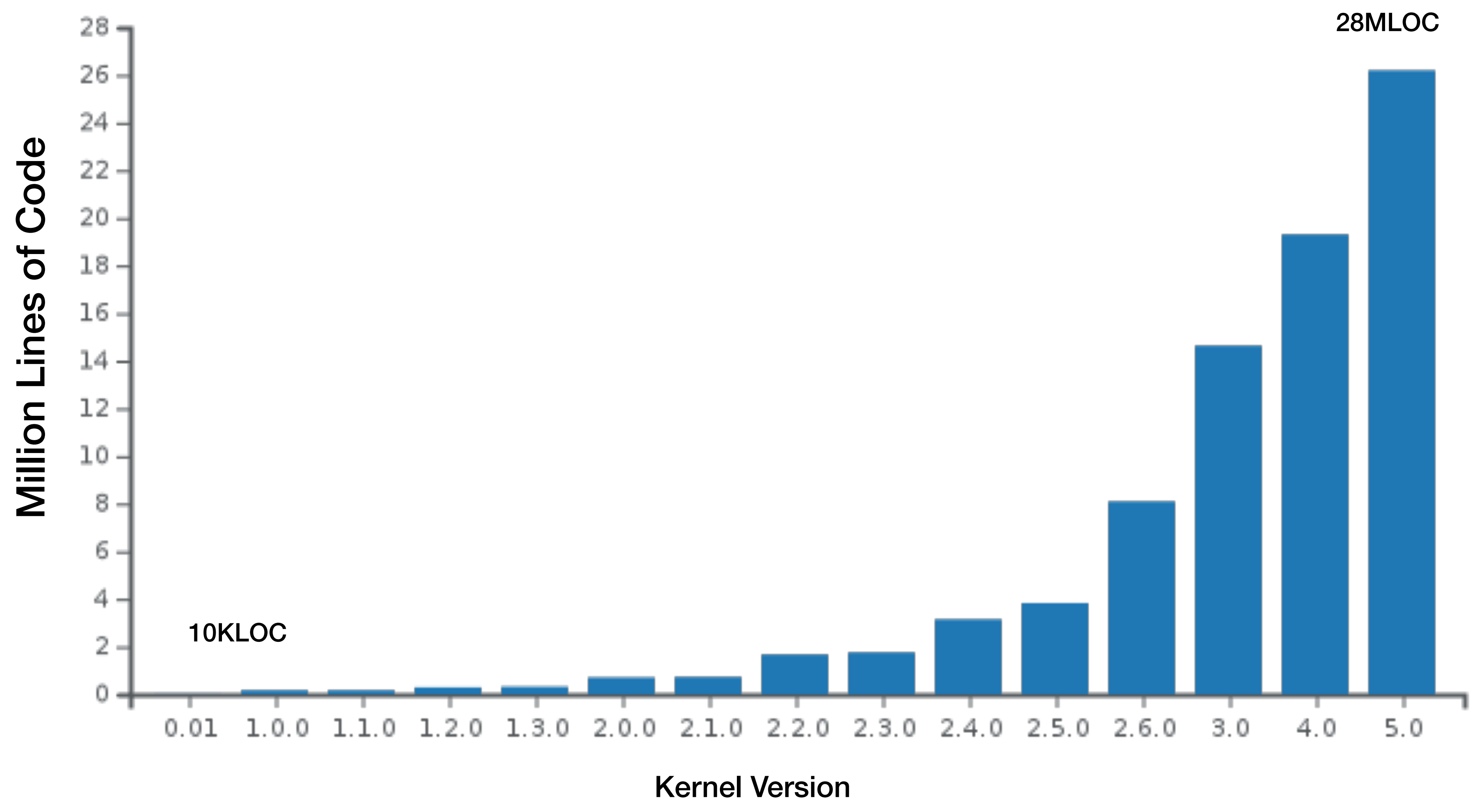
New Waves in Language-based Security



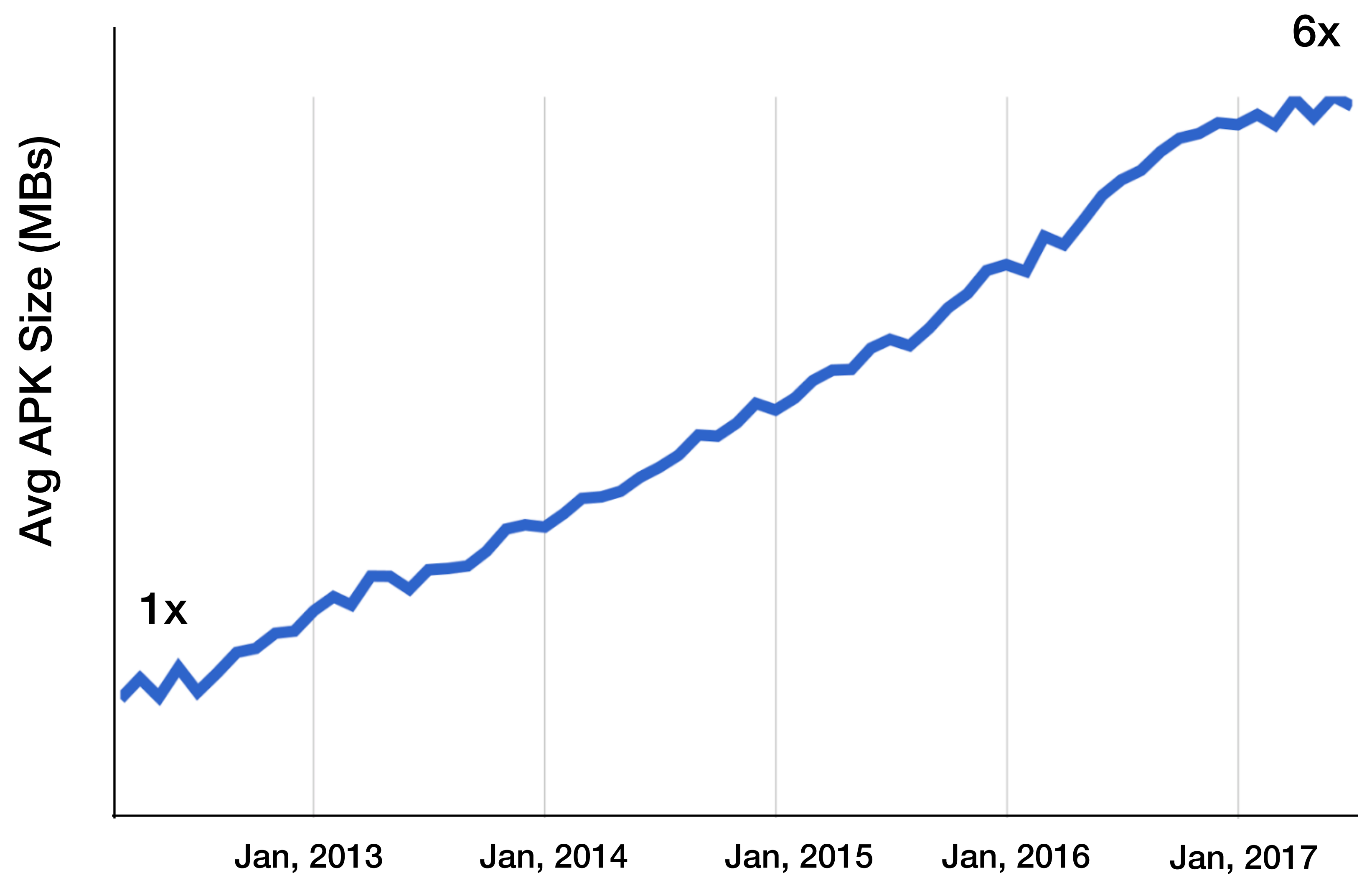


Growth of SW Complexity

Size of Linux Kernel



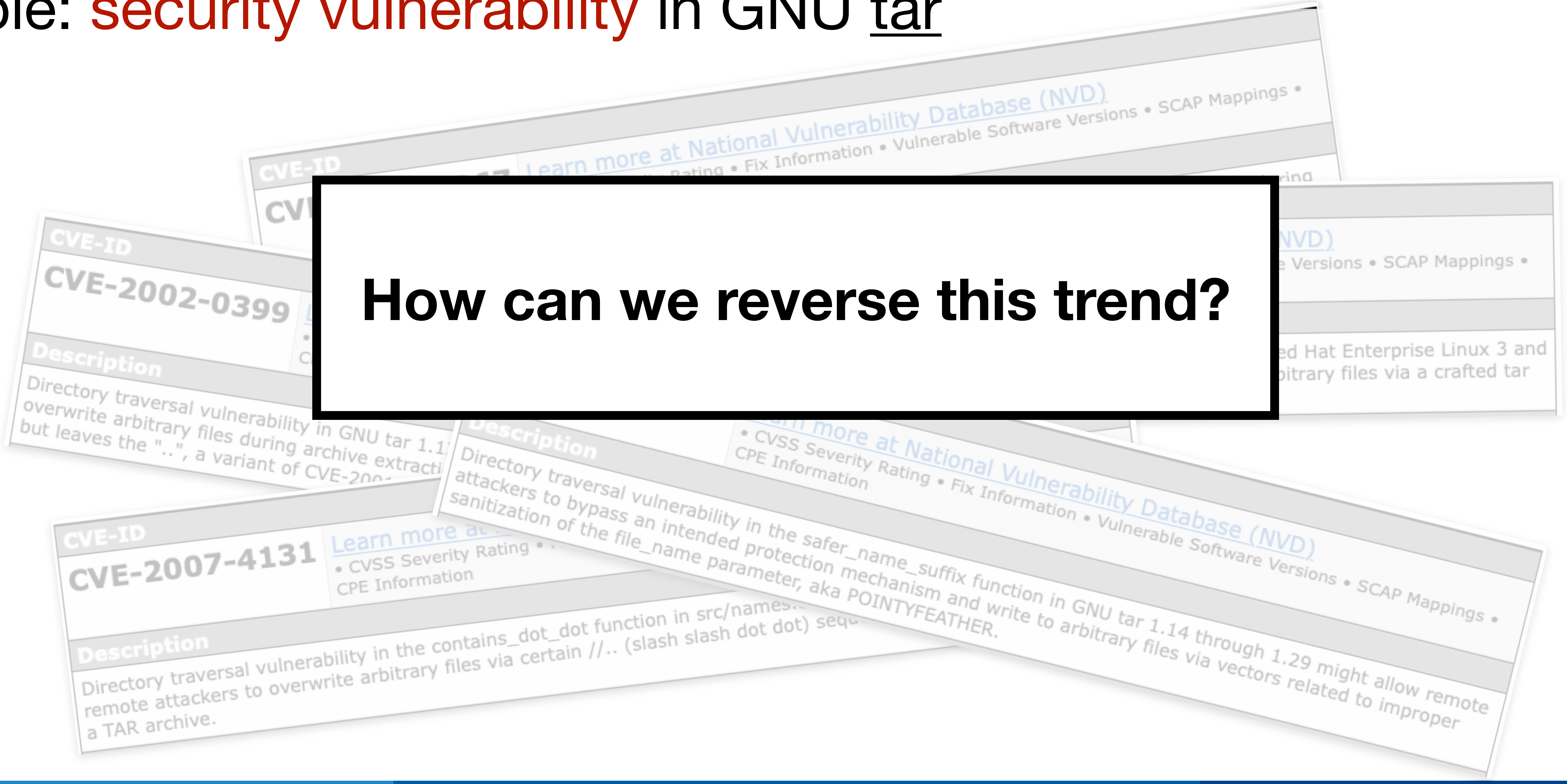
Avg. Size of Android Apps



Consequences of SW Bloat

Performance | **Maintainability** | **Security**

- Example: security vulnerability in GNU tar



CVE-ID

CV

Learn more at National Vulnerability Database (NVD)

Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings •

ring

CVE-ID

CVE-2002-0399

Description

Directory traversal vulnerability in GNU tar 1.1
overwrite arbitrary files during archive extracti
but leaves the "..", a variant of CVE-200

NVD)

e Versions • SCAP Mappings •

ed Hat Enterprise Linux 3 and
bitrary files via a crafted tar

CVE-ID

CVE-2007-4131

Learn more a

• CVSS Severity Rating •

CPE Information

Description

Directory traversal vulnerability in the contains_dot_dot function in src/names
remote attackers to overwrite arbitrary files via certain //.. (slash slash dot dot) seq
a TAR archive.

Description

Directory traversal vulnerability in the safer_name_suffix function in GNU tar 1.14 through 1.29 might allow remote
attackers to bypass an intended protection mechanism and write to arbitrary files via vectors related to improper
sanitization of the file_name parameter, aka POINTYFEATHER.

Learn more at National Vulnerability Database (NVD)

• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings •

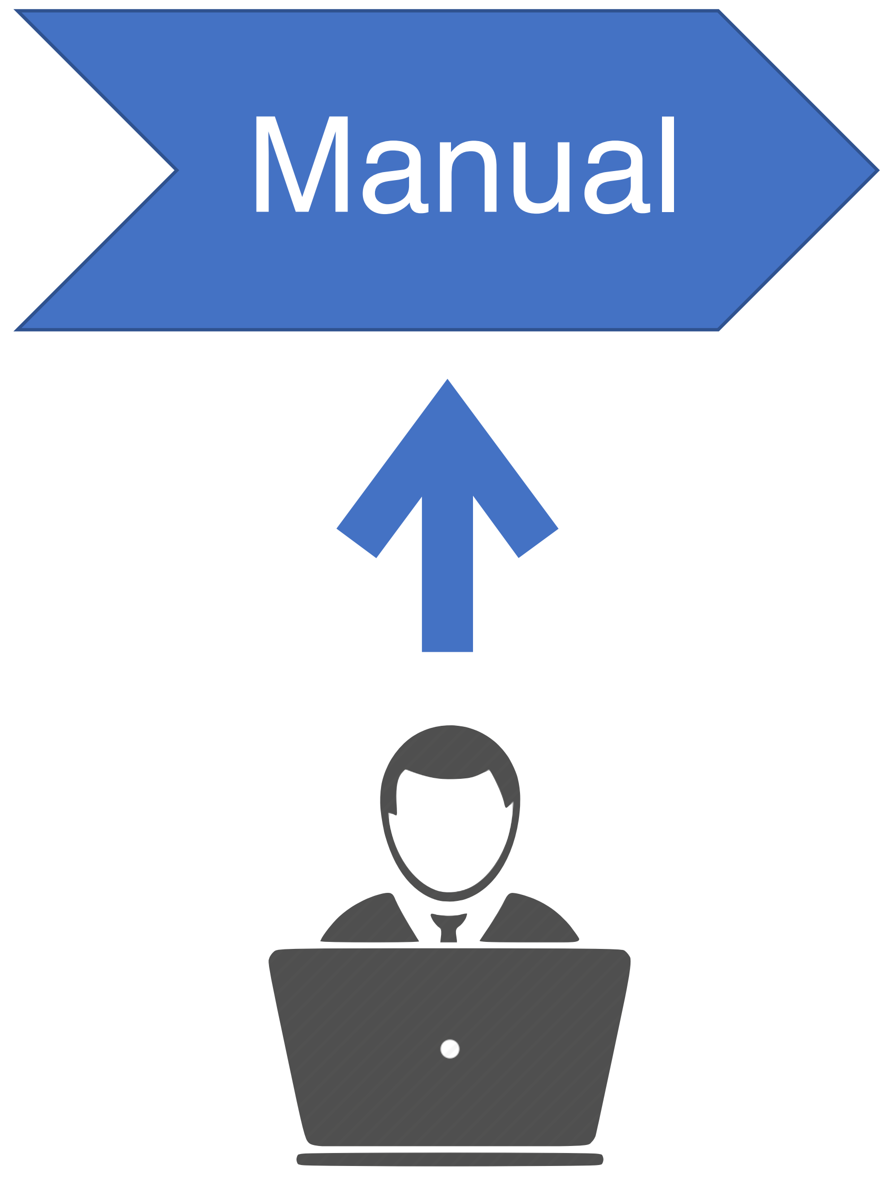
CPE Information

How can we reverse this trend?

