Let’s help symbolic execution SOAR!

Tomasz Kuchta

Samsung R&D Institute Poland

Agenda

- Symbex & others: the state of the art
- Docovery, Shadow & AoT: selective and incremental symbex
- SOAR: in search of the secret sauce
- Academia & Industry: perspectives matter
- Future outlook for symbex

Agenda

- **Symbex & others: the state of the art**
- Docovery, Shadow & AoT: selective and incremental symbex
- SOAR: in search of the secret sauce
- Academia & Industry: perspectives matter
- Future outlook for symbex

Symbolic execution: how did we get here?

- First proposed in mid-70's
- Really took off in 2000's with the advancement of SMT solvers
- Applied for: bug finding, analysis, security, equivalence checking, input recovery, patch testing, etc.
- Many flavors: DSE, concolic execution, hybrid approaches with fuzzing

Symbolic execution: how do we stand?

- Success stories: testing Microsoft Office (SAGE), success of symbex-based tools at DARPA Cyber Grand Challenge (Mayhem, Driller)
- Well established tools: KLEE, Symcc, Symbolic PathFinder, Angr
- Symbex offers great features: no False Positives (FPs) and a *thorough* reasoning about explored execution paths
- Yes, but -> still used more as a boutique approach rather than first choice

Symbex vs others: static analysis

- Static analysis has been widely used in Industry
- Often a project needs to pass Klocwork / Coverity for sign-off
- OSS tools: Clang Static Analyzer, Meta Infer, Ericsson CodeChecker

✓ scalability, ease of use ✗ produces (mostly*) false positives

* More fine-tuning -> fewer FPs

Runs, I need more runs!

Neo

Symbex vs others: fuzzing

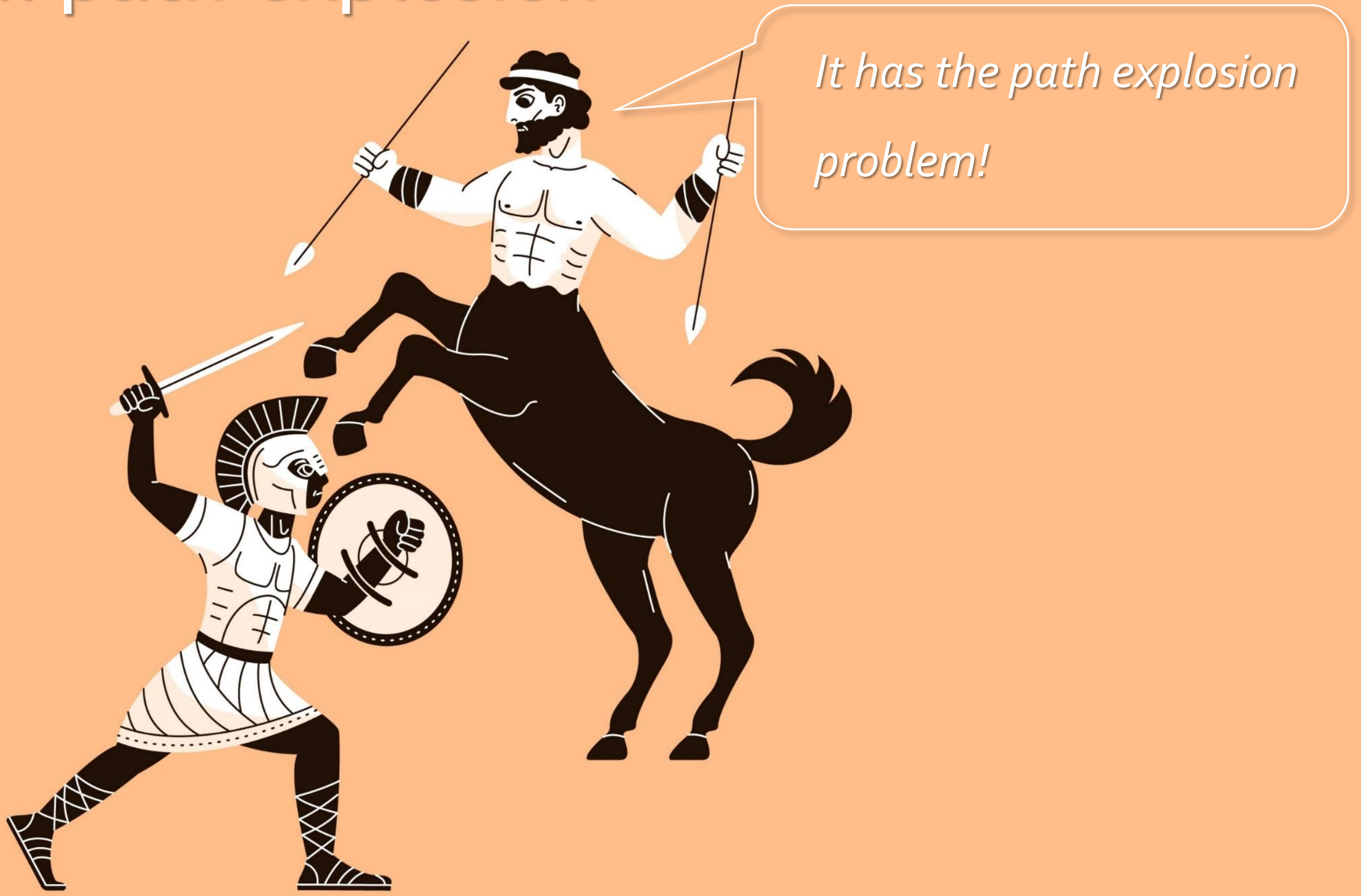
- Fuzzing: current de-facto standard
- Original paper from 1990 but the technique really took off with AFL
- Widely used for bug finding and security testing in particular
- Seems like everyone knows about / heard of fuzzing
- Variety of OSS tools, e.g. AFL++, syzkaller, libfuzzer

✓ scalability, ease of use ✗ lack of reasoning power

The mythical path explosion problem

Why don't you use symbex?!

The mythical path explosion problem



Path explosion refers to the fact that the number of control-flow paths in a program grows exponentially ("explodes") with an increase in program size and can even be infinite in the case of programs with unbounded loop iterations.

Wikipedia

The mythical path explosion problem

- Is it _really_ an issue with symbex then?
- Path explosion happens not because we use symbex
- Software is just that complex and that's the fundamental problem

What is the secret sauce then? What makes a technique widely used?

- Ease of deployment / quick learning curve
- Scalability
- Customization for purpose
- Engineering: lots of small tweaks, bits and pieces that add up
- Bottom line: if we cannot change the fundamental limitation, we should find ways around it – *there is no spoon and there is no secret sauce*

Agenda

- Symbex & others: the state of the art
- **Docovery, Shadow & AoT: selective and incremental symbex**
- SOAR: in search of the secret sauce
- Academia & Industry: perspectives matter
- Future outlook for symbex

Give a man a fish and you feed him for a day; teach a man to do program analysis and you feed him for a lifetime.

Author Unknown

Use this one simple symbex trick to …

- Let's go through 3 projects in which we applied certain "tweaks" to adapt symbex for a certain purpose and help it scale
- Docovery: limiting the search space via selective symbex
- Shadow: targeting only the behavior modified by a patch
- AoT: limiting the search space via target extraction, enabling symbex on difficult targets

Example #1: Docovery

Cristian Cadar, Miguel Castro and Manuel Costa

"Docovery: Toward Generic Automatic Document Recovery"
ASE'14

Challenge

- Broken inputs crash programs, users cannot access the contents
- Reason: corrupt data, buggy programs
- Also: input parsing accounts for a lot of security vulnerabilities

Possible solutions

- Try to fix the program
- Try to protect the program
- Try to fix the document
- ?

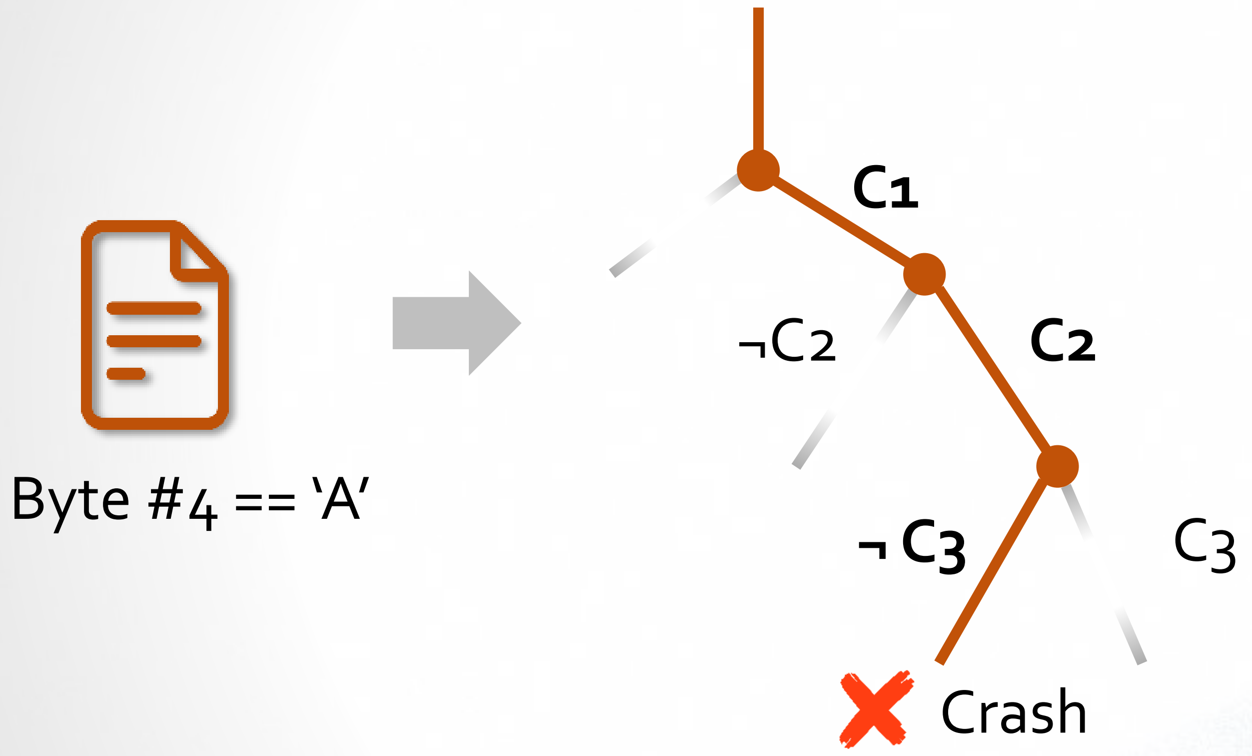
Motivation

Is it possible to fix a broken document, without assuming any input format, in a way that preserves the original contents as much as possible?

Docovery: the idea

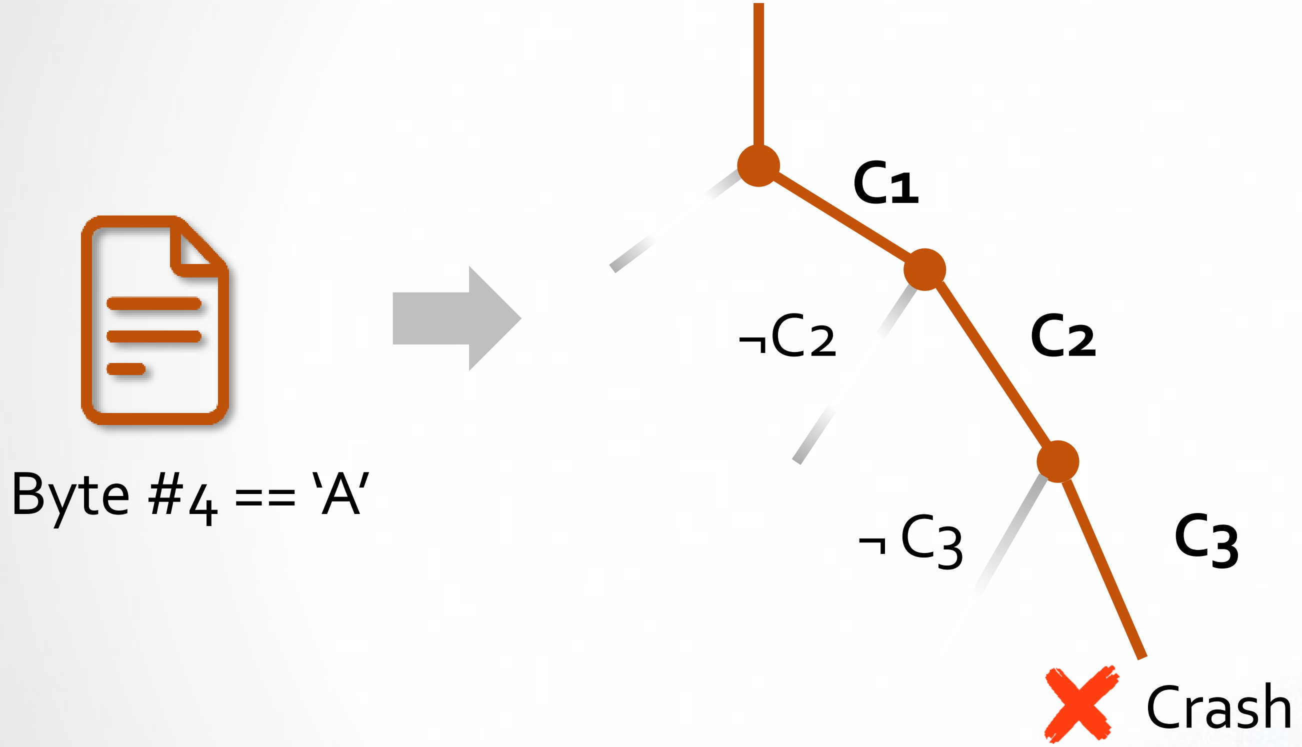
- Leverage the fact that a program knows how to parse its input
- Follow an execution path of a crashing input
- Try to diverge
- Generate a modified input for the alternative path

Follow an execution path of a crashing input



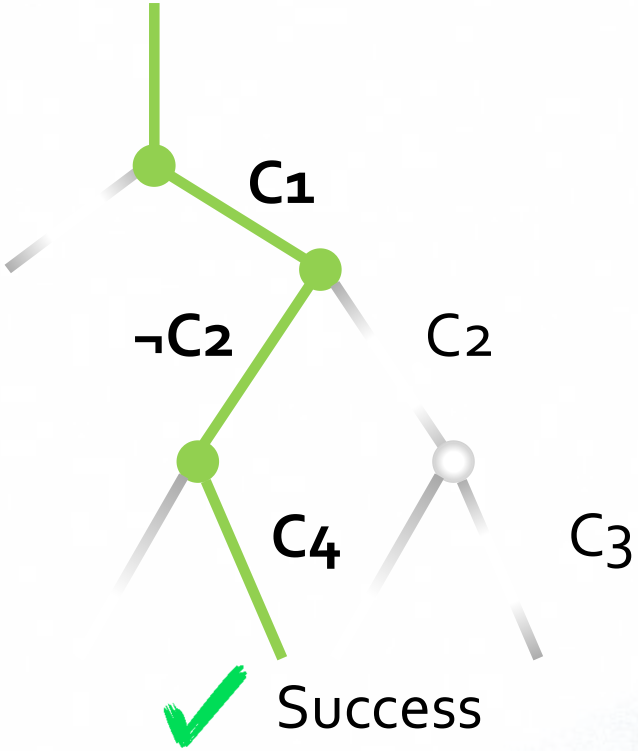
$C_1, \ldots, C_N$: constraints

Try to diverge



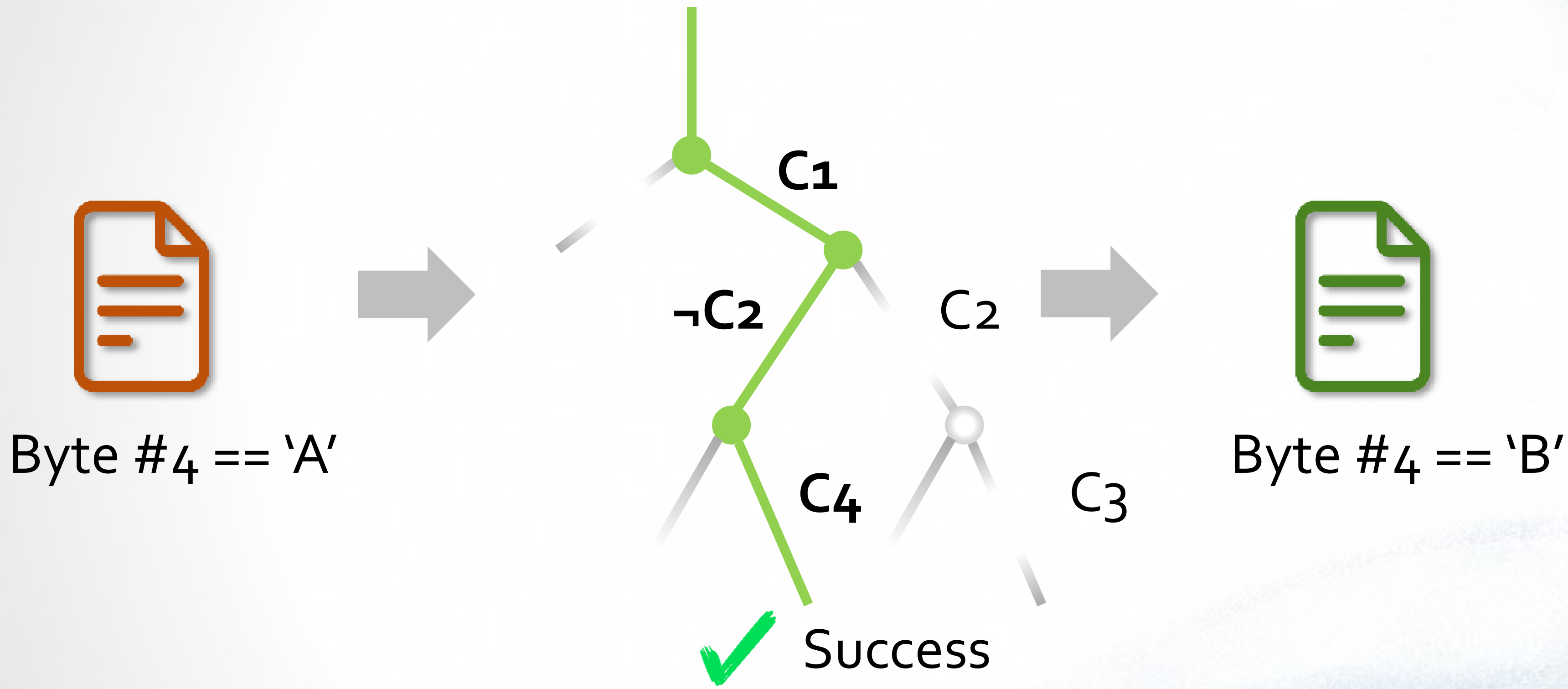
$C_1, \ldots, C_N$: constraints

Try to diverge



$C_1, \ldots, C_N$: constraints

Generate a modified input for the new path



$C_1, \ldots, C_N$: constraints

Challenges

- We learnt that we cannot mark *entire* input as symbolic
- Example: `Pine`, a command line e-mail client

```
From: "\"\"\"\"\"\"\"\"...\"\"\"\"\"\""@host.fubar
```

- Bug: a specially crafted “From” field corrupts the mailbox
- Let’s imagine the mailbox has 1000 emails, the corrupted message is the last to be parsed and entire input data is symbolic

Solution

- Use *Dynamic Taint Tracking*
- Narrow down the part of the input responsible for the crash
- Only mark _*that part*_ as symbolic

```
From: "\"\"\"\"\"\"\"\"\"...\"\"\"\"\"\"\""@host.fubar
```

↓

```
From: "\"\"\"\"\"\"\"\"\"...\"\"\"\"\"\"\""@host.fubar
```

Result

Table 2: Time needed to get the first recovery candidate when the whole document is symbolic ('Whole') and when only the potentially corrupt bytes are symbolic ('Partial').