State-of-the-Practice

General-purpose tar

- Out-of-the-box Linux
- 97 cmd line options
- 45,778 LOC
- 13,227 statements
- CVE-2016-6321



Customized tar

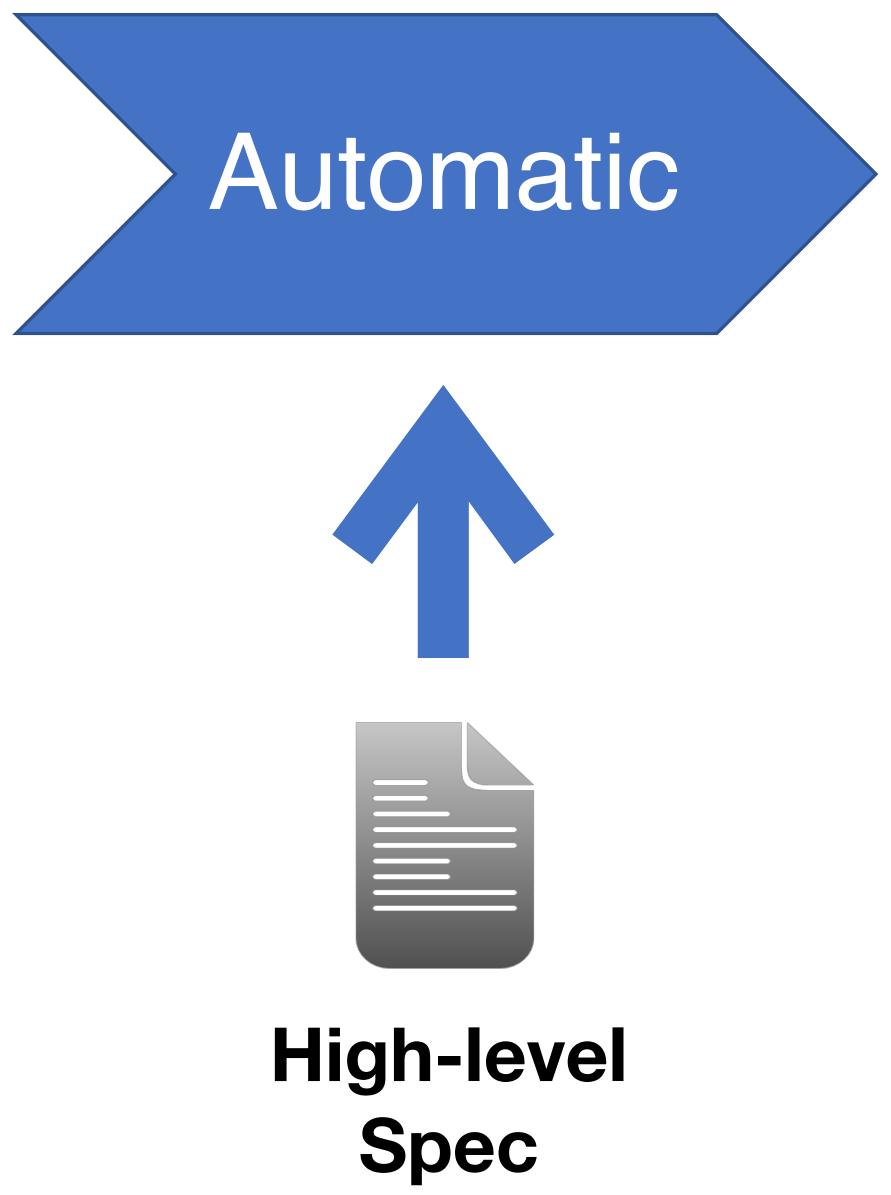
- BusyBox Utility Package*
- 8 cmd line options
- 3,287 LOC
- 403 statements
- No known CVEs

*https://busybox.net

Goal

General-purpose tar

- Out-of-the-box Linux
- 97 cmd line options
- 45,778 LOC
- 13,227 statements
- CVE-2016-6321



Customized tar

- BusyBox Utility Package*
- 8 cmd line options
- **1,646** ~~3,287~~ LOC
- **518** ~~403~~ statements
- No known CVEs

*https://busybox.net

Chisel: A Program Debloating System*

- **minimality**: trim code as aggressively as possible
- **efficiency**: scale to large programs
- **robustness**: avoid introducing new vulnerabilities
- **naturalness**: produce maintainable code
- **generality**: handle a variety of programs and specs

*https://chisel.cis.upenn.edu

Example: tar-1.14

```
int absolute_names;
int ignore_zeros_option;
struct tar_stat_info stat_info;

char *safer_name_suffix (char *file_name, int link_target) {
    int prefix_len;
    char *p;

    if (absolute_names) {
        p = file_name;
    } else {
       /* CVE-2016-6321 */
       /* Incorrect sanitization if “file_name” contains ".." */
       ...
    }
    ...
    return p;
}

void extract_archive() {
    char *file_name = safer_name_suffix(stat_info.file_name, 0);
    /* Overwrite “file_name” if exists */
    ...
}

void list_archive() { ... }
```

```
void read_and(void *(do_something)(void)) {
    enum read_header status;
    while (...) {
        status = read_header();
        switch (status) {
        case HEADER_SUCCESS: (*do_something)(); continue;
        case HEADER_ZERO_BLOCK:
          if (ignore_zeros_option) continue;
          else break;
        ...
        default: break;
        }
    }
    ...
}

/* Supports all options: -x, -t, -P, -i, ... */
int main(int argc, char **argv) {
    int optchar;
    while (optchar = getopt_long(argc, argv) != -1) {
        switch(optchar) {
        case 'x': read_and(&extract_archive); break;
        case 't': read_and(&list_archive); break;
        case 'P': absolute_names = 1; break;
        case 'i': ignore_zeros_option = 1; break;
        ...
        }
    }
    ...
}
```

Example: tar-1.14

Global variable declarations removed

```
int absolute_names;
int ignore_zeros_option;
struct tar_stat_info stat_info;

char *safer_name_suffix (char *file_name, int link_target) {
    int prefix_len;
    char *p;

    if (absolute_names) {
        p = file_name;
    } else {
       /* CVE-2016-6321 */
       /* Incorrect sanitization if "file_name" contains ".." */
       ...
    }
    ...
    return p;
}

void extract_archive() {
    char *file_name = safer_name_suffix(stat_info.file_name, 0);
    /* Overwrite "file_name" if exists */
    ...
}

void list_archive() { ... }
```

Code containing CVE removed

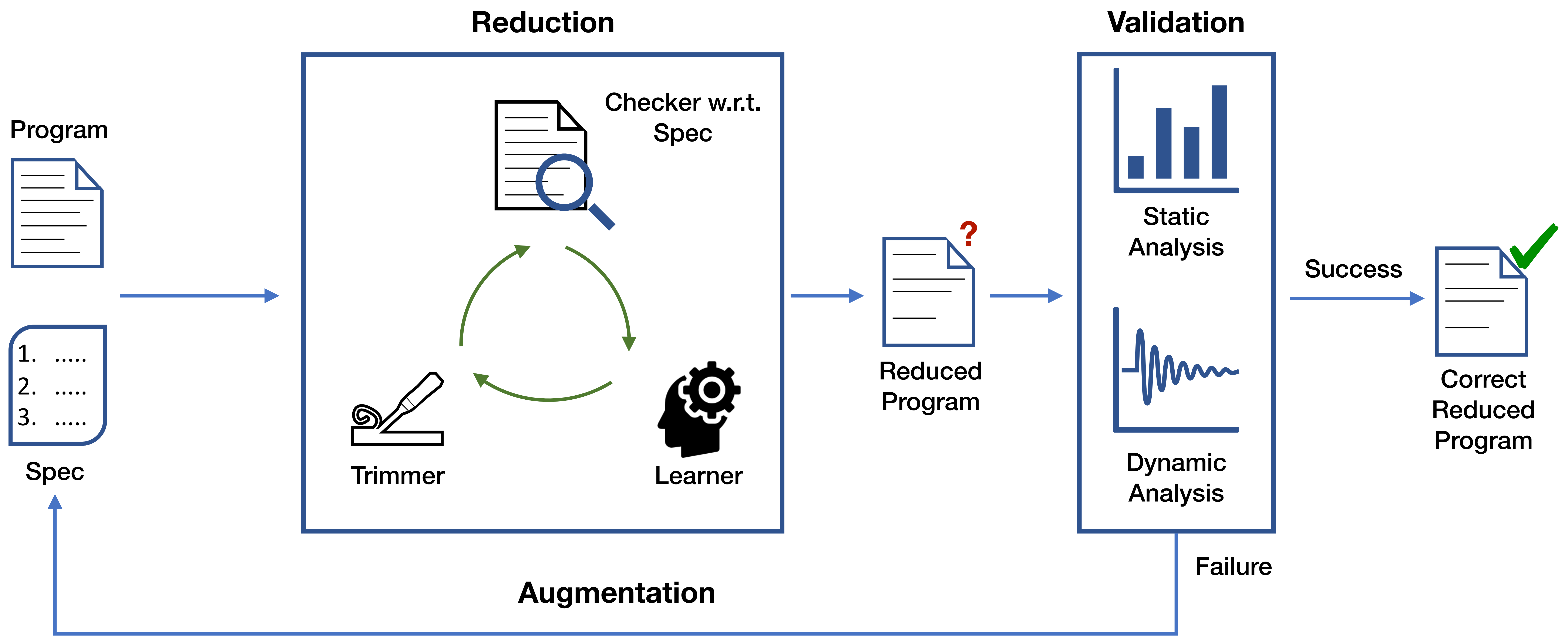
Overwriting functionalities removed

```
void read_and(void *(do_something)(void)) {
    enum read_header status;
    while (...) {
        status = read_header();
        switch (status) {
        case HEADER_SUCCESS: (*do_something)(); continue;
        case HEADER_ZERO_BLOCK:
          if (ignore_zeros_option) continue;
          else break;
        ...
        default: break;
        }
    }
    ...
}

/* Supports all options: -x, -t, -P, -i, ... */
int main(int argc, char **argv) {
    int optchar;
    while (optchar = getopt_long(argc, argv) != -1) {
        switch(optchar) {
        case 'x': read_and(&extract_archive); break;
        case 't': read_and(&list_archive); break;
        case 'P': absolute_names = 1; break;
        case 'i': ignore_zeros_option = 1; break;
        ...
        }
    }
    ...
}
```

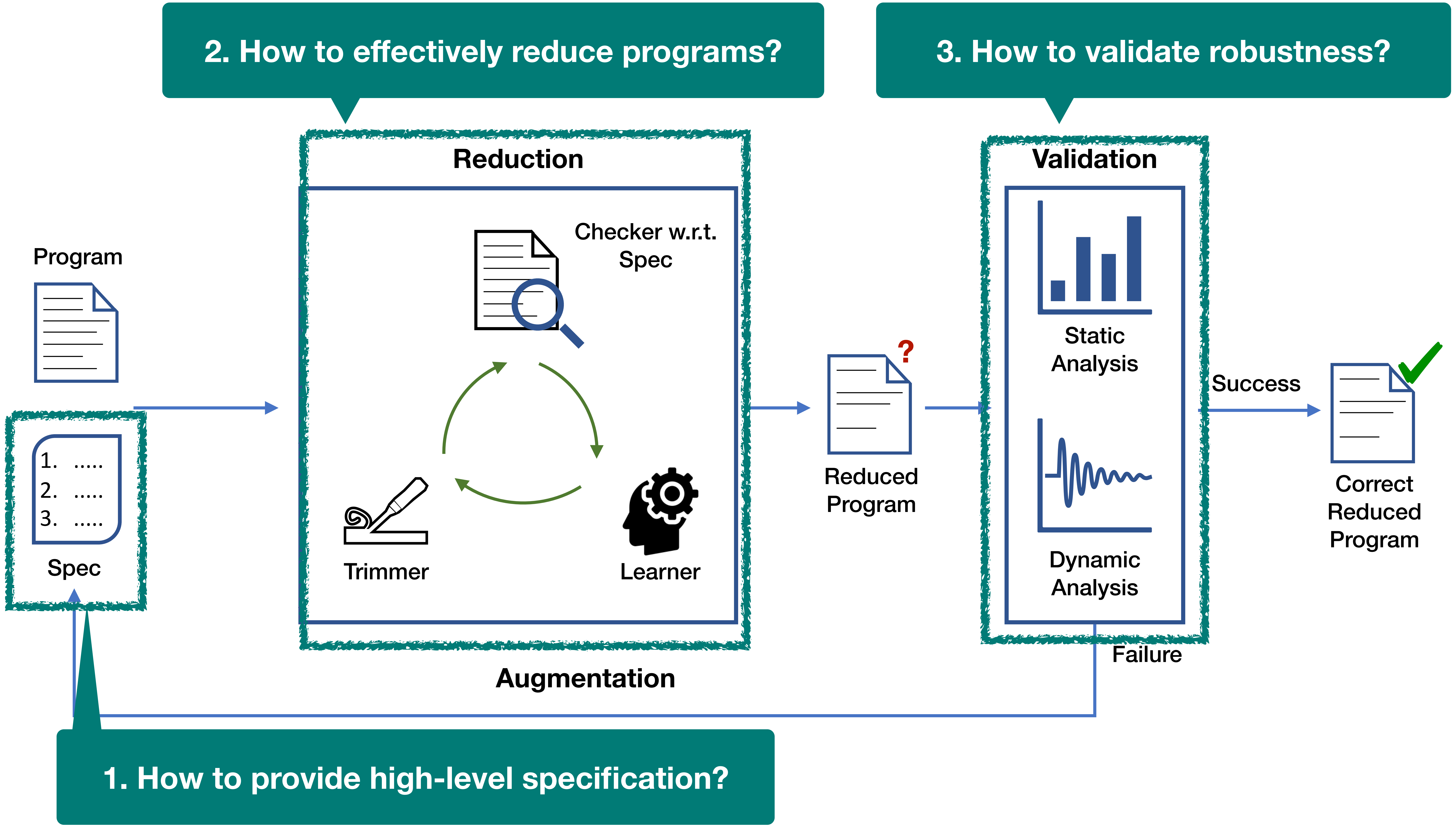
Unnecessary functionalities removed

Unsupported options removed

System Architecture



Key Questions



High-level Specification

```bash
#!/bin/bash

function compile {
  clang -o tar.debloat tar-1.14.c
  return $?
}

# tests for the desired functionalities
function desired {
  # 1. archiving multiple files
  touch foo bar
  ./tar.debloat cf foo.tar foo bar
  rm foo bar
  ./tar.debloat xf foo.tar
  test -f foo -a -f bar || exit 1

  # 2. extracting from stdin
  touch foo
  ./tar.debloat cf foo.tar foo
  rm foo
  cat foo.tar | ./tar.debloat x
  test -f foo || exit 1

  # other tests
  …
  return 0
}
```

```bash
# tests for the undesired functionalities
function undesired {
  for test_script in `ls other_tests/*.sh`
  do
    { sh -x -e $test_script; } >& log
    grep 'Segmentation fault' log && exit 1
  done
  return 0
}

compile || exit 1
desired || exit 1
undesired || exit 1
```

High-level Specification

```
#!/bin/bash

function compile {
  clang -o tar.debloat tar-1.14.c
  return $?
}

# tests for the desired functionalities
function desired {
  # 1. archiving multiple files
  touch foo bar
  ./tar.debloat cf foo.tar foo bar
  rm foo bar
  ./tar.debloat xf foo.tar
  test -f foo -a -f bar || exit 1

  # 2. extracting from stdin
  touch foo
  ./tar.debloat cf foo.tar foo
  rm foo
  cat foo.tar | ./tar.debloat x
  test -f foo || exit 1

  # other tests
  …
  return 0
}
```

1. The program is compilable.

```
# tests for the undesired functionalities
function undesired {
  for test_script in `ls other_tests/*.sh`
  do
    { sh -x -e $test_script; } >& log
    grep 'Segmentation fault' log && exit 1
  done
  return 0
}

compile || exit 1
desired || exit 1
undesired || exit 1
```

High-level Specification

```
#!/bin/bash

function compile {
  clang -o tar.debloat tar-1.14.c
  return $?
}

# tests for the desired functionalities
function desired {
  # 1. archiving multiple files
  touch foo bar
  ./tar.debloat cf foo.tar foo bar
  rm foo bar
  ./tar.debloat xf foo.tar
  test -f foo -a -f bar || exit 1

  # 2. extracting from stdin
  touch foo
  ./tar.debloat cf foo.tar foo
  rm foo
  cat foo.tar | ./tar.debloat x
  test -f foo || exit 1

  # other tests
  …
  return 0
}
```

2. The program produces the same results with the desired functionalities. (e.g., using regression test suites)

```
# tests for the undesired functionalities
function undesired {
  for test_script in `ls other_tests/*.sh`
  do
    { sh -x -e $test_script; } >& log
    grep 'Segmentation fault' log && exit 1
  done
  return 0
}

compile || exit 1
desired || exit 1
undesired || exit 1
```