Benchmark	Whole	Partial
`pr`	timeout (3600s)	5.1s
`pine`	timeout (3600s)	338.9s
`dwarfdump`	timeout (3600s)	2.8s
`readelf`	14.8s	< 1s

Result

```
  PINE 4.44   MESSAGE INDEX   Folder: INBOX(READONLY)  Message 1 of 6 NEW

  N    1 Dec  5 Bob                    (1381) Subject 1
  N    2 Dec  9 Alice                  (1497) Subject 2
  N    3 Dec 10 John                   (4627) Subject 3
  N    4 Dec 10 Jenny                  (1399) Subject 4
       5 Dec 16 Brian                  (2889) Subject 5
  N    6        "\"\\????????????        (81)

? Help          < FldrList   P PrevMsg      - PrevPage D Delete     R Reply
O OTHER CMDS    > [ViewMsg]  N NextMsg    Spc NextPage U Undelete   F Forward
```

Docovery: highlights

- We used concolic execution -> limiting the search to a single path and its divergences
- We selectively marked only certain bytes as symbolic -> no longer possible to branch at _*any*_ branch point
- We lazily collected execution paths (no SMT queries upfront)
- Selective symbex was the key performance enabler

Example #2: Shadow

Hristina Palikareva and Cristian Cadar

"Shadow of a Doubt: Testing for Divergences Between Software Versions" ICSE'16

"Shadow Symbolic Execution for Testing Software Patches" TOSEM'18

Shadow – the problem

- Software patches are at the core of development
- Example: bug fixes, new features, performance and usability improvements
- Testing software patches is hard
- They are poorly tested in practice
- May introduce bugs

Shadow – the motivation

- A lot of behaviors in the old and the new version are _*exactly*_ the same
- We may achieve 100% test coverage but not 100% behavior coverage

Shadow – the motivation

```
// Old
int gt_100(unsigned x) {
unsigned y = x;
if (y > 100)
  return 1;
else
  return 0;
}
```

```
// New
int gt_100(unsigned x) {
unsigned y = x + 1;
if (y > 100)
  return 1;
else
  return 0;
}
```

- Test cases: x = 0, x = 100, x = 101 -> 100% code coverage
- Only 50% new behavior coverage

Shadow: the idea

- Only focus on exploring the behaviors which are different across two versions
- Limiting the search space by pruning identical paths and entire execution subtrees
- We achieve that through *4-way fork*:
  - Both versions combined in a single symbolic execution instance
  - The old version shadows the new one

4-way fork

The best fork since 2-way fork

4-way fork

```
// Old
int gt_100(unsigned x) {
unsigned y = x;
if (y > 100)
  return 1;
else
  return 0;
}
```

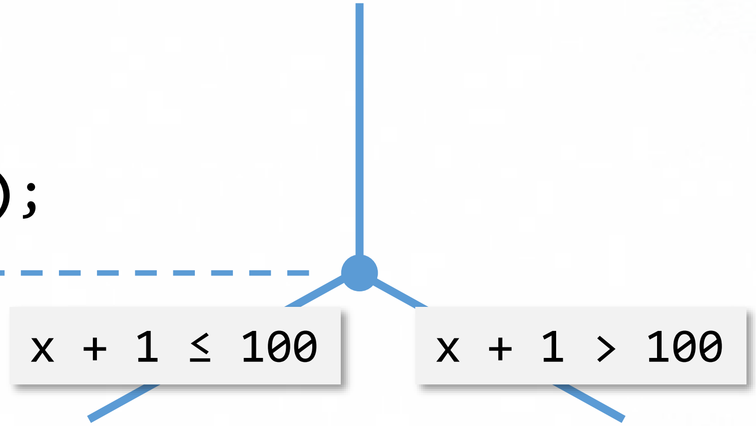
```
// New
int gt_100(unsigned x) {
unsigned y = x + 1;
if (y > 100)
  return 1;
else
  return 0;
}
```

4-way fork

```
// Combined
int gt_100(unsigned x) {
unsigned y = change(x, x + 1);
if (y > 100)
  return 1;
else
  return 0;
}
```

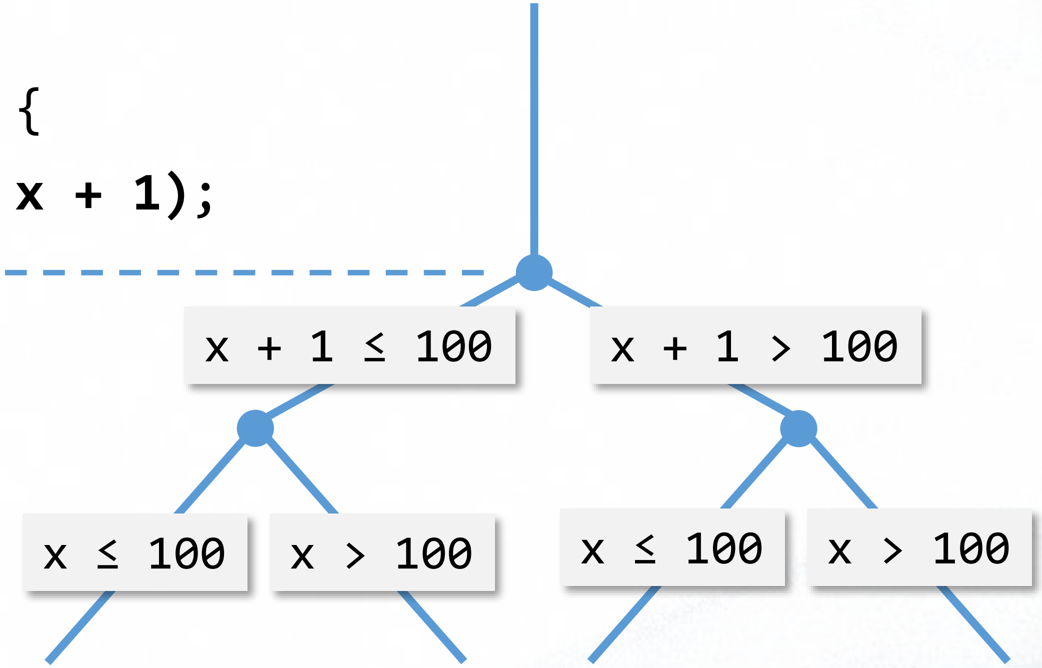
4-way fork

```
// Combined
int gt_100(unsigned x) {
unsigned y = change(x, x + 1);
if (y > 100)
  return 1;
else
  return 0;
}
```



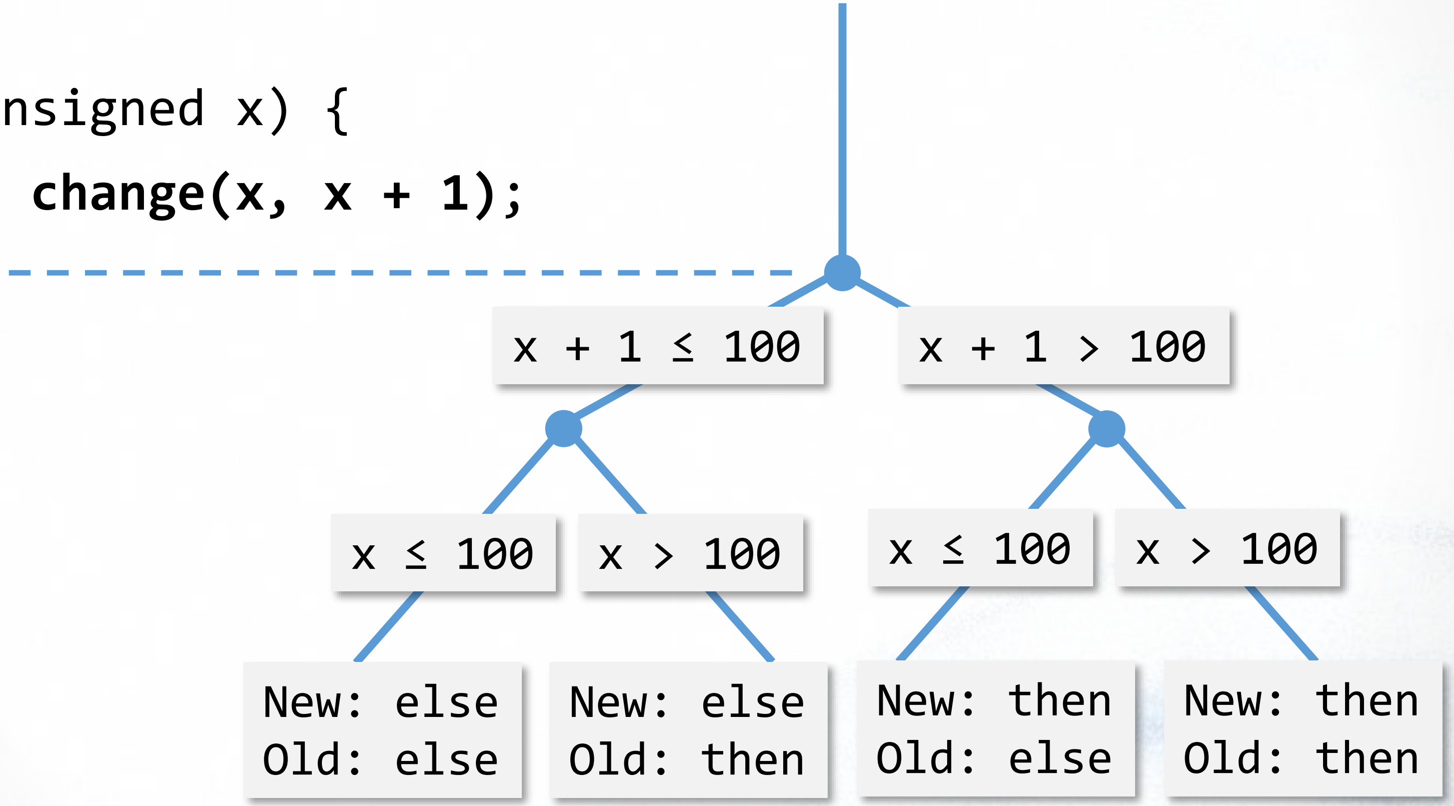
4-way fork

```
// Combined
int gt_100(unsigned x) {
unsigned y = change(x, x + 1);
if (y > 100)
  return 1;
else
  return 0;
}
```