High-level Specification

```bash
#!/bin/bash
```

3. The program does not crash with the undesired functionalities. (e.g., using Clang sanitizers)

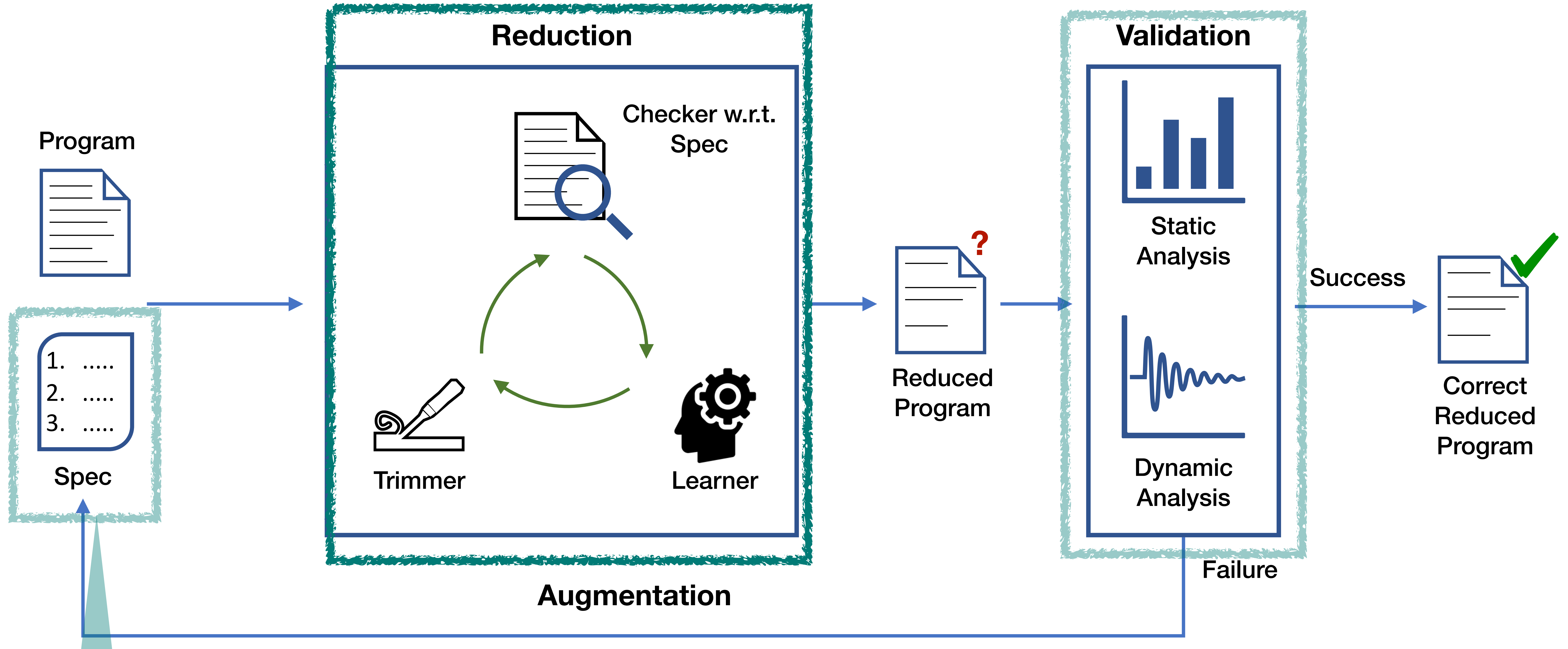
```bash
function desired {
  # 1. archiving multiple files
  touch foo bar
  ./tar.debloat cf foo.tar foo bar
  rm foo bar
  ./tar.debloat xf foo.tar
  test -f foo -a -f bar || exit 1

  # 2. extracting from stdin
  touch foo
  ./tar.debloat cf foo.tar foo
  rm foo
  cat foo.tar | ./tar.debloat x
  test -f foo || exit 1

  # other tests
  …
  return 0
}
```

```bash
# tests for the undesired functionalities
function undesired {
  for test_script in `ls other_tests/*.sh`
  do
    { sh -x -e $test_script; } >& log
    grep 'Segmentation fault' log && exit 1
  done
  return 0
}
compile || exit 1
desired || exit 1
undesired || exit 1
```

Key Questions

2. How to effectively reduce programs?

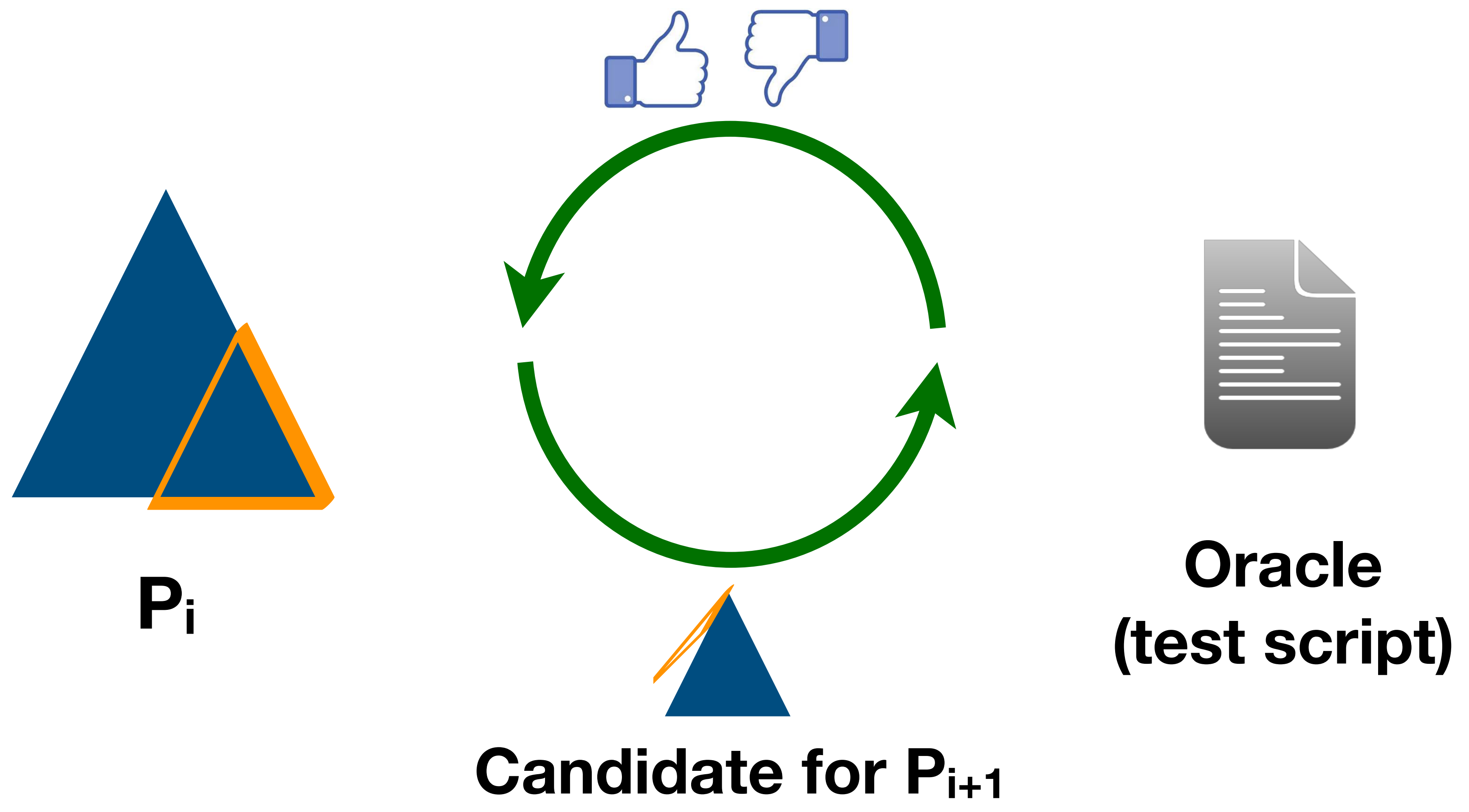
3. How to validate robustness?



1. How to provide high-level specification?

Program Debloating by Delta Debugging*

- Oracle $\mathcal{O}$ takes a program and returns **PASS** or **FAIL**
- Given a program P, find a **1-minimal** P^* such that $\mathcal{O}(P^*)$ = **PASS**
- **1-minimal:** removing any single element of P^* does not pass $\mathcal{O}$
- Time complexity: $O(|P|^2)$



*Zeller and Hildebrandt, simplifying and isolating failure-inducing input, TSE, 2002

Delta Debugging

- Originally proposed to **minimize** failing test cases
  - “What is the minimal test case that reproduce the failure?”
- Why minimize?
  - Easier debugging: Does failure really depend on 10,000 lines of code?
  - Identify duplicates: A minimal test case implies a most general context
- General and efficient algorithm
  - Arbitrary granularity: line-level, character level, etc
  - Polynomial time complexity: $O(n^2)$

Example

- Property of interest: termination with return code zero

Original

```
int f1() { return 0; }
int f2() { return 1; }
int f3() { return 1; }
int f4() { return 1; }
int f5() { return 1; }
int f6() { return 1; }
int f7() { return 1; }
int main() { return f1(); }
```

Minimal version

```
int f1() { return 0; }






int main() { return f1(); }
```

Example (Cont’d)

(🟧 included)

	🟧 f1	🟧 f2	🟧 f3	🟧 f4	🟧 f5	🟧 f6	🟧 f7	🟧 main	✔
1	🟧 f1	🟧 f2	🟧 f3	🟧 f4	f5	f6	f7	main	✘
2	f1	f2	f3	f4	🟧 f5	🟧 f6	🟧 f7	🟧 main	✘
3	🟧 f1	🟧 f2	f3	f4	f5	f6	f7	main	✘
4	f1	f2	🟧 f3	🟧 f4	f5	f6	f7	main	✘
5	f1	f2	f3	f4	🟧 f5	🟧 f6	f7	main	✘
6	f1	f2	f3	f4	f5	f6	🟧 f7	🟧 main	✘
7	f1	f2	🟧 f3	🟧 f4	🟧 f5	🟧 f6	🟧 f7	🟧 main	✘
8	🟧 f1	🟧 f2	f3	f4	🟧 f5	🟧 f6	🟧 f7	🟧 main	✔
9	🟧 f1	🟧 f2	f3	f4	f5	f6	🟧 f7	🟧 main	✔
10	🟧 f1	f2	f3	f4	f5	f6	f7	main	✘
11	f1	🟧 f2	f3	f4	f5	f6	f7	main	✘
12	f1	f2	f3	f4	f5	f6	🟧 f7	main	✘
13	f1	f2	f3	f4	f5	f6	f7	🟧 main	✘
14	f1	🟧 f2	f3	f4	f5	f6	🟧 f7	🟧 main	✘
15	🟧 f1	f2	f3	f4	f5	f6	🟧 f7	🟧 main	✔
16	🟧 f1	f2	f3	f4	f5	f6	f7	🟧 main	✔

*All duplications are omitted

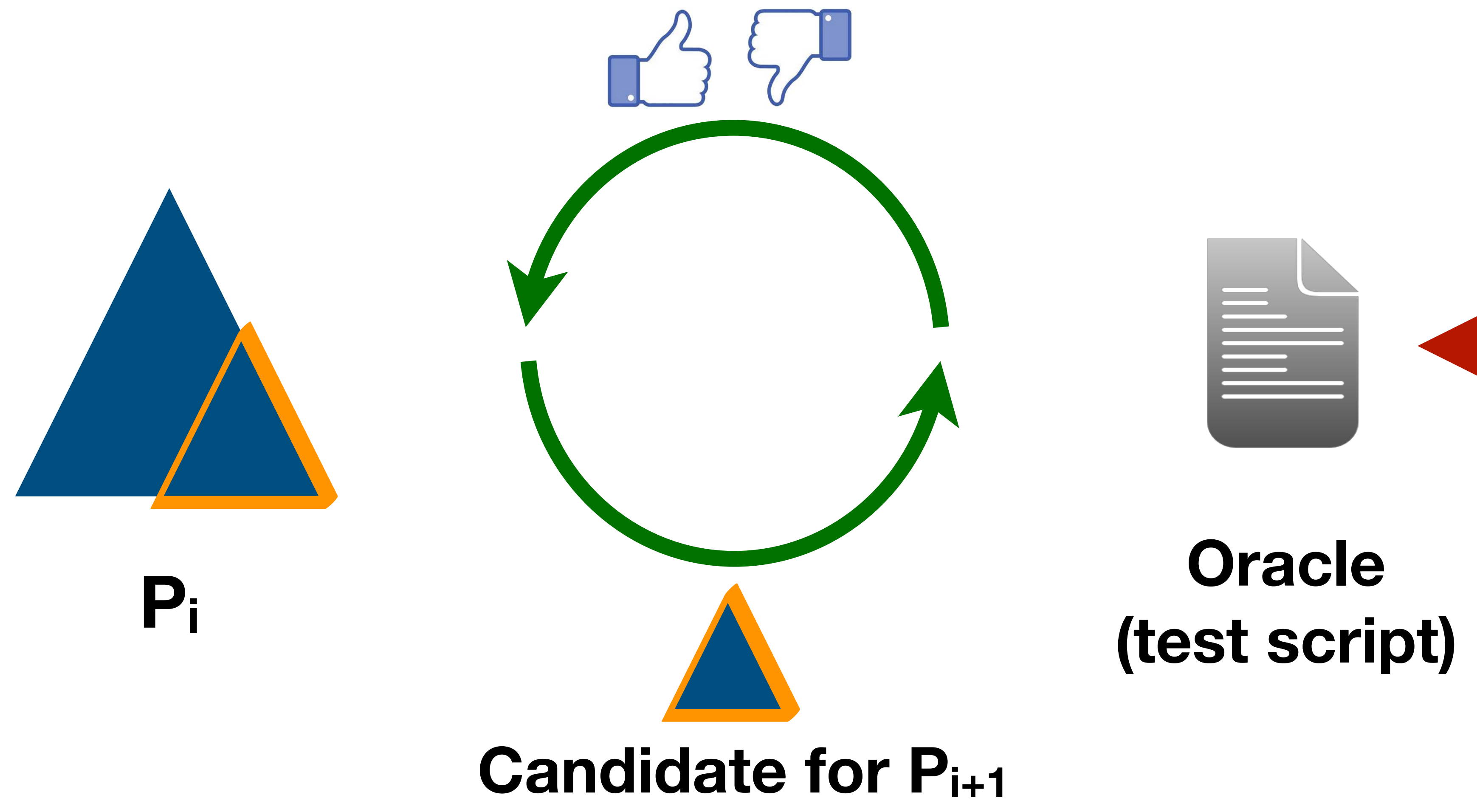
Delta Debugging Algorithm

$$ddmin(P) = ddmin_2(P, 2) \quad \text{where}$$

$$ddmin(P', n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\}.\ test(\Delta_i) = \mathsf{true} \ (\text{"reduce to subset"}) \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{if } \exists i \in \{1, \ldots, n\}.\ test(\nabla_i) = \mathsf{true} \ (\text{"reduce to complement"}) \\ ddmin_2(P', \min(|P'|, 2n)) & \text{if } n < |P'| \ (\text{"increase granularity"}) \\ P' & \text{otherwise } (\text{"done"}) \end{cases}$$

$$\text{where } test(P) = \mathsf{true}, test(\emptyset) = \mathsf{false}, \nabla_i = P' - \Delta_i, P' = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$$

Problems



Blind search through a large space

Nontrivial cost (e.g., compile, testing)

Many synatctically & semantically invalid programs

1. Can we rule out ill-formed programs?
2. Can we learn useful knowledge to guide the search from trial and error?

Grammar-based Delta Debugging

- Delta debugging w.r.t a given **grammar** (i.e., rule out ill-formed programs)
- Idea: **Hierarchically** perform delta debugging + **tree-reduction** rules

```
// A simple grammar
<program> ::= <func_def>*
<func_def> ::= <id> <block>
     <block> ::= <stmt>*
       <stmt> ::= <assignment>
                      | <if_stmt>
                      | <block>
    <if_stmt> ::= if <expr> <block> <block>
```

list: original DD

list: original DD

tree: tree-reduction

$$ddif(\texttt{if}\ E\ B_1\ B_2) = \begin{cases} B_1 & \text{if the replacement to } B_1 \text{ leads to success} \\ B_2 & \text{if the replacement to } B_2 \text{ leads to success} \\ \texttt{if}\ E\ B_1\ B_2 & \text{otherwise} \end{cases}$$

Example

- Property of interest: print a string including “Hello world!”

```
int f() { return 1; }
int main() {
  int a = f();
  if (a) {
    printf(“%d\n”, a);
    printf(“Hello ”);
    printf(“world!\n”);
    printf(“End\n”);
  }
  return 0;
}
```

```
int f() { return 1; }
int main() {
  int a = f();

    printf(“%d\n”, a);
    printf(“Hello ”);
    printf(“world!\n”);
    printf(“End\n”);

  return 0;
}
```

```
int f() { return 1; }
int main() {


    printf(“Hello ”);
    printf(“world!\n”);


  return 0;
}
```

```

int main() {


    printf(“Hello ”);
    printf(“world!\n”);


  return 0;
}
```