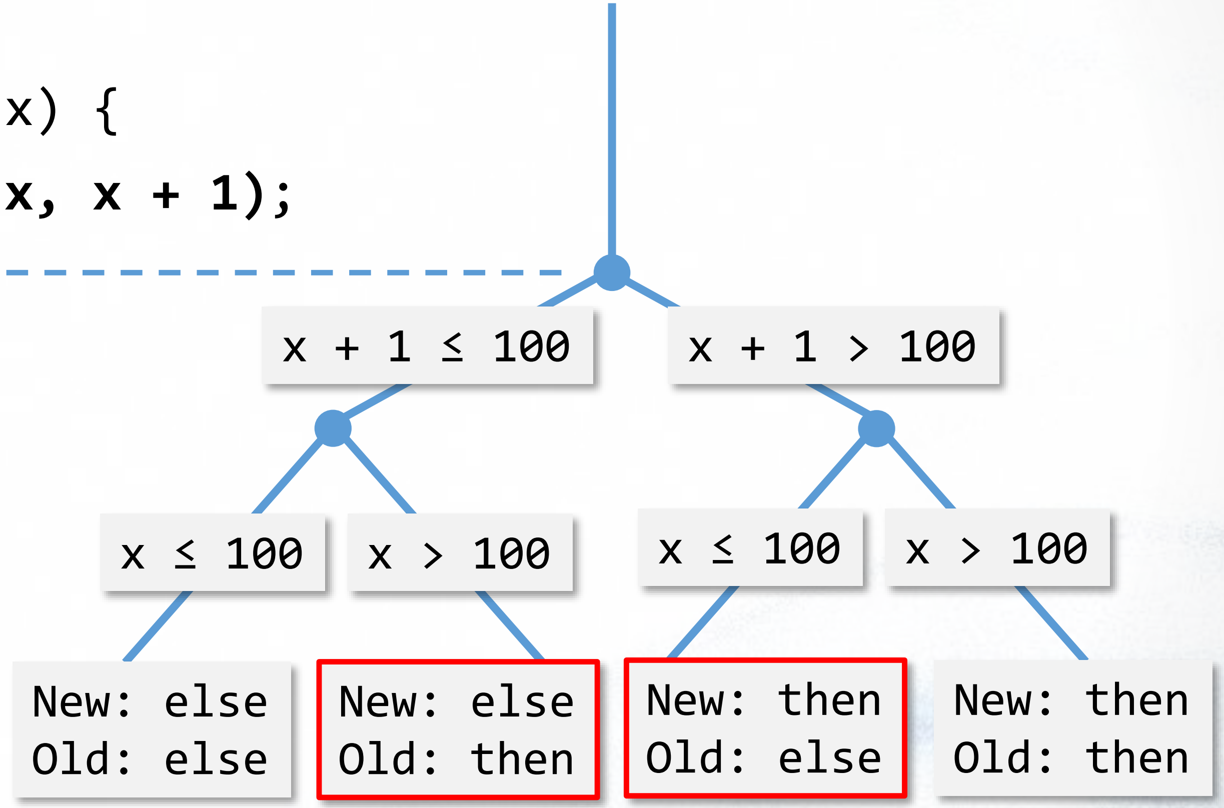
4-way fork

```
// Combined
int gt_100(unsigned x) {
unsigned y = change(x, x + 1);
if (y > 100)
  return 1;
else
  return 0;
}
```



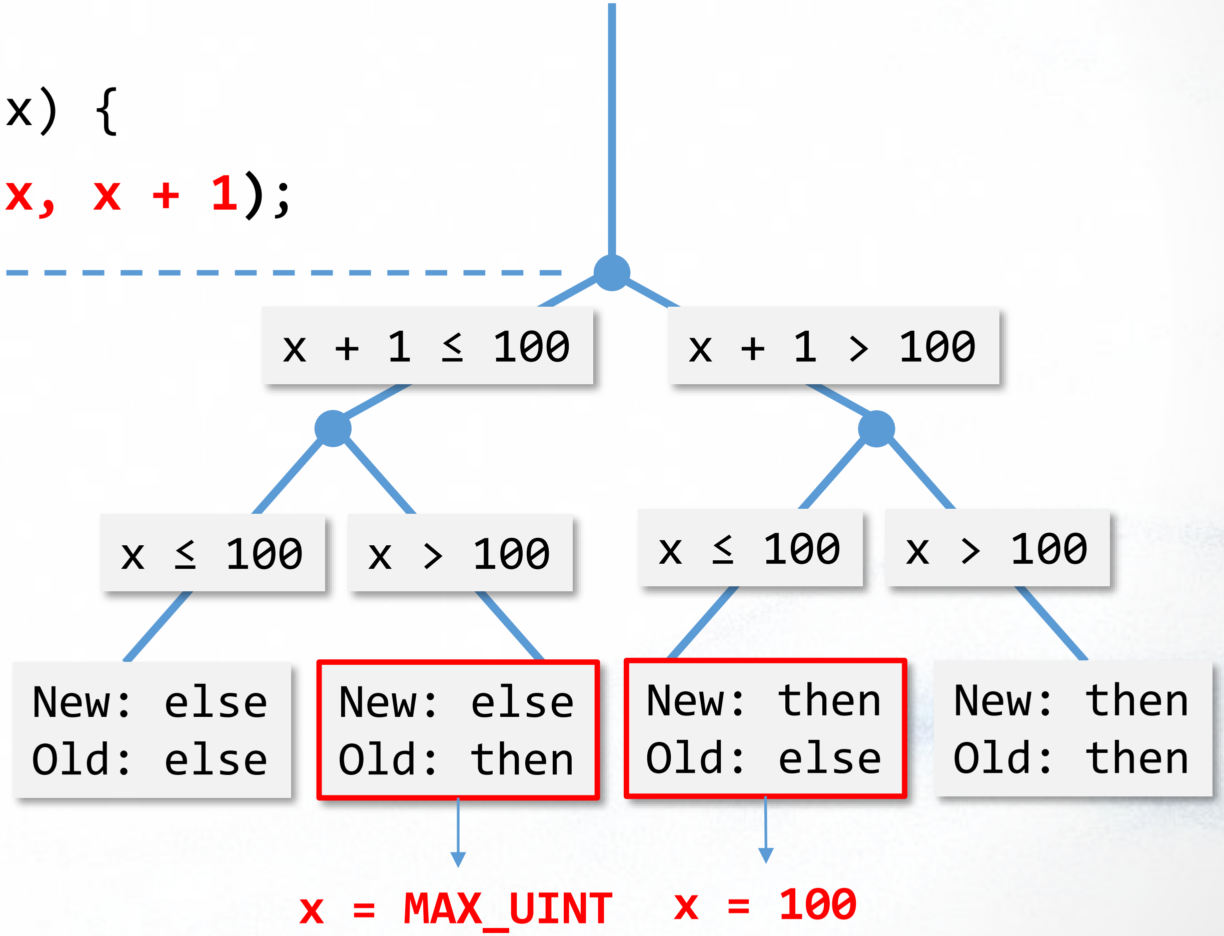
4-way fork

```
// Combined
int gt_100(unsigned x) {
unsigned y = change(x, x + 1);
if (y > 100)
  return 1;
else
  return 0;
}
```



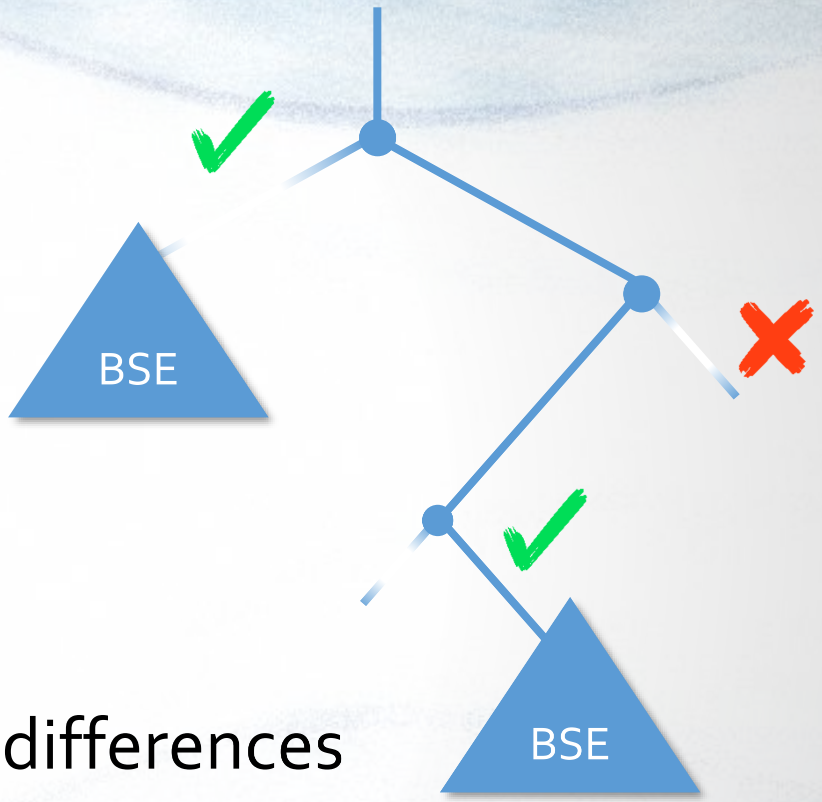
4-way fork

```
// Combined
int gt_100(unsigned x) {
unsigned y = change(x, x + 1);
if (y > 100)
  return 1;
else
  return 0;
}
```



Testing with Shadow

- Use test suite inputs
- Find divergent paths
- Perform bounded symbolic execution
- Check if divergences translate to functional differences
- Check program output, return code, memory violations



Shadow: highlights

- Concolic execution of test cases that touch the patch
- Pruning execution paths via 4-way fork
- Space efficiency: 2 versions combined in a single execution
- Unchanged common path prefix is executed only once

Example #3: Auto Off-Target

Bartosz Zator

"Auto Off-Target: Enabling Thorough and Scalable Testing for Complex Software Systems", ASE'22

Auto Off-Target – the problem

- Software is increasingly complex: size, variety of configurations
- Crucial software systems we rely on are often built with unsafe languages, e.g. C/C++
- Examples: OS kernels, bootloaders, modems, WLAN, IoT, automotive, firmware, etc.

Auto Off-Target – the problem

- Working with such systems is challenging, e.g.
  - The code base size
  - Variety of configurations
- Thorough testing is necessary but often difficult:
  - Custom hardware –> no virtualization available
  - Non-trivial setup of testing and debugging
  - Toolchain not always available on device
  - Hard to run techniques such as symbolic execution

Auto Off-Target – the problem

- Challenge #1: large system size leads to path explosion
- Challenge #2: not easy to build
- Challenge #3: no obvious entry points

```
$ klee kernel.bc <my symbolic input>
```

- Modern smartphone: over 70M LOC, > 300k C/C++ source files, ARM-based

One does not simply run symbolic execution on a bootloader.

Boromir

On-target testing: baseband message parser

0001111001101110001011001001001001001000
0111001101001001001100011001100111100110 1

- Setup a testing mobile network
- Send test messages over the air
- When a crash occurs: capture logs, start analysis
- Reboot and repeat

Motivation

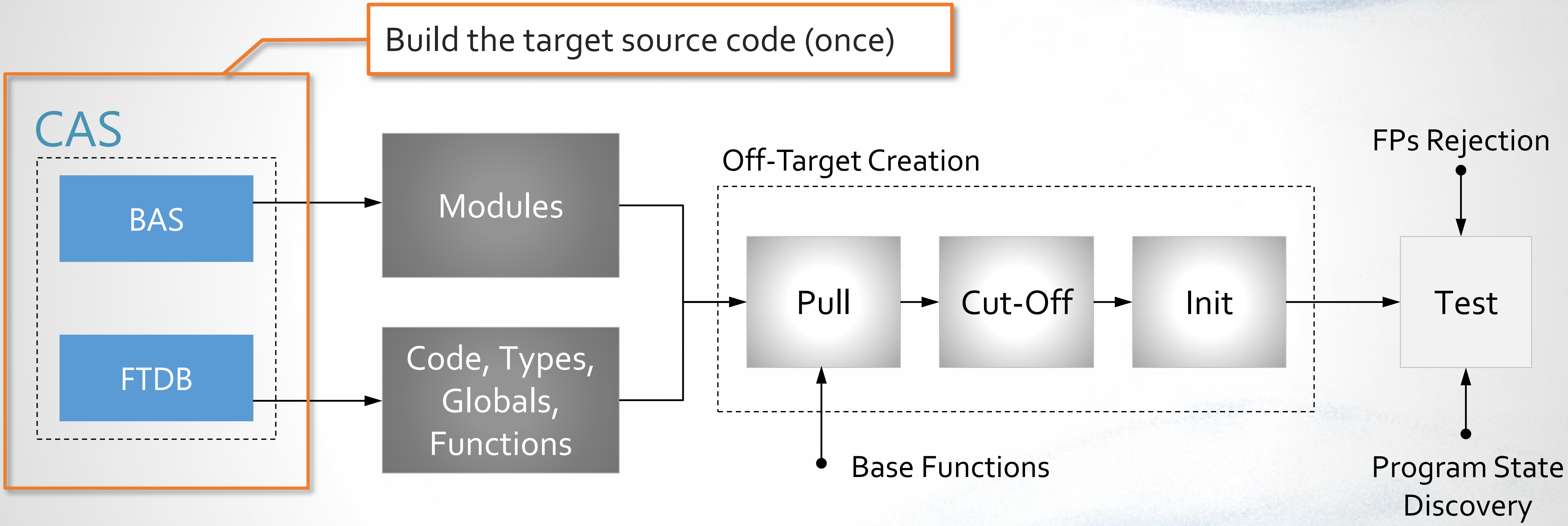
Many components, e.g., a modem or a bootloader, are hard to test on-target (on the device) and difficult to extract for off-target testing.

Can we thoroughly test system-level C/C++ software regardless of the component and provide stronger quality guarantees?

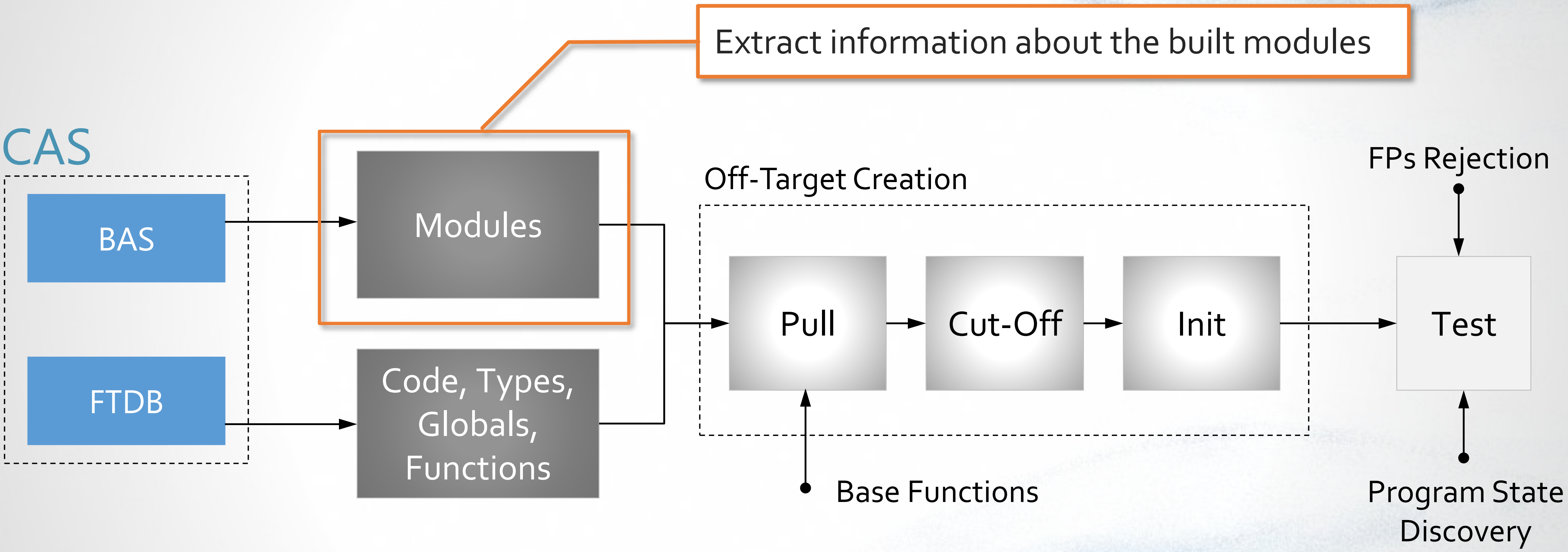
AoT: the idea

- Automatically extract selected critical part of target code
- Create a test harness, called an Off-Target (OT) program
- Test the harness on powerful x86_64 servers
- We can use available toolchain for fuzzing, analysis, debugging, etc.
- In particular, we can run symbex on OT