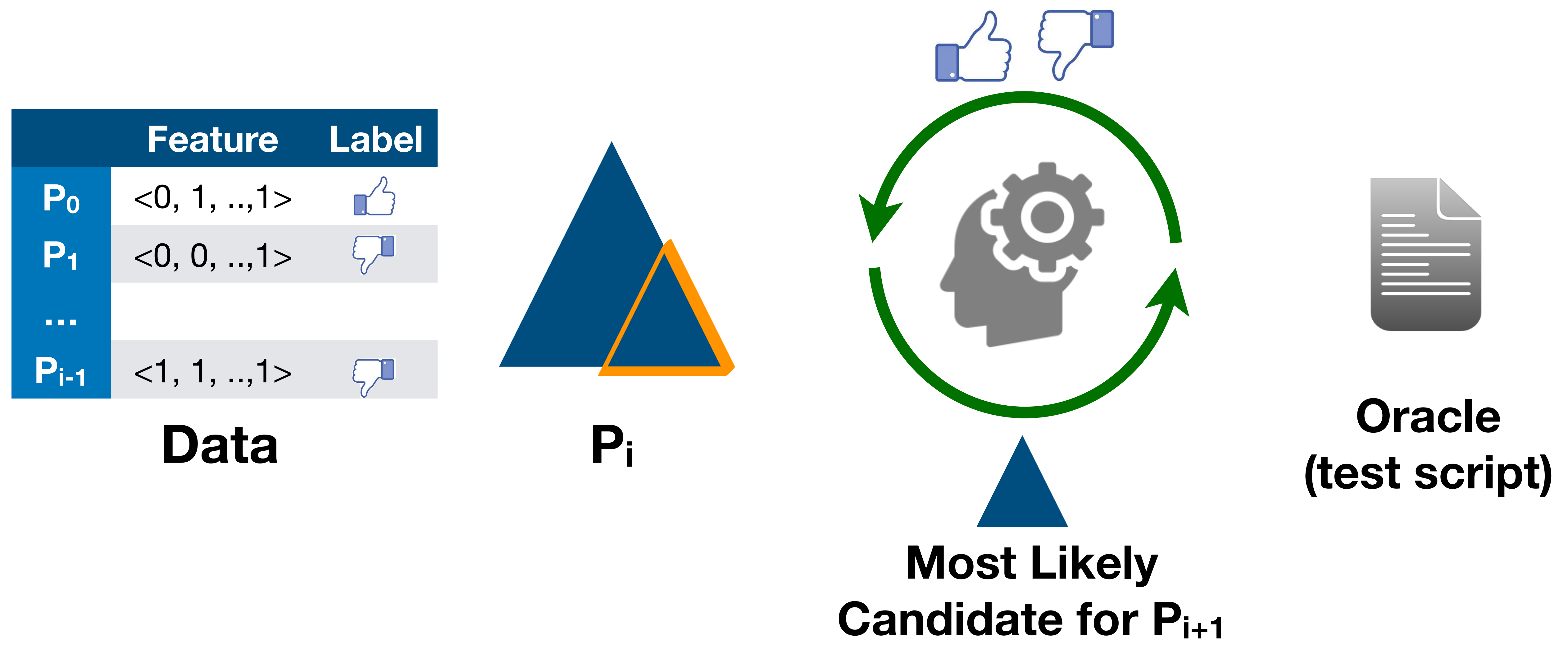
1. DD on the list of the functions
2. DD on the list of stmts of main
3. Reduction of the if-statement

4. DD on the list of stmts of the block

5. DD on the list of functions (again)
   (… until reaching a fixpoint)

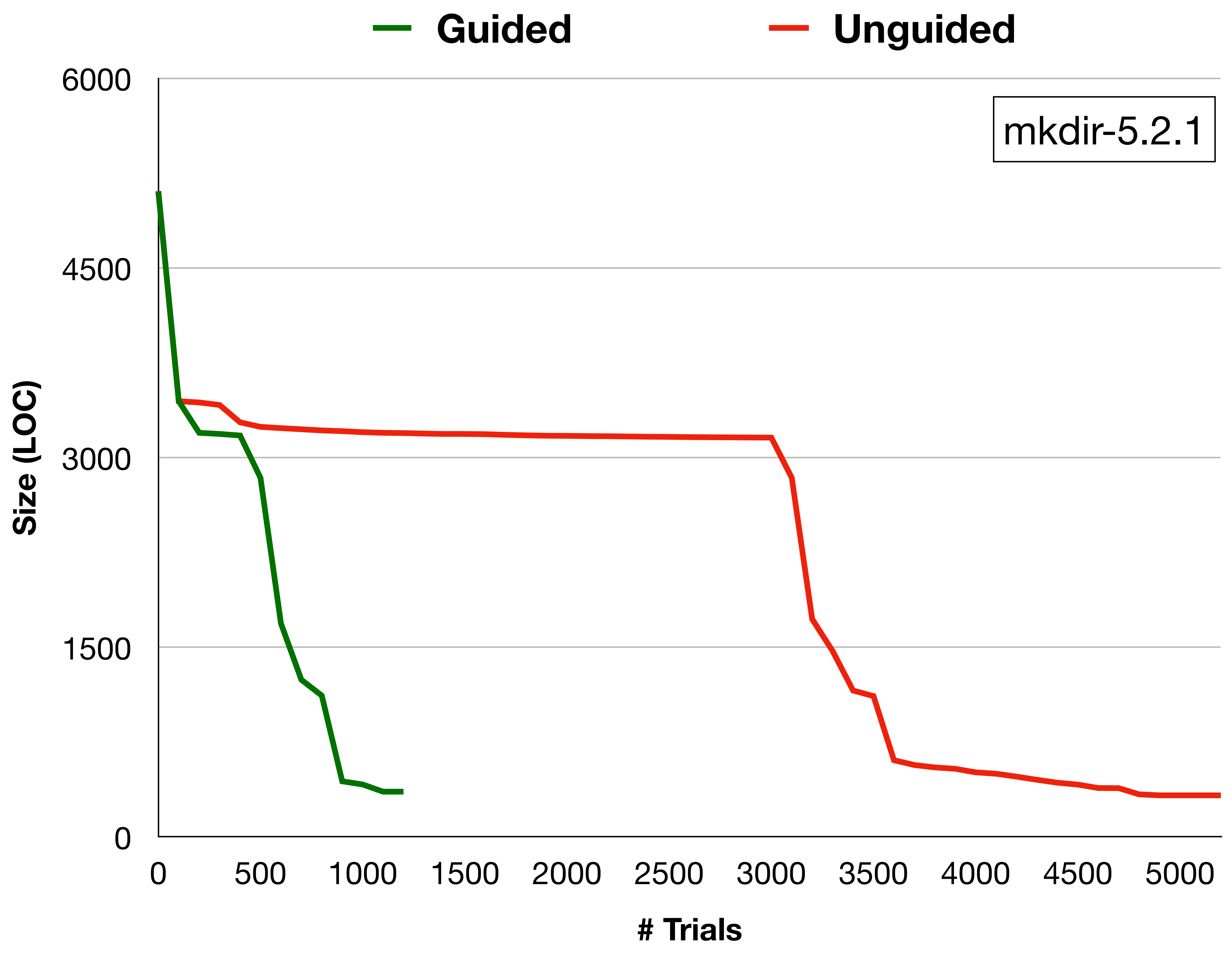
Learning-guided Delta Debugging

- **Learn a policy** for DD using reinforcement learning (RL)
- **Guide the search** based on the prediction of the learned policy
- Still guarantee **1-minimality** and **O(|*P*|²)** time complexity



	Feature	Label
P_0	<0, 1, ..,1>	👍
P_1	<0, 0, ..,1>	👎
…		
P_{i-1}	<1, 1, ..,1>	👎

Effectiveness



Example

```c
/* mkdir-5.2.1 */
int xstrtol(char *s, char **ptr, int strtol_base, strtol_t *val,
               char *valid_suffixes) {
    err = 0;
    assert(0 <= strtol_base && strtol_base <= 36);
    p = ptr ? ptr : &t_ptr;
    q = s;
    while(ISSPACE (*q)) ++q;
    if (*q == '-') return LONGINT_INVALID;
    errno = 0;
    tmp = strtol(s, p, strtol_base);
    if (*p == s) { … }
    if (!valid_suffixes) { … }
    if (**p != '\0') { … }
    *val = tmp;
    return err;
}
```

▨ : removed code

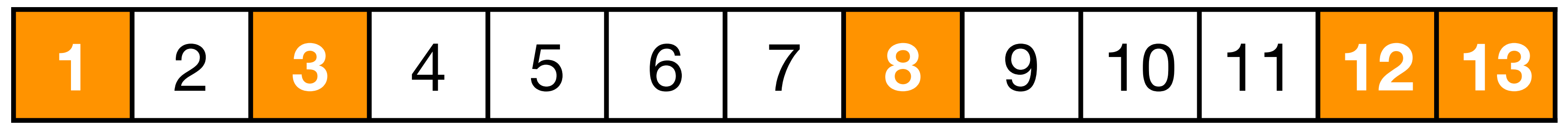
Example

```
/* mkdir-5.2.1 */
int xstrtol(char *s, char **ptr, int strtol_base, strtol_t *val,
            char *valid_suffixes) {
 1: err = 0;
 2: assert(0 <= strtol_base && strtol_base <= 36);
 3: p = ptr ? ptr : &t_ptr;
 4: q = s;
 5: while(ISSPACE (*q)) ++q;
 6: if (*q == '-') return LONGINT_INVALID;
 7: errno = 0;
 8: tmp = strtol(s, p, strtol_base);
 9: if (*p == s) { … }
10: if (!valid_suffixes) { … }
11: if (**p != '\0') { … }
12: *val = tmp;
13: return err;
}
```

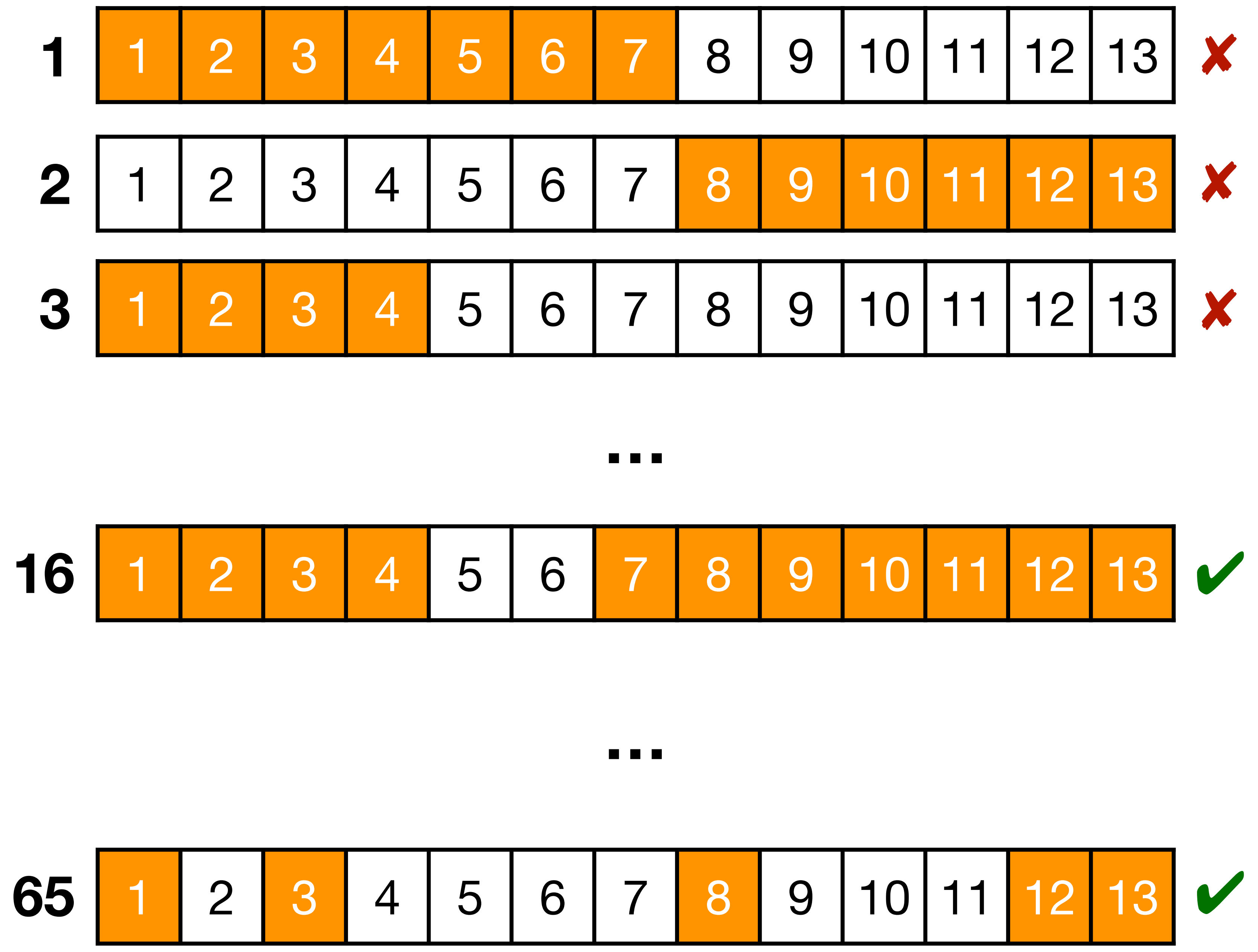
▨ : removed code

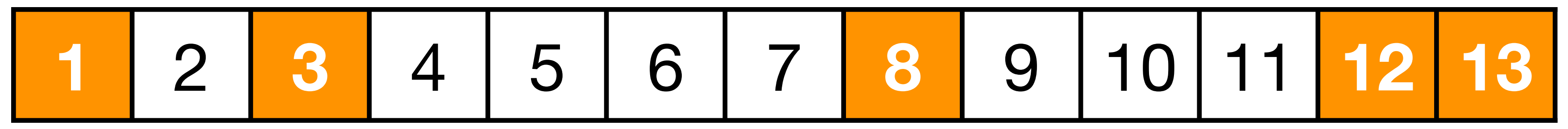
Minimal Desired Program:

1	2	**3**	4	5	6	7	**8**	9	10	11	**12**	**13**

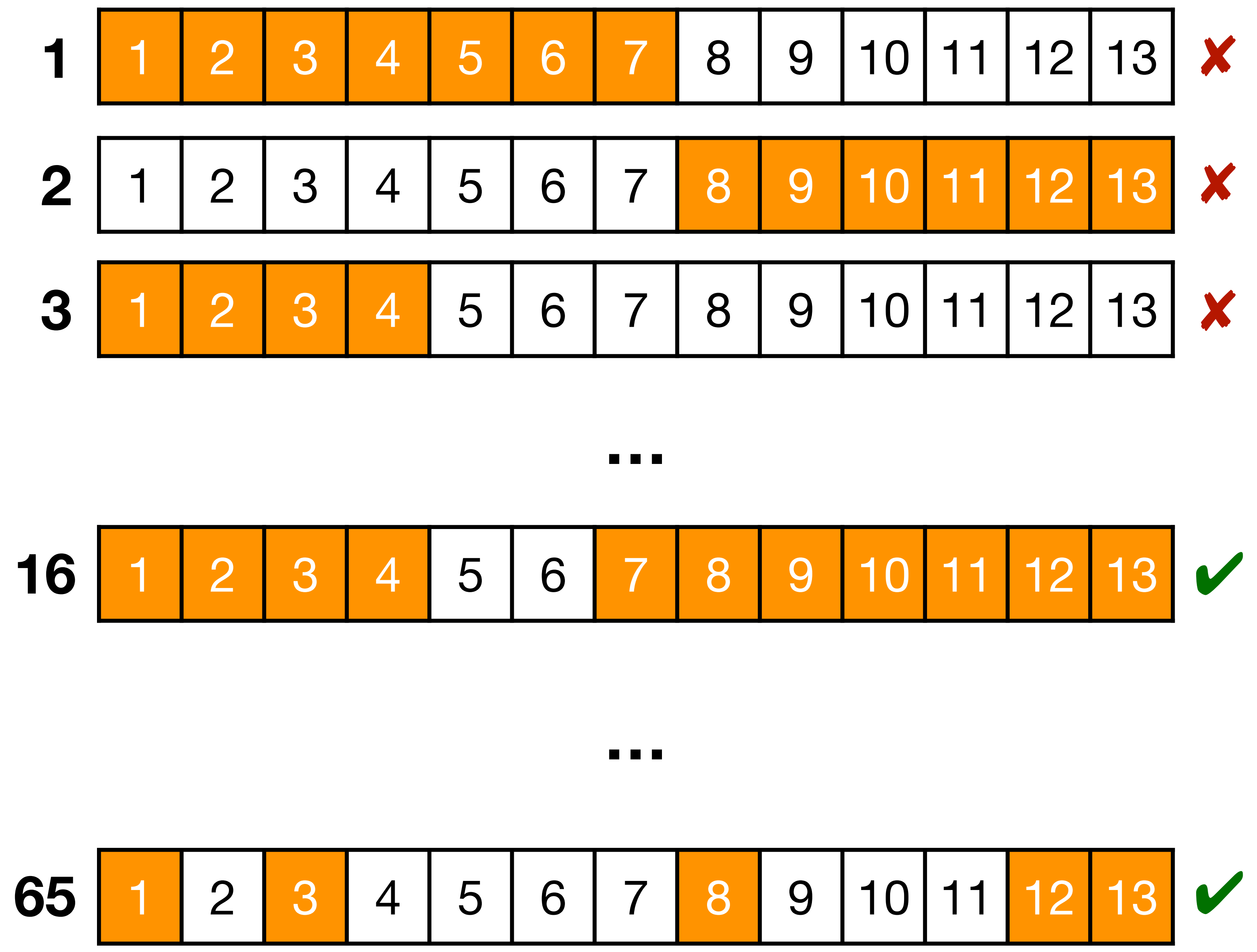


Unguided Delta-Debugging

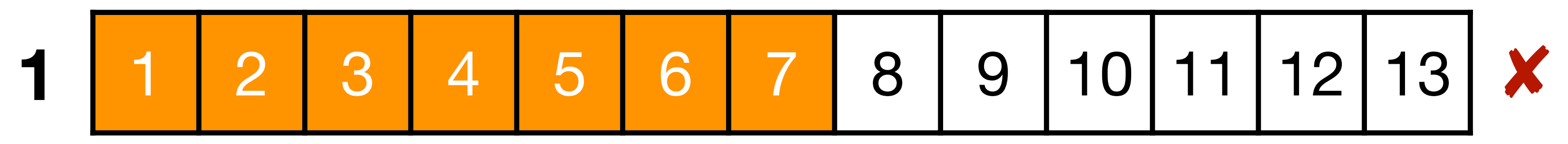


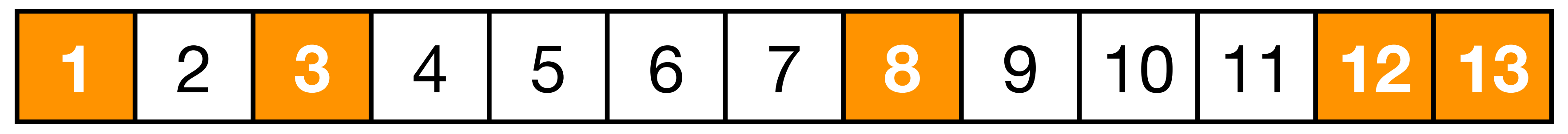


Unguided Delta-Debugging

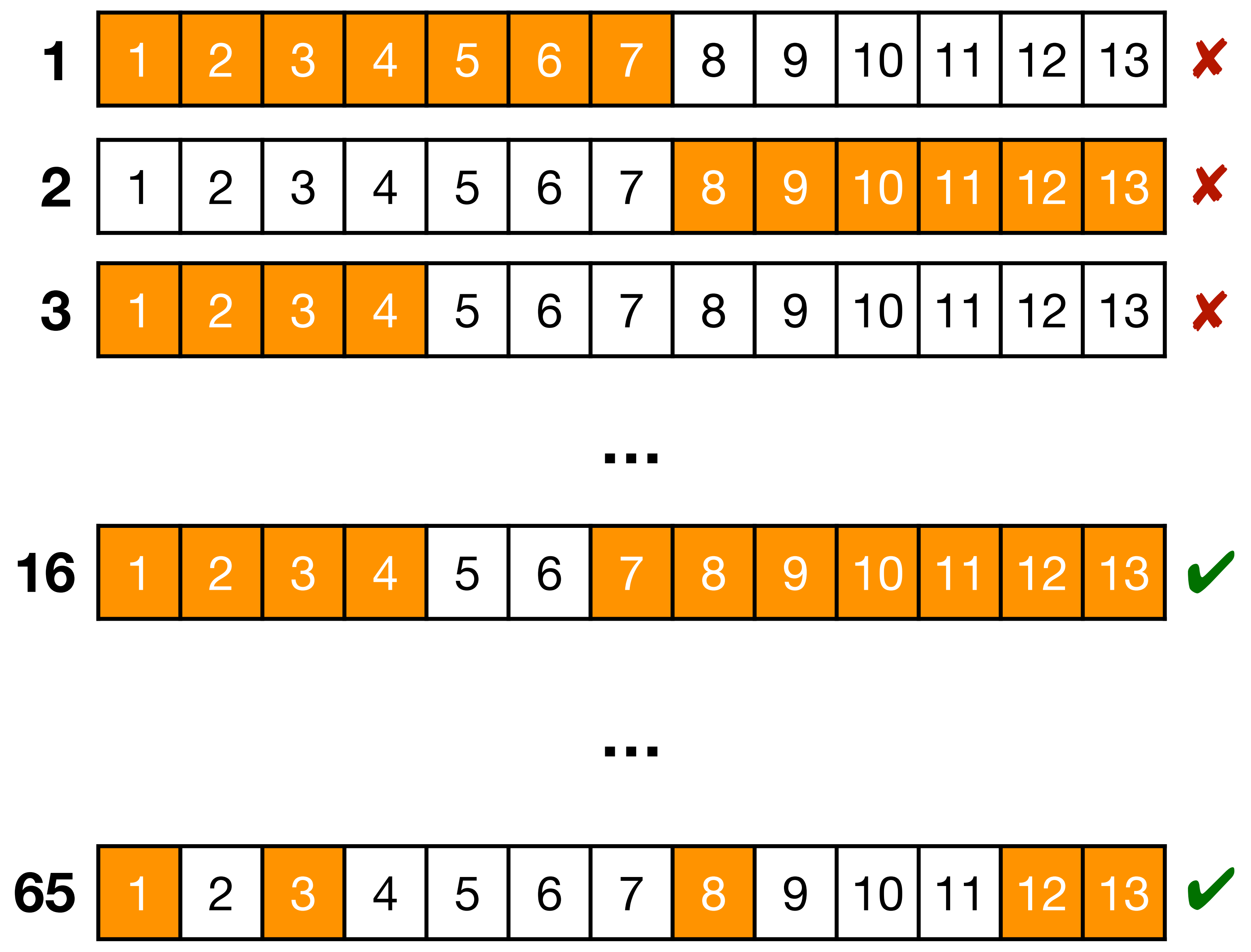


Guided Delta-Debugging

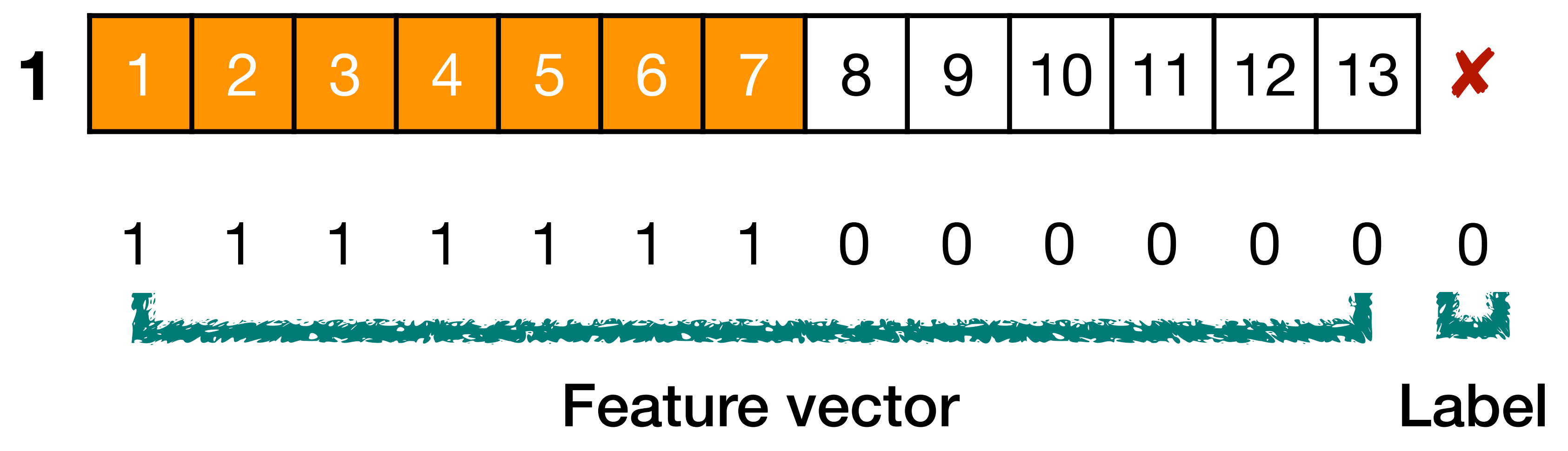


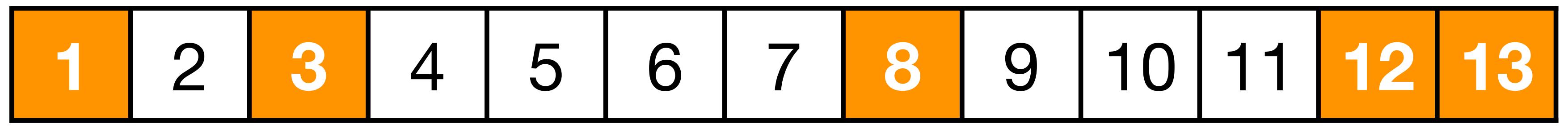


Unguided Delta-Debugging

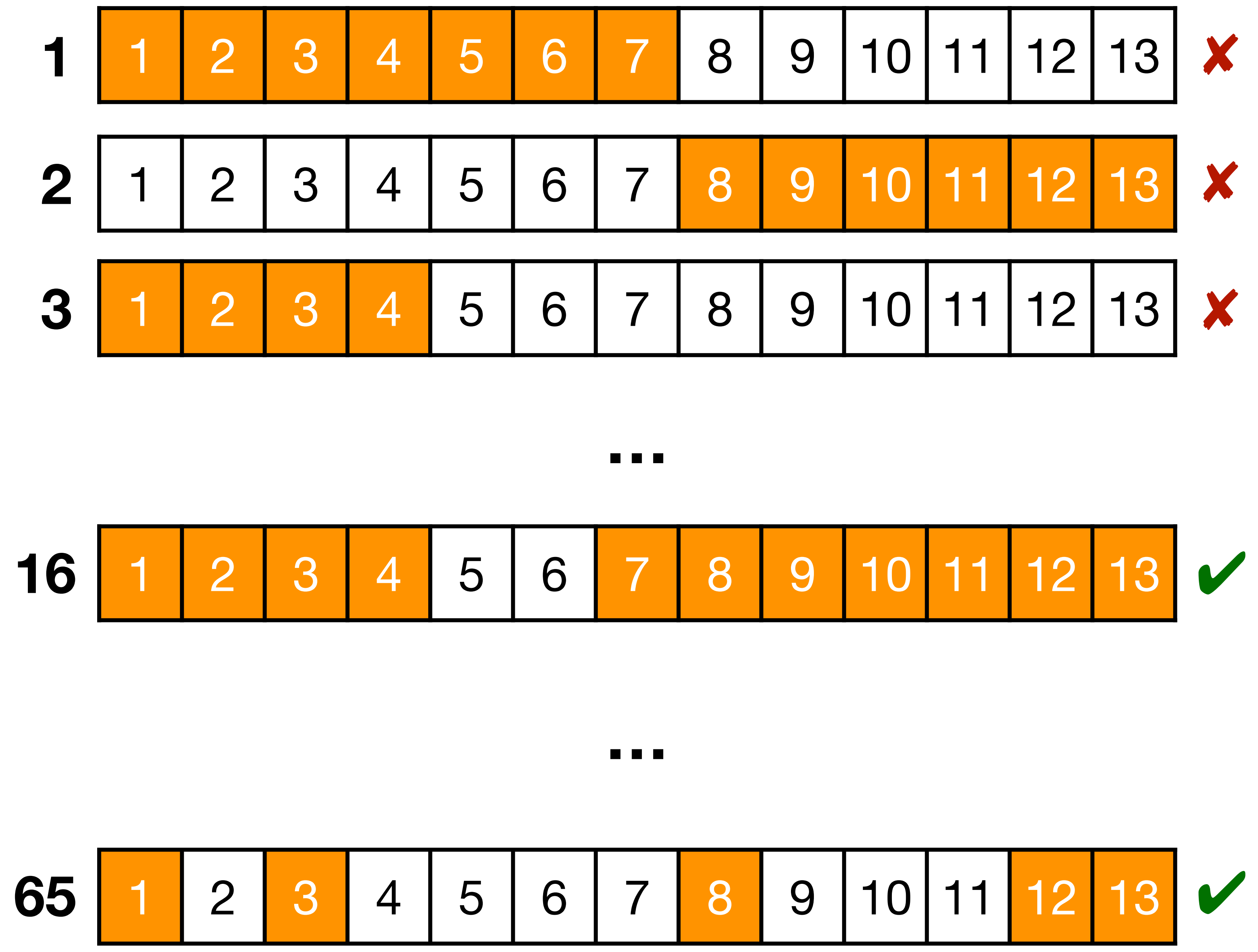


Guided Delta-Debugging

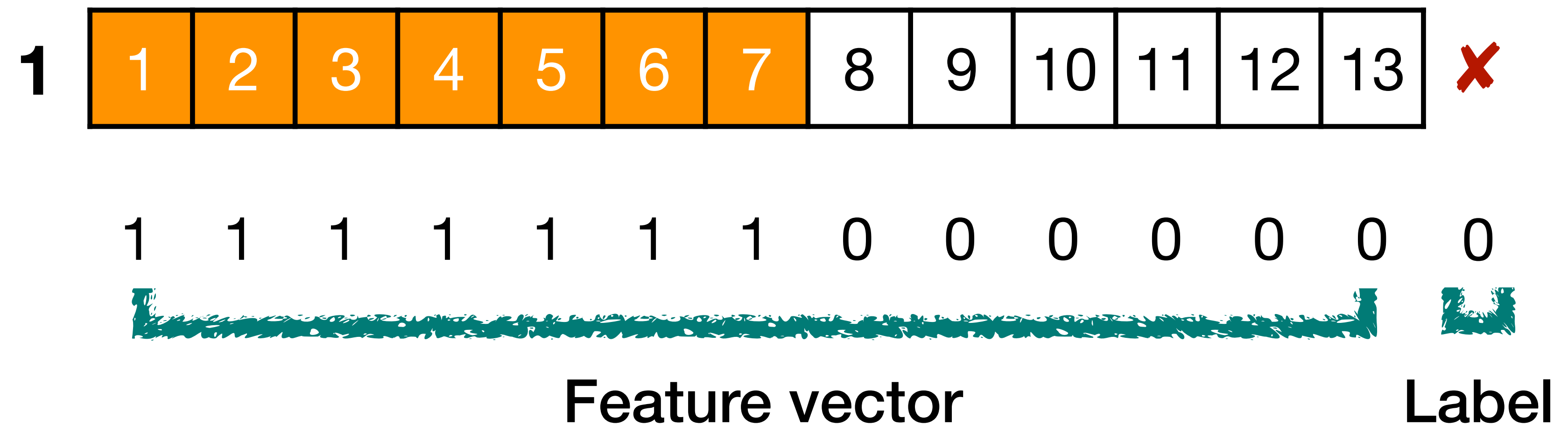