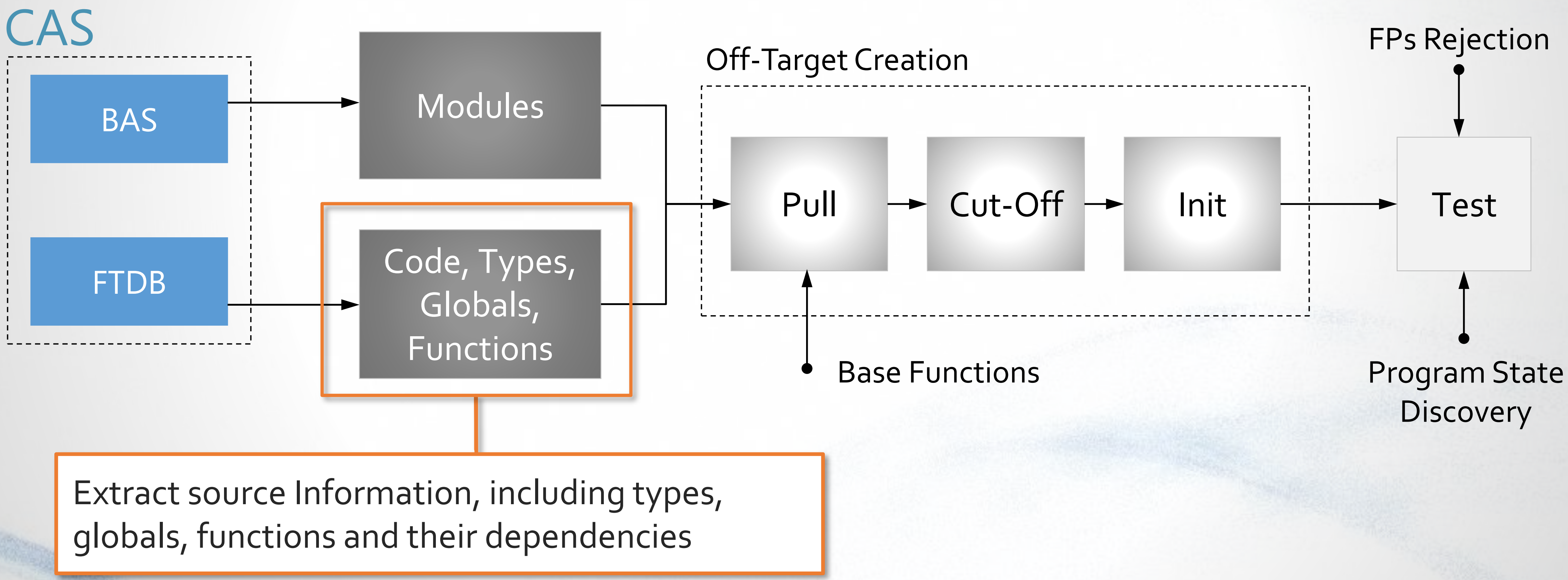
AoT: overview



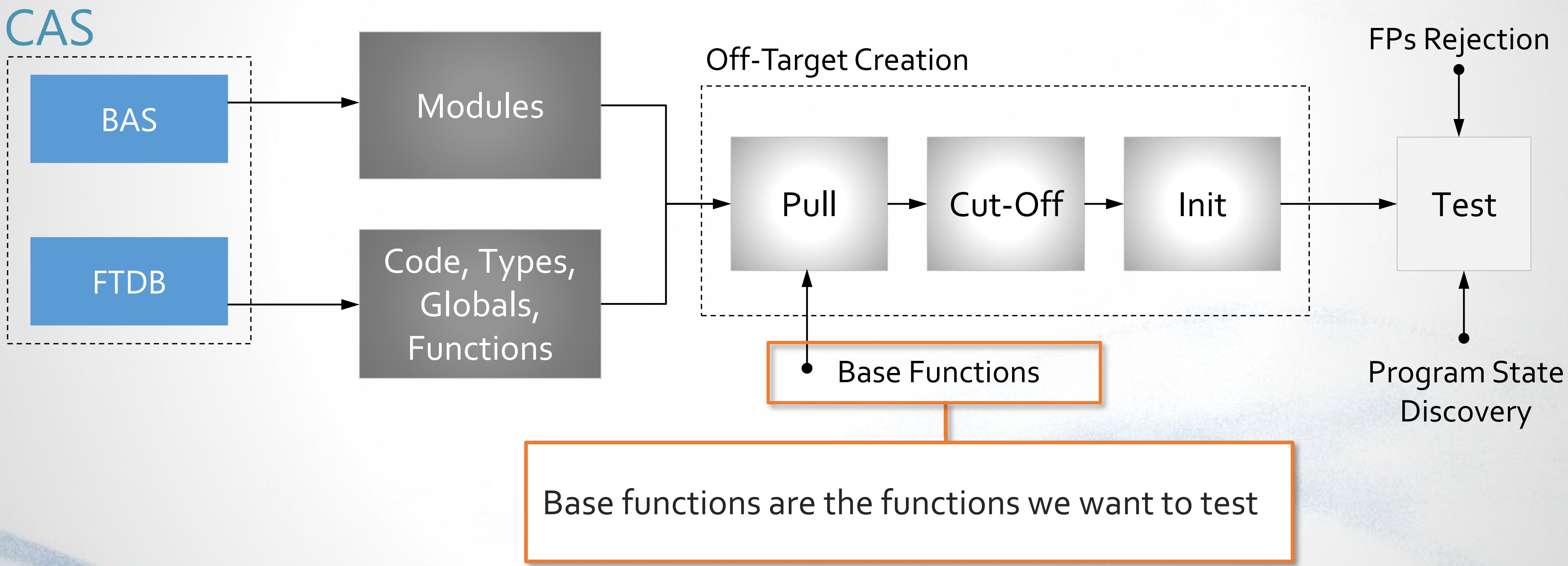
AoT: overview



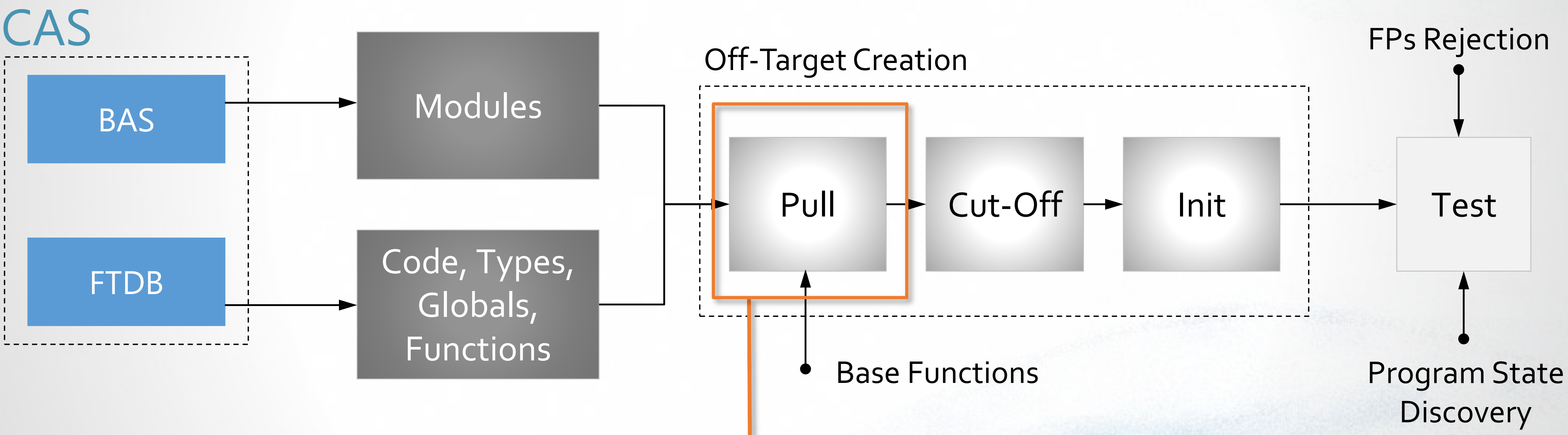
AoT: overview



AoT: overview

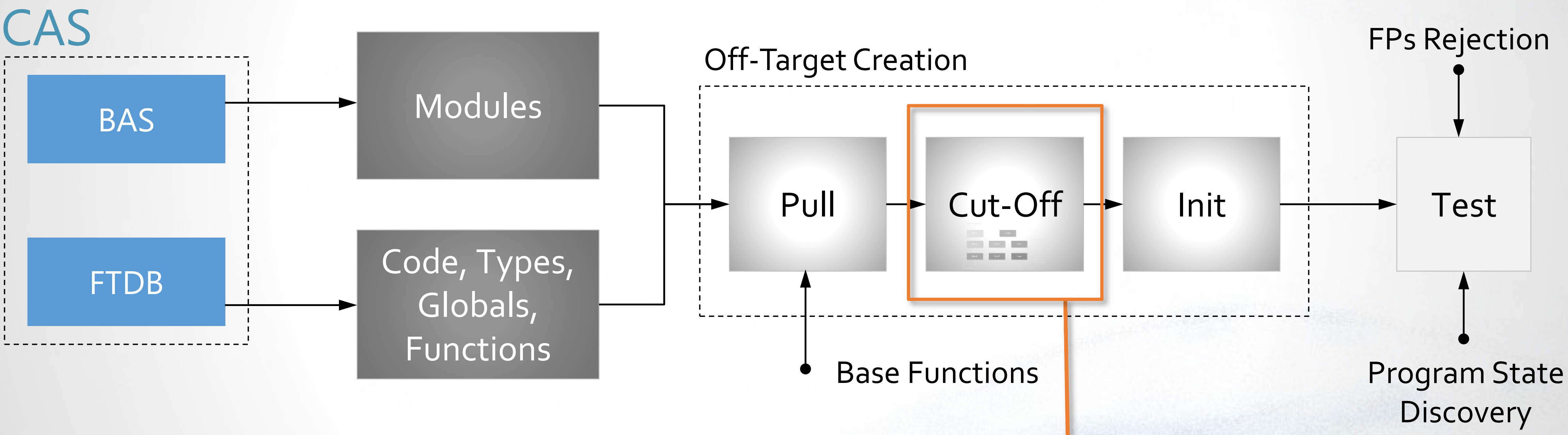


AoT: overview



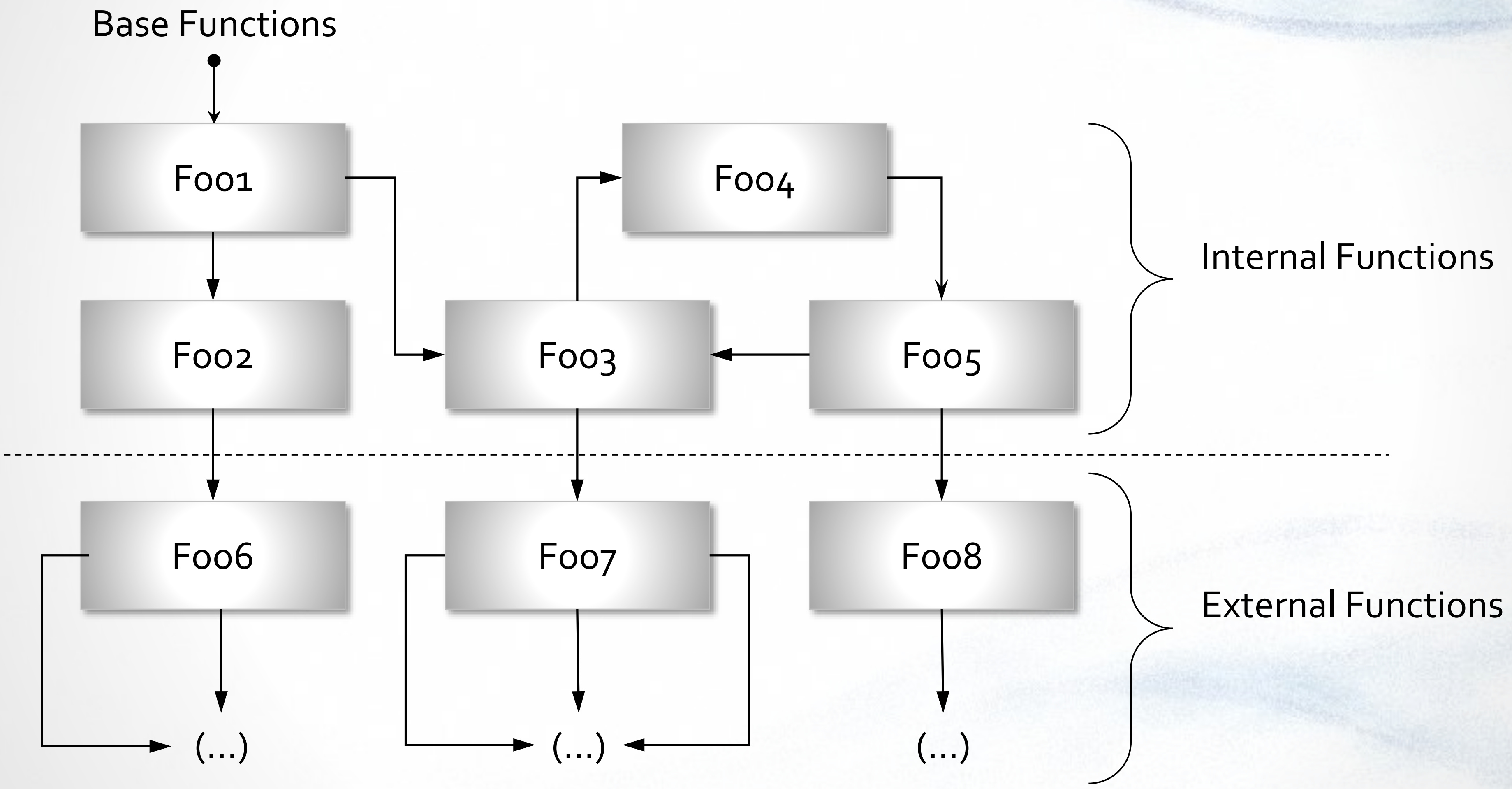
Recursively pull in all function in a call hierarchy of the tested function

AoT: overview

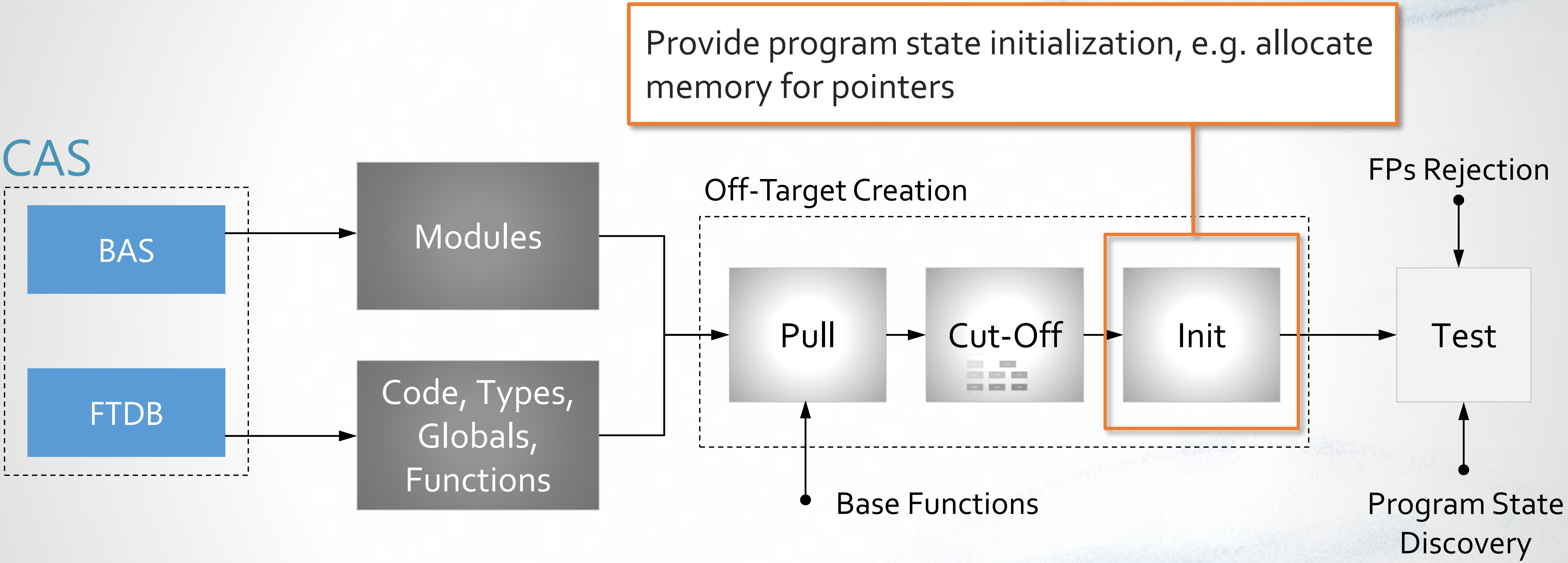


Cut off the code the is outside of the current module + generate stubs

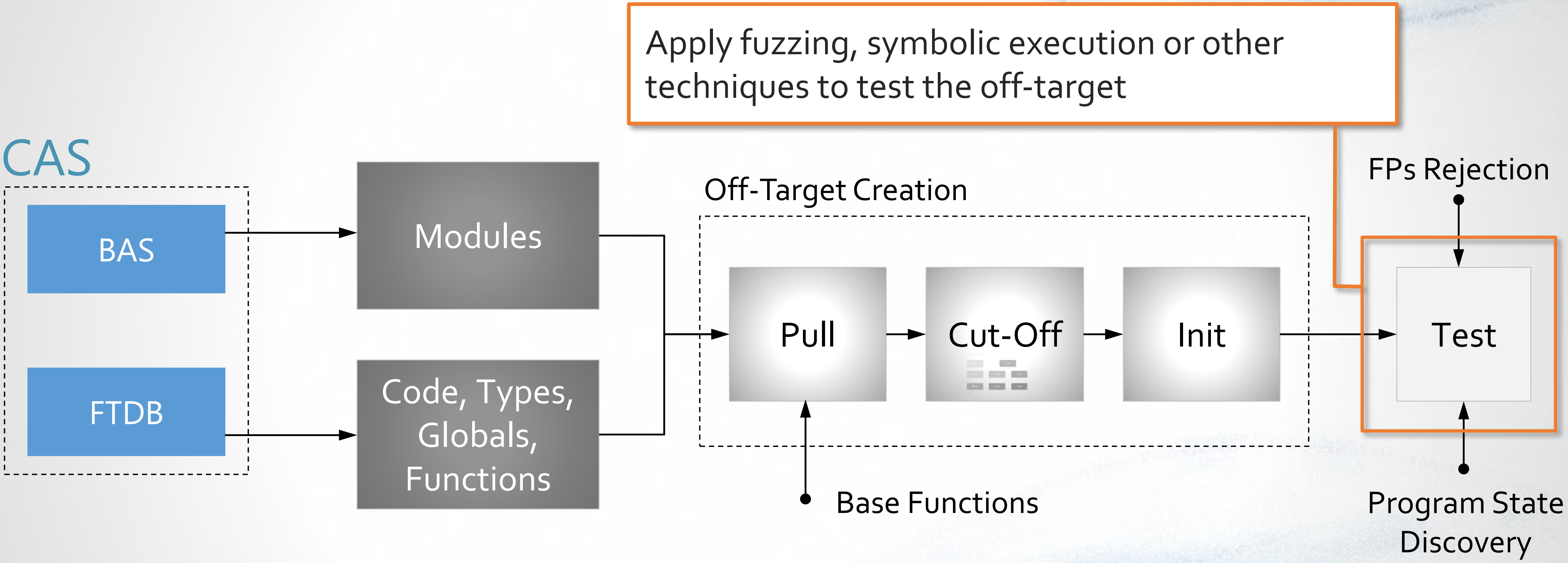
Implementation of cut-off



AoT: overview



AoT: overview



How does it work in practice?

- Example: test IncrementalFS ioctl handler from AOSP kernel
- 1) Perform the kernel build to obtain CAS databases (once)
- 2) Generate OT for `pending_reads_dispatch_ioctl()`: ~42s

```
$ aot.py --config=./cfg.json
  --product=aosp --version=cheetah_android-13.0.0_r66 --build-type=eng
  --functions pending_reads_dispatch_ioctl
  --output-dir=pending_reads_dispatch_ioctl_out
  --db=vmlinux_db_aot.img
```

What's inside OT

```
// test driver and main header
aot.c
aot.h

// aot libraries & headers
aot_fuzz_lib.c
aot_dfsan.c.lib
aot_mem_init_lib.c
aot_lib.c
aot_log.c
aot_recall.c
aot_replacements.h
fptr_stub.c.template
fptr_stub_known_funcs.c.template
vlayout.c.template

// literals for fuzzing
aot_literals

Makefile
```

```
// source files
common_18.c
core_920.c
cpufeature_1345.c
data_mgmt_2430.c
file_1923.c
format_3435.c
fse_compress_20.c
fsnotify_372.c
...
percpu-rwsem_2027.c
pseudo_files_1525.c
read_write_2502.c
rwsem_2924.c
splice_1300.c
strnlen_user_3295.c
tree_3058.c
util_2104.c
verity_1115.c
vfs_2350.c
```

```
// stub files
attr_stub_1520.c
auditsc_stub_496.c
common_stub_18.c
core_stub_920.c
cred_stub_767.c
data_mgmt_stub_2430.c
dcache_stub_957.c
filemap_stub_3843.c
...
open_stub_3030.c
percpu-rwsem_stub_2027.c
read_write_stub_2502.c
rwsem_stub_2924.c
srcutree_stub_1825.c
timekeeping_stub_3614.c
tree_stub_3058.c
verity_stub_1115.c
vfs_stub_2350.c
xattr_stub_1884.c
```

What’s inside OT

```
// test driver and main header
aot.c
aot.h

// aot libraries & headers
aot_fuzz_lib.c
aot_dfsan.c.lib
aot_mem_init_lib.c
aot_lib.c
aot_log.c
aot_recall.c
aot_replacements.h
fptr_stub.c.template
fptr_stub_known_funcs.c.template
vlayout.c.template

// literals for fuzzing
aot_literals

Makefile
```

```
// source files
common_18.c
core_920.c
cpufeature_1345.c
data_mgmt_2430.c
file_1923.c
format_3435.c
fse_compress_20.c
splice_1300.c
strnlen_user_3295.c
tree_3058.c
util_2104.c
verity_1115.c
vfs_2350.c
```

```
// stub files
attr_stub_1520.c
auditsc_stub_496.c
common_stub_18.c
core_stub_920.c
cred_stub_767.c
data_mgmt_stub_2430.c
dcache_stub_957.c
lemap_stub_3843.c
.
en_stub_3030.c
rcpu-rwsem_stub_2027.c
ad_write_stub_2502.c
wsem_stub_2924.c
srcutree_stub_1825.c
timekeeping_stub_3614.c
tree_stub_3058.c
verity_stub_1115.c
vfs_stub_2350.c
xattr_stub_1884.c
```

Targets: afl, aflgo, asan, daikon, debug, dfsan, GCC fanalyzer, FramaC, gcov, **klee**, msan, symcc, ubsan

What's inside the OT

- Types: 4223
- Struct types: 1089
- Globals: 14
- Internal funcs: 251
- External funcs: 90

```
$ cloc .
Language                files          blank        comment           code
----------------------------------------------------------------------------
C/C++ Header                7           1802            776          15691
C                          60           2268           6403          14422

// excluding aot.c

$ cloc .
Language                files          blank        comment           code
----------------------------------------------------------------------------
C/C++ Header                7           1802            776          15691
C                          59           1825           3777           4404
```

Let’s test it!