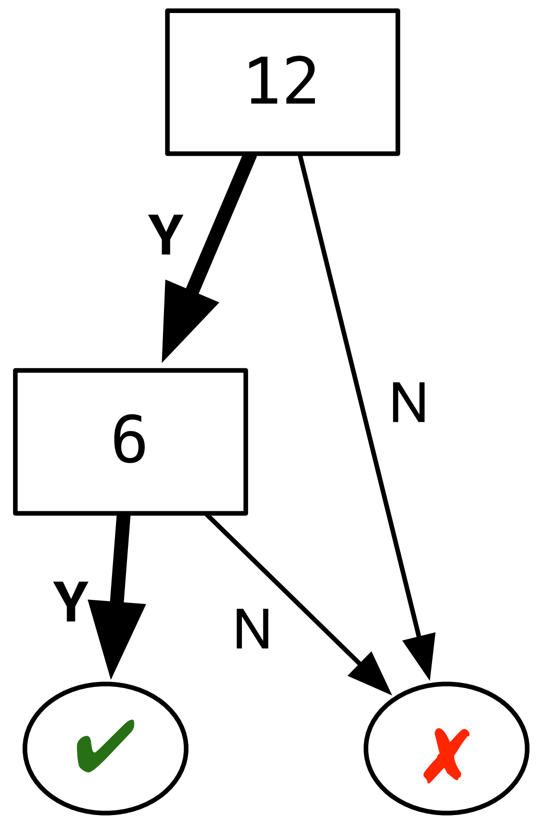


Unguided Delta-Debugging

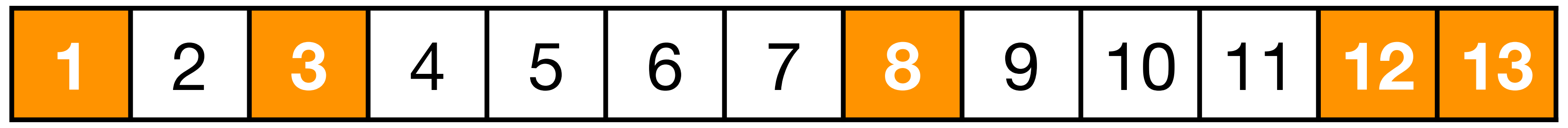


Guided Delta-Debugging

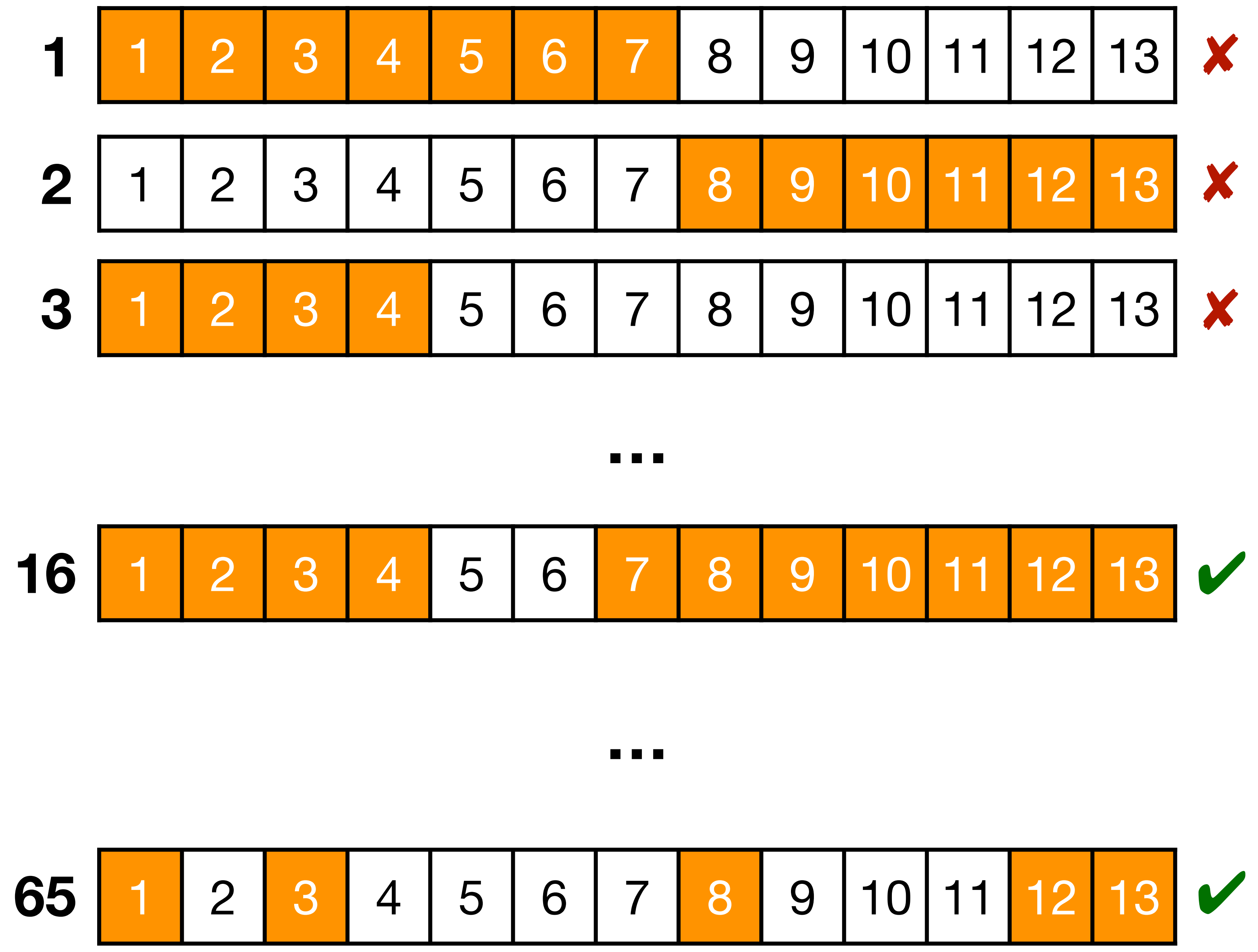




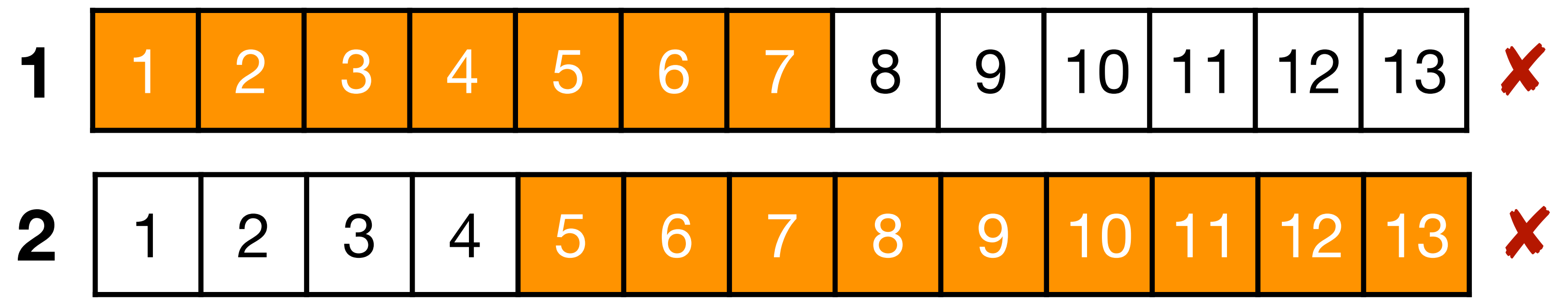
*P** should include 6 and 12

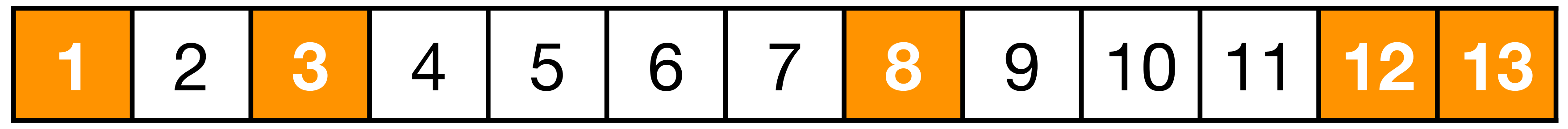


Unguided Delta-Debugging

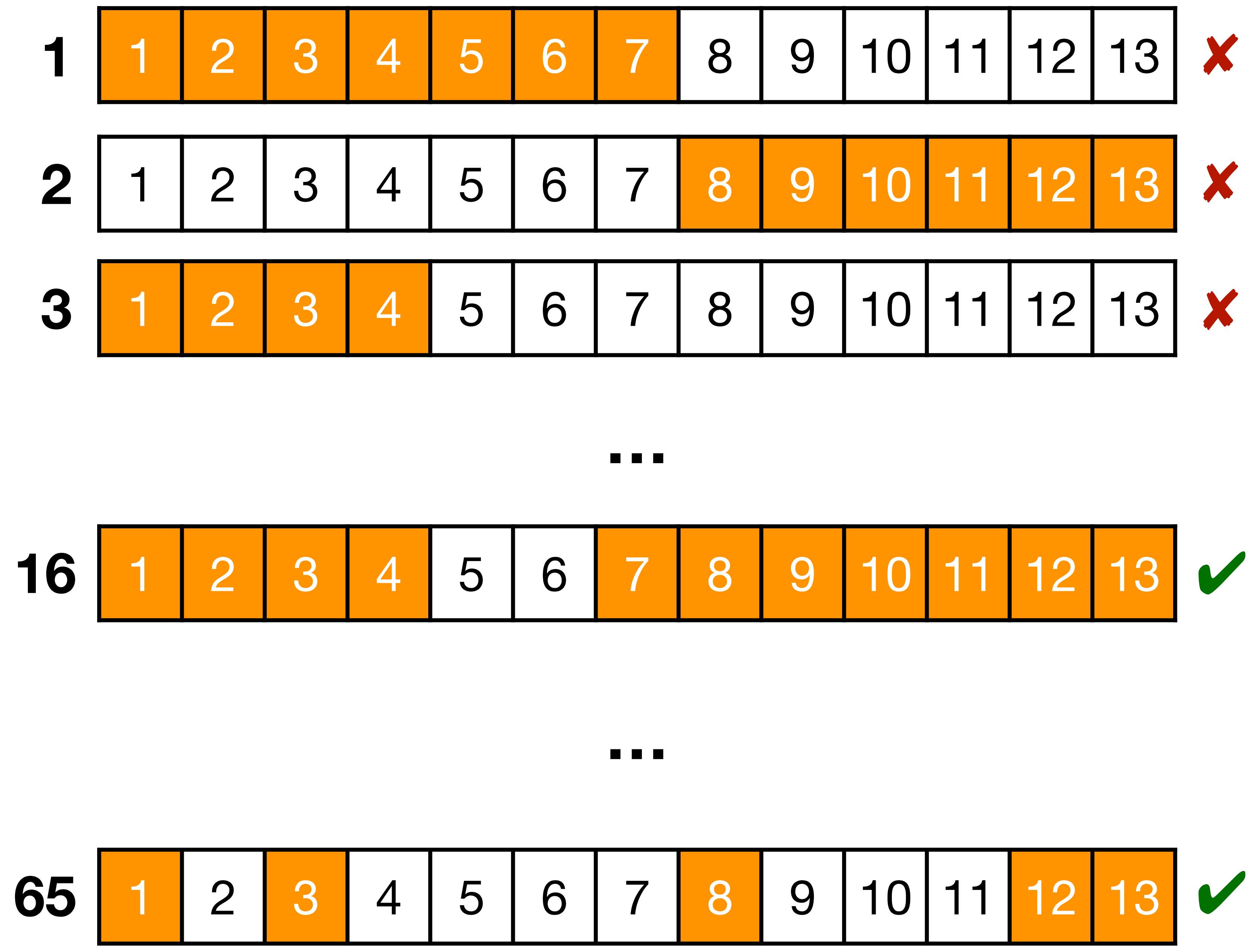


Guided Delta-Debugging

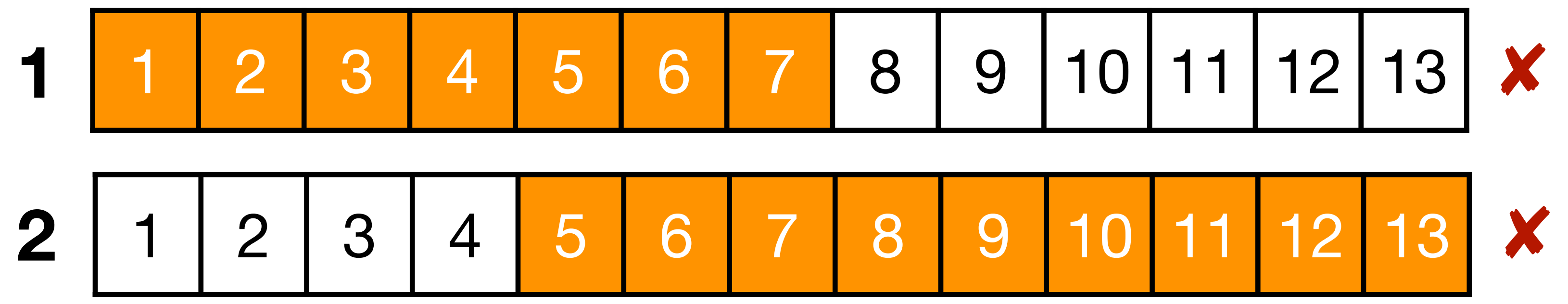


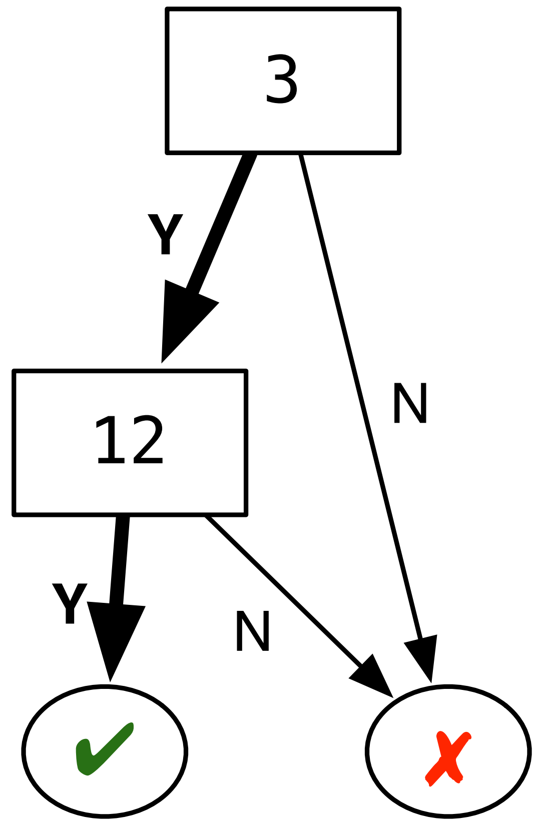


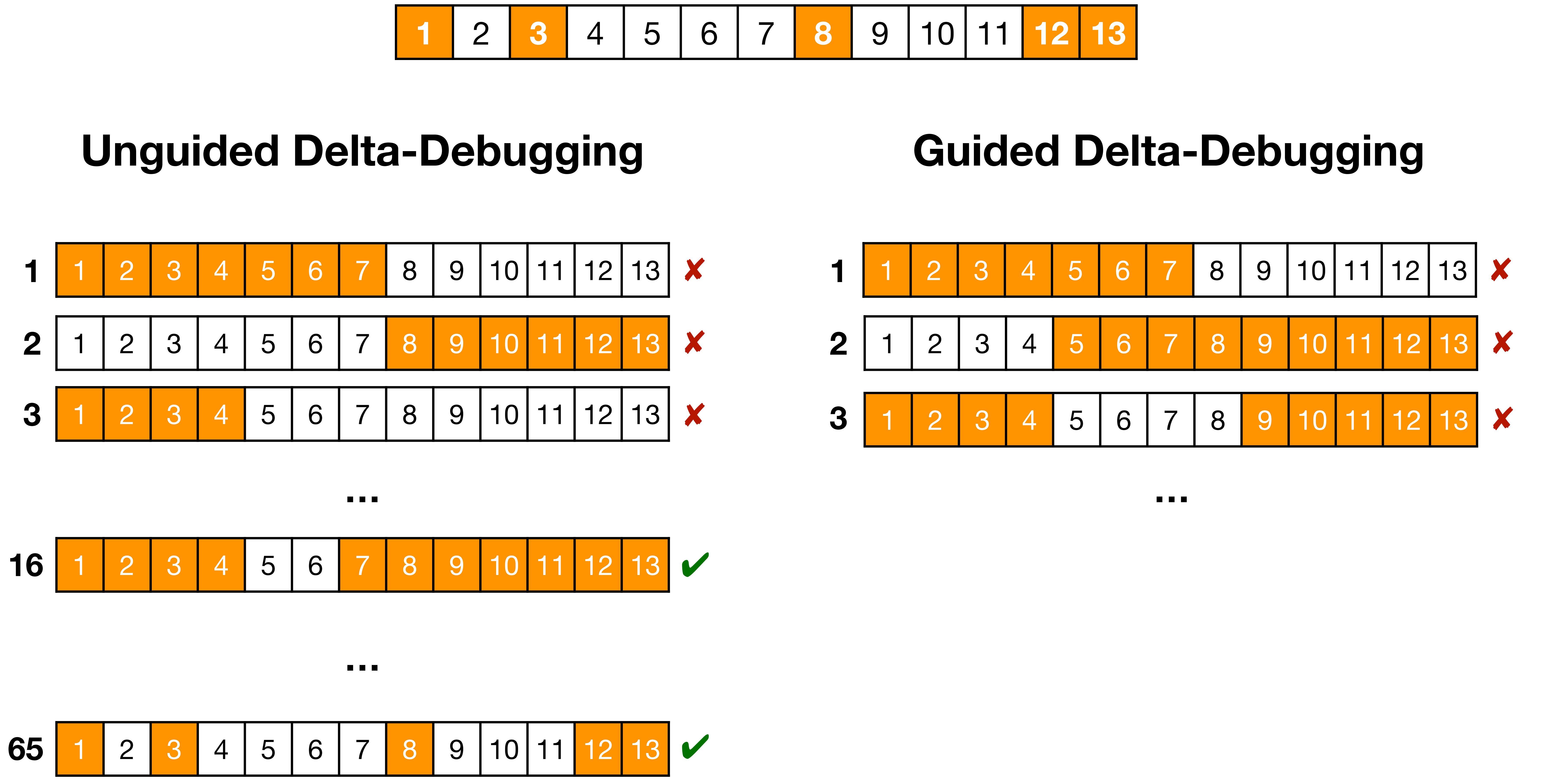
Unguided Delta-Debugging

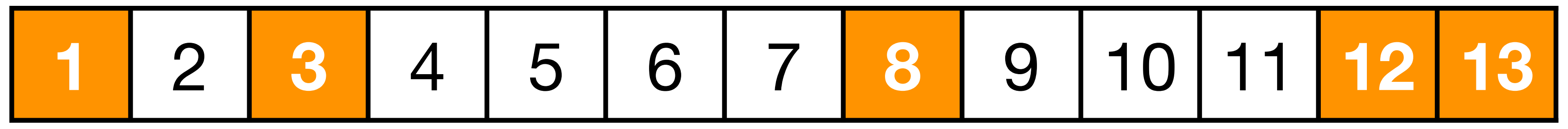


Guided Delta-Debugging

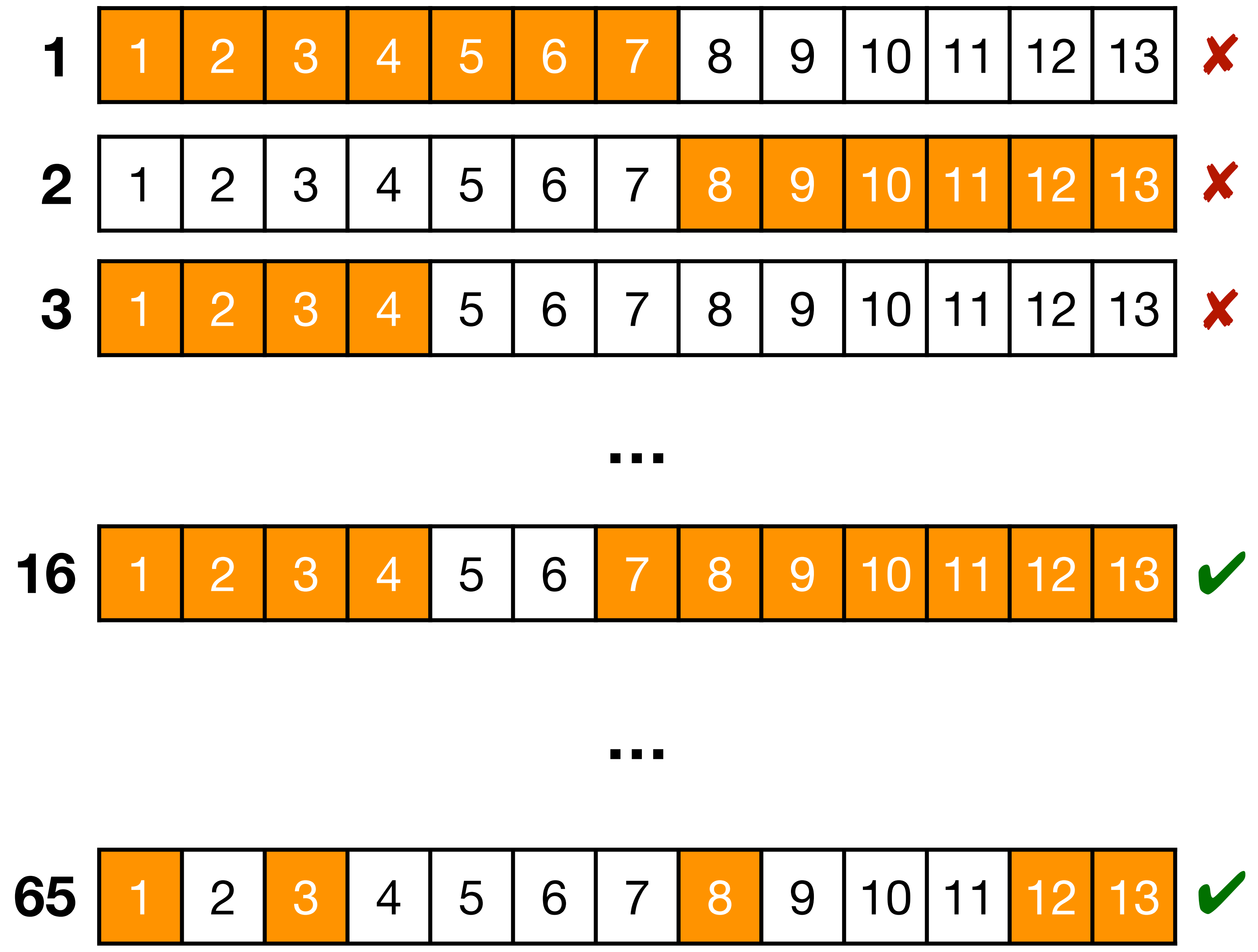




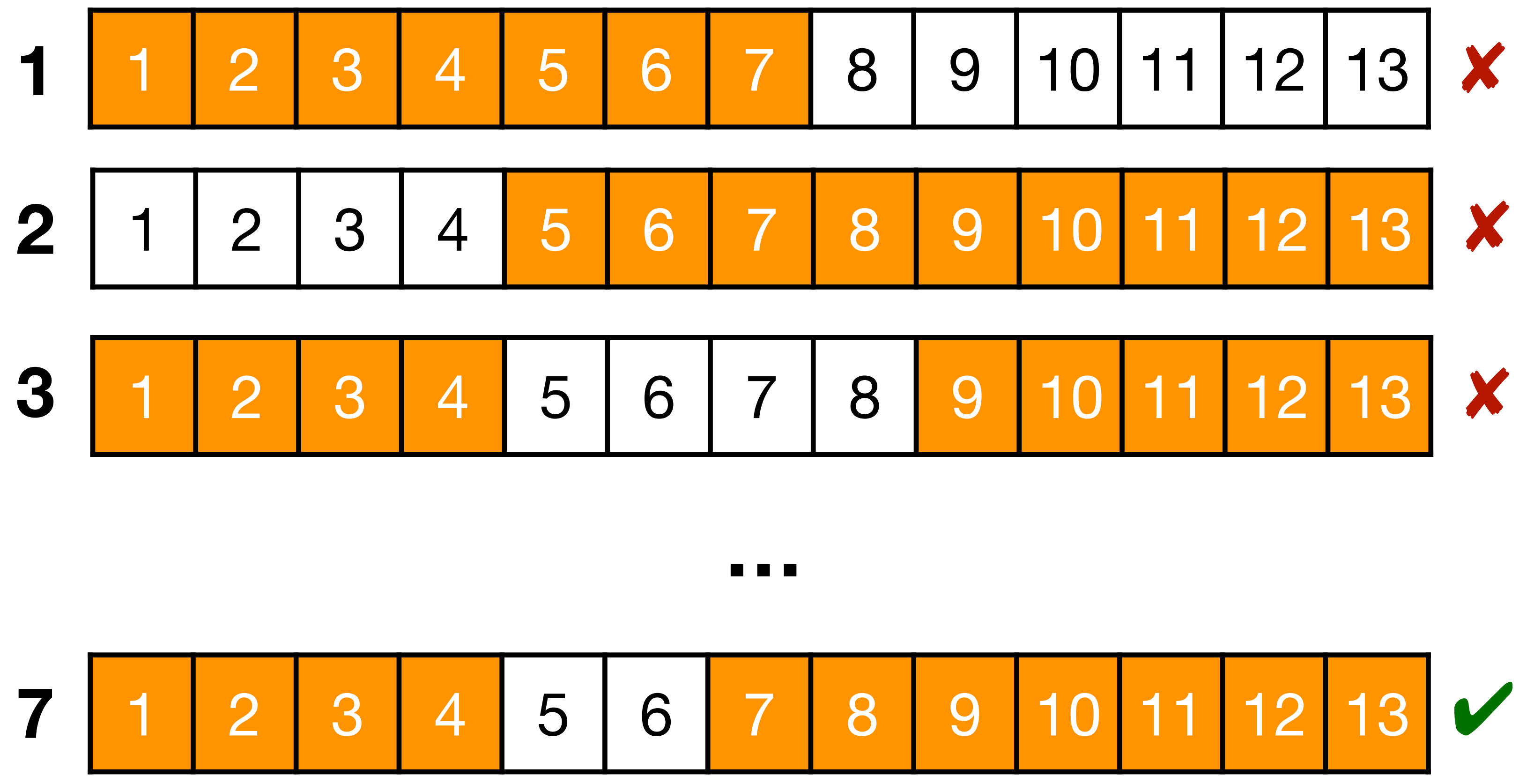

1 2 3 4 5 6 7 8 9 10 11 12 13
Unguided Delta-Debugging
1 ✗
2 ✗
3 ✗
...
16 ✓
...
65 ✓
Guided Delta-Debugging
1 ✗
2 ✗
3 ✗
...

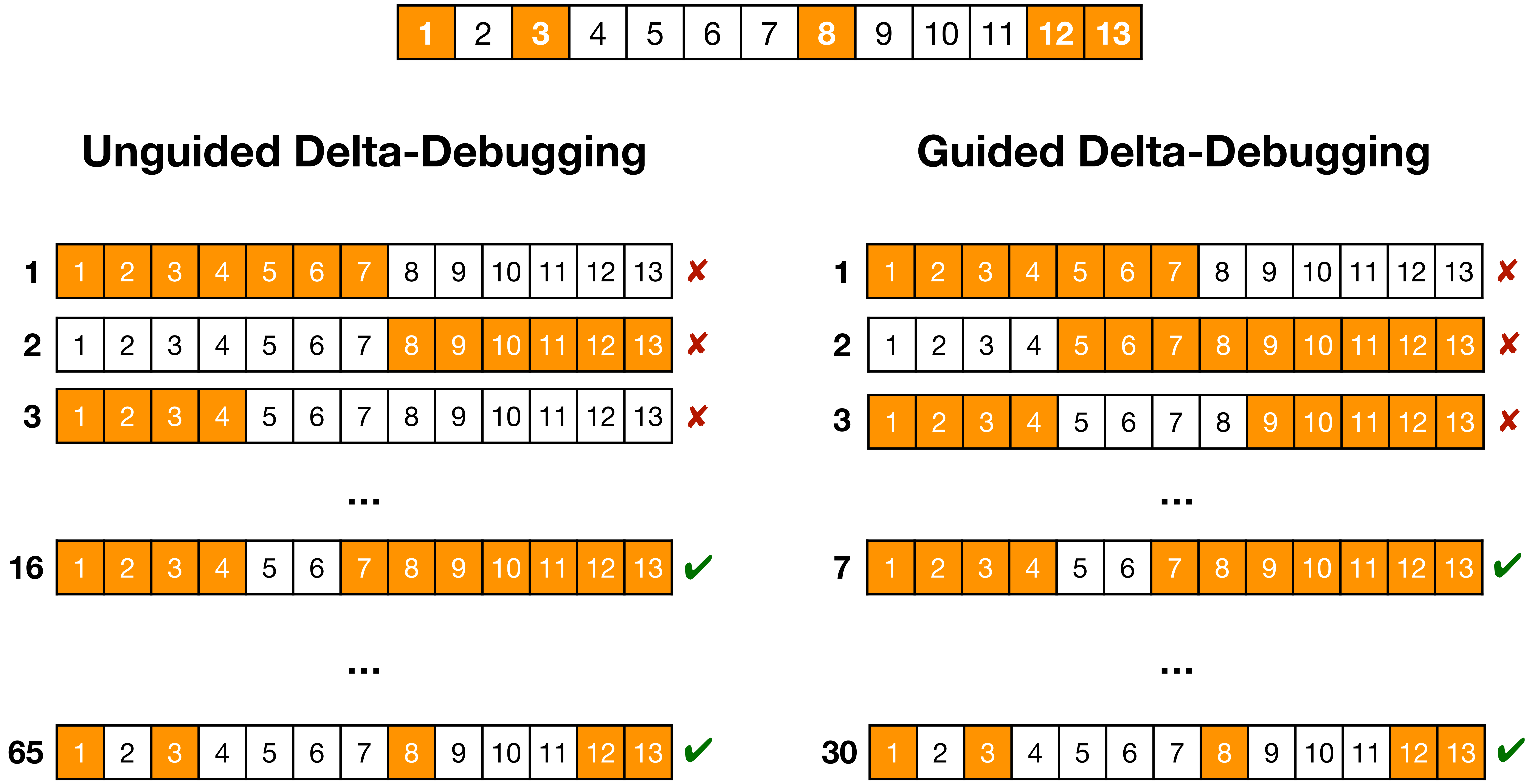


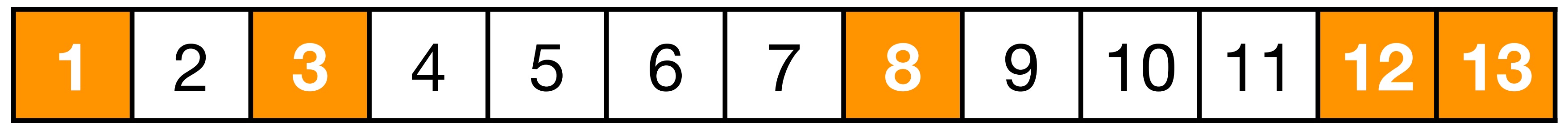
Unguided Delta-Debugging



Guided Delta-Debugging

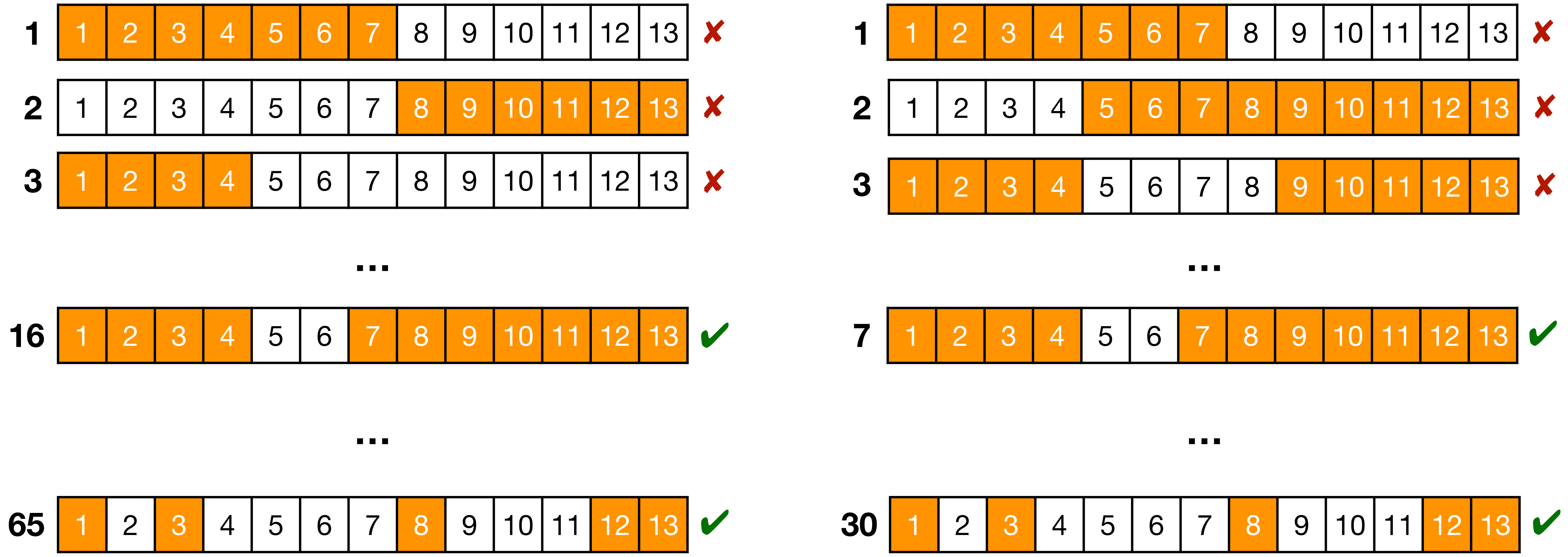



Unguided Delta-Debugging
Guided Delta-Debugging
1
2
3
...
16
...
65
1
2
3
...
7
...
30