- Build targets for KLEE and AFL++
- Run KLEE for 1h, then AFL++ with symcc for 1h
- Results: 47TCs, 8 crashes, including 3 FPs and …

```
$ ./asan out_dir/default/crashes/id\:000007\,sig\:06\,src\:000044+000009\,time\:2557397\,execs\:2034693\,op\: ...
=================================================================
==3794212==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000005cf1 at pc 0x000000492f60 bp
0x7ffd1fa48110 sp 0x7ffd1fa478d8
WRITE of size 17 at 0x602000005cf1 thread T0
    #0 0x492f5f in __asan_memcpy asan+0x492f5f
    #1 0x4c632b in ioctl_get_read_timeouts pseudo_files_1525.c:873:13
    #2 0x4c3286 in pending_reads_dispatch_ioctl pseudo_files_1525.c:179:16
    #3 0x4c3172 in wrapper_pending_reads_dispatch_ioctl_112617 pseudo_files_1525.c:987:9
    #4 0x5620b5 in main aot.c:13079:19
```

WARNING: the following slides contain source code in a memory-unsafe programming language.

```c
// aot.c
int main(int AOT_argc, char *AOT_argv[]) {
    ...
    // Global vars init
    aot_memory_init(&fsnotify_mark_srcu, sizeof(struct srcu_struct),
                    0 /* fuzz */, 0);
    ...
    // Call site for function 'pending_reads_dispatch_ioctl'
    {
      struct file *f;
      aot_memory_init_ptr((void **)&f, sizeof(struct file), 1 /* count */,
                          0 /* fuzz */, 0);
      ...
      aot_memory_init_func_ptr(&f->f_mapping->a_ops->readpage,
                               aotstub_f_f_mapping_a_ops_readpage);
      unsigned int req;
      aot_memory_init(&req, sizeof(unsigned int), 1 /* fuzz */, 0);

      unsigned long arg;
      unsigned long *arg_ptr;
      aot_memory_init_ptr((void **)&arg_ptr, sizeof(unsigned long), 512,
                          1 /* fuzz */, "aot_var_1");
      aot_tag_memory(arg_ptr, sizeof(unsigned long) * 512, 0);
      aot_tag_memory(&arg_ptr, sizeof(arg_ptr), 0);
      arg = (unsigned long)arg_ptr;

      ret_value = wrapper_pending_reads_dispatch_ioctl_112617(f, req, arg);
```

The bug

```
static long pending_reads_dispatch_ioctl(struct file *f, unsigned int req,
                                          unsigned long arg)
{
    struct mount_info *mi = get_mount_info(file_superblock(f));

    switch (req) {
        case INCFS_IOC_CREATE_FILE:
            return ioctl_create_file(f, (void __user *)arg);
        case INCFS_IOC_PERMIT_FILL:
            return ioctl_permit_fill(f, (void __user *)arg);
        case INCFS_IOC_CREATE_MAPPED_FILE:
            return ioctl_create_mapped_file(f, (void __user *)arg);
        case INCFS_IOC_GET_READ_TIMEOUTS:
            return ioctl_get_read_timeouts(mi, (void __user *)arg);
        case INCFS_IOC_SET_READ_TIMEOUTS:
            return ioctl_set_read_timeouts(mi, (void __user *)arg);
        case INCFS_IOC_GET_LAST_READ_ERROR:
            return ioctl_get_last_read_error(mi, (void __user *)arg);
        default:
            return -EINVAL;
        }
```

```c
static long ioctl_get_read_timeouts(struct mount_info *mi, void *arg) {
    struct incfs_get_read_timeouts_args *args_usr_ptr = arg;
    struct incfs_get_read_timeouts_args args = {};
    int error = 0;
    struct incfs_per_uid_read_timeouts *buffer;
    int size;
    if (copy_from_user(&args, args_usr_ptr, sizeof (args))) {
        return -22;
    }
    if (args.timeouts_array_size_out > 4096) {
        return -22;
    }
    buffer = kzalloc(args.timeouts_array_size_out, (((gfp_t)(1024U | 2048U)) | ((gfp_t)64U)));
    if (!buffer) {
        return -12;
    }
    spin_lock(&mi->mi_per_uid_read_timeouts_lock);
    size = mi->mi_per_uid_read_timeouts_size;
    if (args.timeouts_array_size < size) {
        error = -7;
    } else {
        if (size) {
            memcpy(buffer, mi->mi_per_uid_read_timeouts, size);
        }
    }
```

```
static long ioctl_get_read_timeouts(struct mount_info *mi, void *arg) {
    struct incfs_get_read_timeouts_args *args_usr_ptr = arg;
    struct incfs_get_read_timeouts_args args = {};
    int error = 0;
    struct incfs_per_uid_read_timeouts *buffer;
    int size;
    if (copy_from_user(&args, args_usr_ptr, sizeof (args))) {
        return -22;
    }
    if (args.timeouts_array_size_out > 4096) {
        return -22;
    }
    buffer = kzalloc(args.timeouts_array_size_out, (((gfp_t)(1024U | 2048U)) | ((gfp_t)64U)));
    if (!buffer) {
        return -12;
    }
    spin_lock(&mi->mi_per_uid_read_timeouts_lock);
    size = mi->mi_per_uid_read_timeouts_size;
    if (args.timeouts_array_size < size) {
        error = -7;
    } else {
        if (size) {
            memcpy(buffer, mi->mi_per_uid_read_timeouts, size);
        }
    }
```

To KLEE, or not to KLEE, that is the question

Hamlet

The role of symbex in AoT

- Find bugs
- Bootstrap the program state, provide “data virtualization”
- Is that really helping? Let’s check on 4k entry points in AOSP kernel:

	KLEE + AFL/symcc	AFL/symcc	AFL only
# TCs total	50.387 + 73.951	73.750	71.768

Program state discovery

- We over-approximate program state values
- This leads to FPs: behaviors that are only possible in the OT code
- In the kernel, a big source of FPs is the system state, not related to user-controlled data

```
static long pending_reads_dispatch_ioctl(struct file *f, unsigned int req,
                                         unsigned long arg)
```

KFLAT: selective code-level memory dumps

- KFLAT is a novel approach to memory dumps
- *Selectively* dumps system memory on the *source code* level
- The dumps can be restored on a different machine but with *the same* code structures

$\text{AoT}_\flat$: AoT + KFLAT

- We collect *real* memory values on the device and plug them into OTs
- System state is concrete, user data is symbolic / fuzzed
- Also, we could selectively mark data as symbolic if needed
- Advantages:
  - Less over-approximation -> fewer FPs
  - Greatly limiting the search space on non user-controlled data

AoT: highlights

- Makes is possible to execute parts of complex low-level systems
- Enables easy symbex on low-level code
- Symbex enables execution of OT without knowing the program state
- AoT reduces complexity by limiting the executed code size
- AoT provides flexibility on how much data is symbolic

Mobile Security Group @ SRPOL

- We have some other cool projects in Mobile Security Group
- We release our tools to open source
  - AoT: https://github.com/Samsung/auto_off_target
  - CAS: https://github.com/Samsung/cas
  - KFLAT: https://github.com/Samsung/kflat
  - SEAL: https://github.com/Samsung/seal

Mobile Security Group @ SRPOL

- We give talks
  - DPE Summit'23: https://youtu.be/FZrhHgor4NE?si=4hv77EtI-CZN5E4b
  - OSS NA'23: https://youtu.be/Ynunpuk-Vfo?si=i83R6ZANwpXPASet
  - LSS NA'22: https://youtu.be/M7gl7MFU_Bc?si=LmLmySHbwINSldCg&t=648
- Interested? Feel free to reach out!

Agenda

- Symbex & others: the state of the art
- Docovery, Shadow & AoT: selective and incremental symbex
- **SOAR: in search of the secret sauce**
- Academia & Industry: perspectives matter
- Future outlook for symbex

How can we help symbex SOAR?

- We propose the following directions:
- Selective
- Open-source
- Approachable
- Real-world

S is for Selective

by data

Selectively mark only certain bytes / variables as symbolic

by target

Symbolically execute selected parts of larger systems

- Reasoning: less symbolic data => smaller search space

O is for Open Source

- Standing on the shoulders of giants
- Opportunity to converge various “little” tweaks
- Add-on: peer reviews usually make the end result better
- Caveat: for this to work, forks need to go back to the mainline
- AoT: 2 PRs for KLEE (one in 3.1), 4 PRs for LLVM (DFSAN)

A is for Approachable

- Mind the audience: some might not have heard of SMT
- One-liner is king
- Ideally: easy to deploy, easy to use, easy to analyze, easy to extend
- User docs != developer docs (good to have both)

R is for Real-World

- Real-world users work on real-world targets
- Aim for hard targets: web browsers, embedded, stateful, etc.
- Needed: scalability, ease of deployment

Agenda

- Symbex & others: the state of the art
- Docovery, Shadow & AoT: selective and incremental symbex
- SOAR: in search of the secret sauce
- **Academia & Industry: perspectives matter**
- Future outlook for symbex

Academia & Industry

- Different objectives: research work vs product development

↓ translates to

- What people have time working on

Common misconceptions

- Academia:
  - Industry has unlimited resources for engineering
  - Engineering details can be sorted out easily
- Industry:
  - The paper should work out of the box
  - We have the best stuff, not much interesting stuff comes out of Academia

Academia & Industry

- How are the tools evaluated in the Industry:
    - With constrained resources (time & people), often as a side task
    - On a specific real-world target
    - Either it works or it doesn't
    - Research contribution might be – sadly – underappreciated
- What should a great symbex tool strive for:
    - Ease of use, being straightforward
    - Scalability
    - The tool outcomes are easy to understand and process

Agenda

- Symbex & others: the state of the art
- Docovery, Shadow & AoT: selective and incremental symbex
- SOAR: in search of the secret sauce
- Academia & Industry: perspectives matter
- **Future outlook for symbex**

Future outlook for symbex

- Symbex now more of a boutique approach than commonplace
- If a major breakthrough doesn’t happen (e.g. quantum symbex, custom HW, etc.), we need to keep working on the little things that add up
- How do we move forward?

Future outlook for symbex

- Academia:
  - Aim for real-world applications
  - Often, a lot of value comes from the little engineering tricks
- Industry:
  - Merge changes back to the mainline
  - Spend more resources to appreciate research

Symbex's not dead, Jim

Dr Leonard McCoy, USS Enterprise

Summary

- There is no secret sauce – just a lot of engineering and small tweaks
- Since we can't defeat the path explosion problem we need to find smart ways around it
- Examples: Docovery, Shadow & Auto Off-Target

Summary

- We propose the following directions for symbex:
  - Selective
  - Open-source
  - Approachable
  - Real-world
- Symbex can and should soar!

Credits

- The following graphics were used in the slides:
- Image by Freepik
- Image by Freepik
- Image by Harryarts on Freepik
- Image by macrovector on Freepik
- Image by pikisuperstar on Freepik
- Icon by Freepik
- Icon by Freepik
- Image by Freepik
- Image by rawpixel.com on Freepik
- Image by KamranAydinov on Freepik
- Image by starline on Freepik