Unguided Delta-Debugging

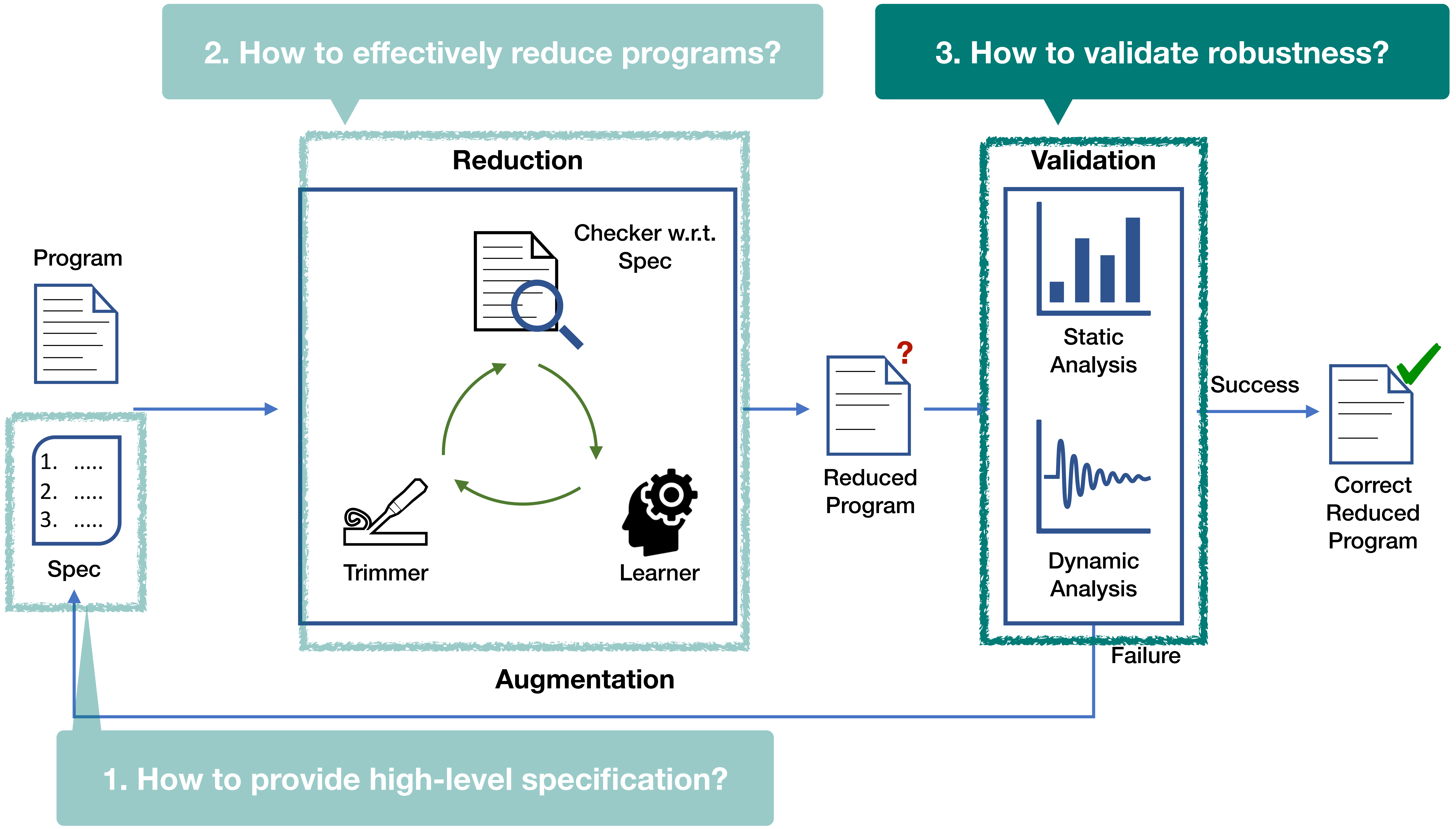
Guided Delta-Debugging



5,169 trials (4,872 failures)

1,174 trials (901 failures)

Key Questions



Validation

- Check the **robustness** of the reduced program
  - preventing newly introduced security holes
- Sound static buffer overflow analyzer (Sparrow)
  - #alarms in tar: 1,290 → 19 (feasible for manual inspection)
- Random fuzzer (AFL)
  - no crashing input found in 3 days for tar

Augmentation

- Augment the test script with crashing inputs by AFL
- Typically converges in up to 3 iterations in practice
- But, may be incomplete

```
/* grep-2.19 */
void add_tok (token t) {
    /* removed in the first trial and restored after augmentation */
    if (dfa->talloc == dfa->tindex)
        dfa->tokens = (token *) realloc (/* large size */);
    *(dfa->tokens + (dfa->tindex++)) = t;
}
```

Experimental Setup

- 10 widely used **UNIX utility programs** (13—90 KLOC)
    - each program has a **known CVE**
    - **remove unreachable code** by static analysis upfront
- Specification:
    - supporting **the same cmd line options** as BusyBox
    - with the **test suites** by the original developers

Size of Reduced Program

#Statement

Program	Original	Chisel	Hand-written
bzip-1.05	6,316	1,575	2,342
chown-8.2	3,422	186	141
date-8.21	4,100	913	107
grep-2.19	10,816	1,071	355
gzip-1.2.4	4,069	1,042	1,058
mkdir-5.2.1	1,746	142	94
rm-8.4	[illegible]	[illegible]	[illegible]
sort-8.16	[illegible]	[illegible]	[illegible]
tar-1.14	[illegible]	[illegible]	[illegible]
uniq-8.16	1,923	192	51
Total	55,848	6,111	4,729

Reachable code by static analysis

Chisel reduced 89%

Comparable to hand-written versions

Security Hardening

Remove 4 and 2 CVEs in undesired and desired functionalities.
4 CVEs are not easily fixable by reduction (e.g., race condition).

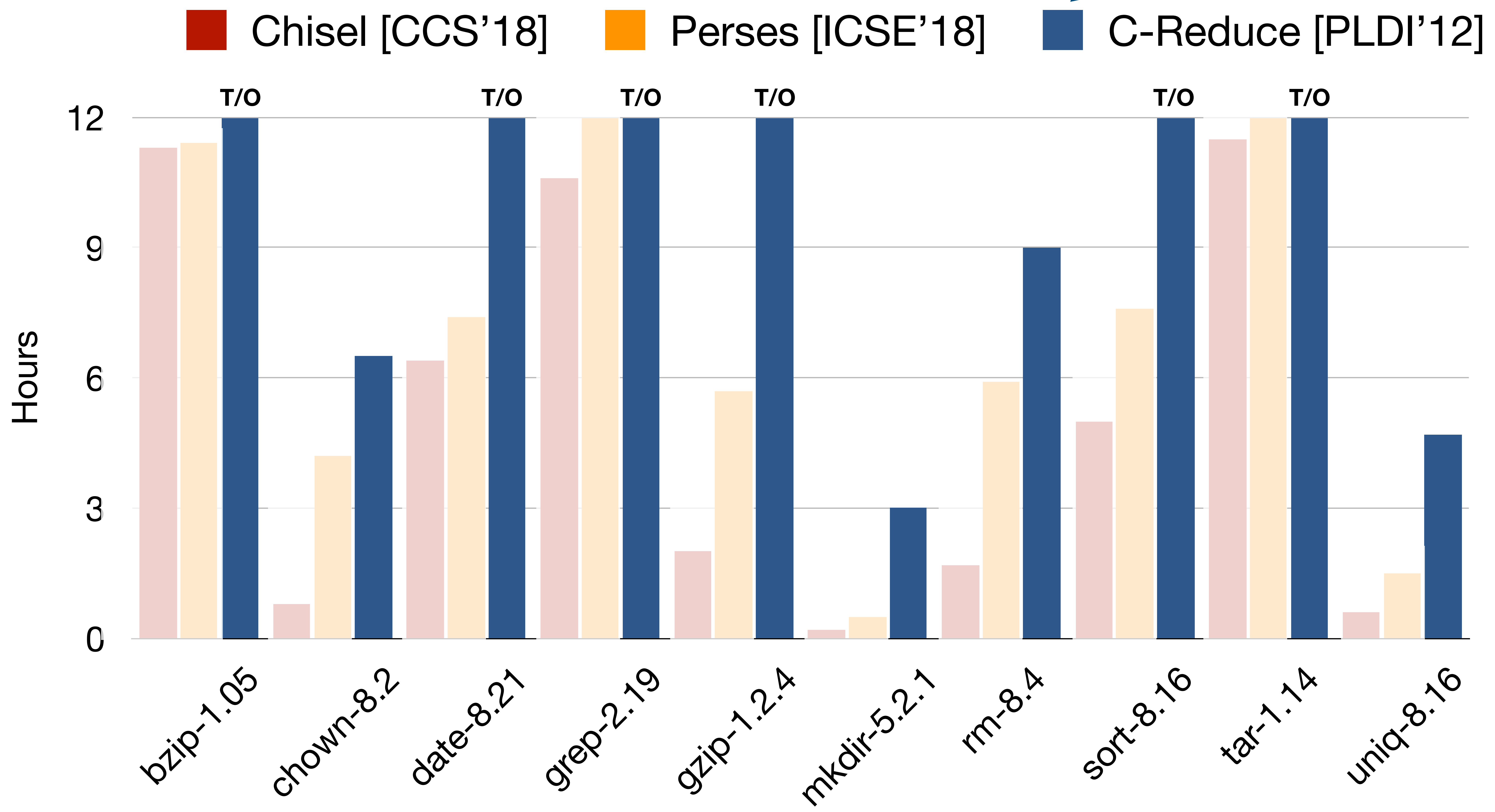
		#ROP Gadgets			#Alarms		
Program	CVE	Original	Reduced		Original	Reduced	
bzip-1.05	✗	662	298	(55%)	1,991	33	(98%)
chown-8.2	✓	534	162	(70%)	47	1	(98%)
date-8.21	✓	479	233	(51%)	201	23	(89%)
grep-2.19	✓	1,065	411	(61%)	619	31	(95%)
gzip-1.2.4	✓	456	340	(25%)	326	128	(61%)
mkdir-5.2.1	✗	229	124	(46%)	43	2	(95%)
rm-8.4	✗	565	95	(83%)	48	0	(100%)
sort-8.16	✓						
tar-1.14	✓						
uniq-8.16	✗	349	[illegible]	(69%)	60	1	(98%)
Total		6,752	2,285	(66%)	5,298	243	(95%)

Reduced potential attack surface

Make it feasible for manual alarm inspection

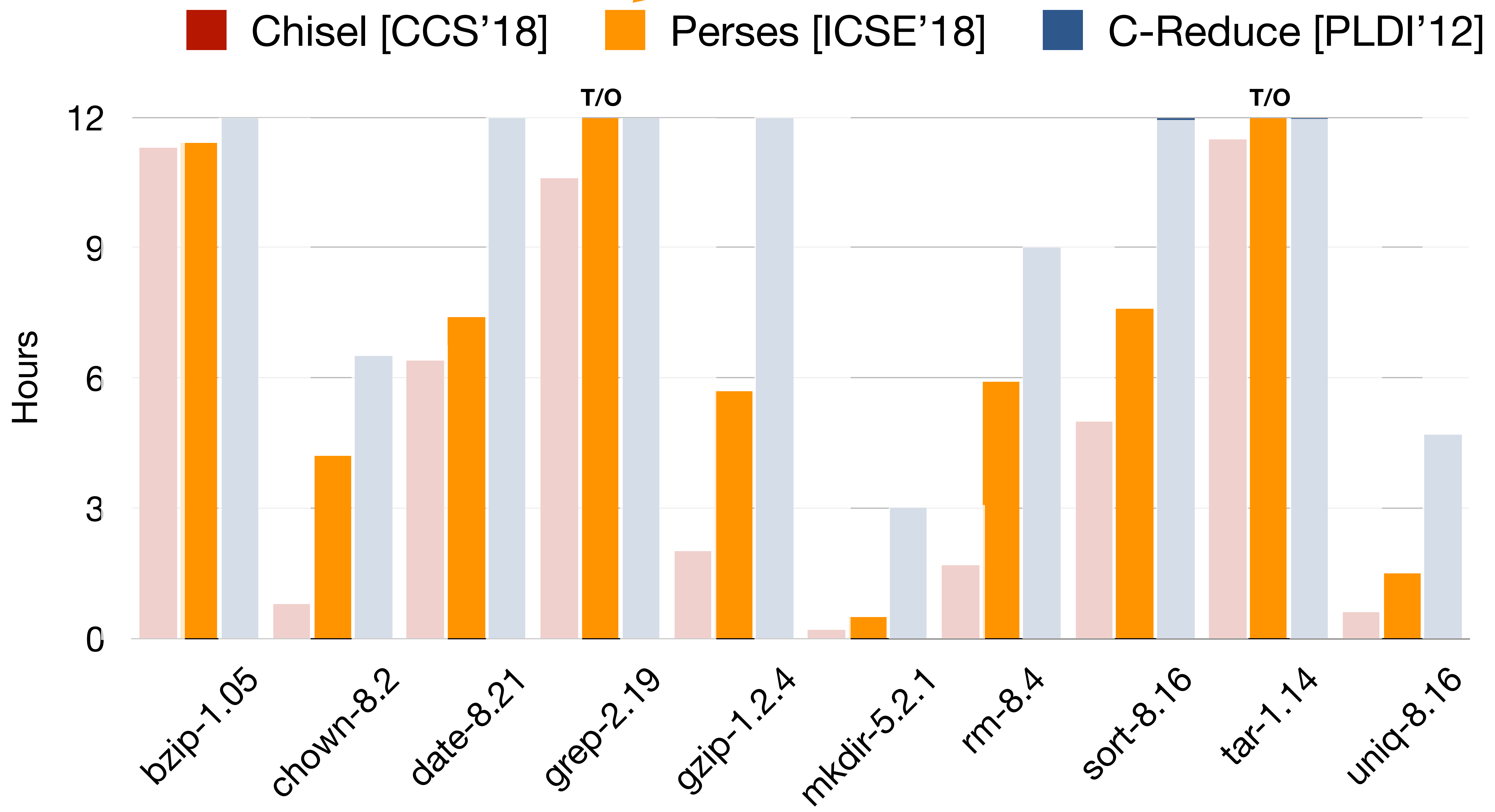
Reduction Time

Line-based reducer ran out of time for 6 programs



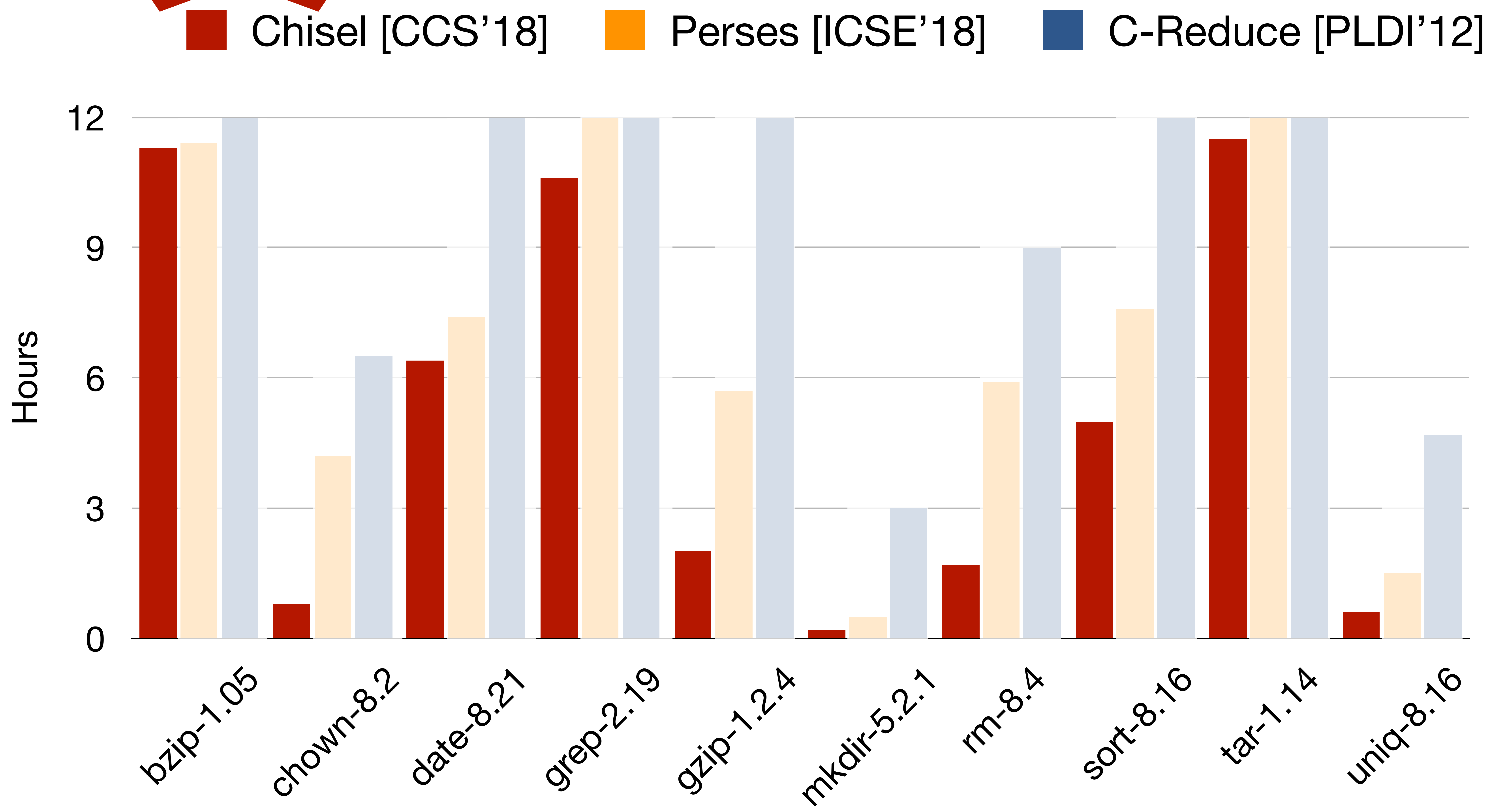
Reduction Time

Grammar-based reducer ran out of time for 2 programs



Reduction Time

7x and 4x faster than
C-Reduce and Perses



Summary

- Program debloating: **simplifying and hardening** large & complex SW
- Chisel: automated software debloating system
  - **tractable search** via learning-guided delta debugging
  - **security hardening** by removing undesired features
  - **robustness** via static & dynamic analyses
- Need a lot more research on efficiency and effectiveness
  - E.g., advanced learning techniques, system-level debloating (inter-program)