SAMSUNG
Mobile Security Team Warsaw, Poland

AUTHORS:

Tomasz Kuchta
@Tomasz_Kuchta

Bartosz Zator
@bartosz_zator

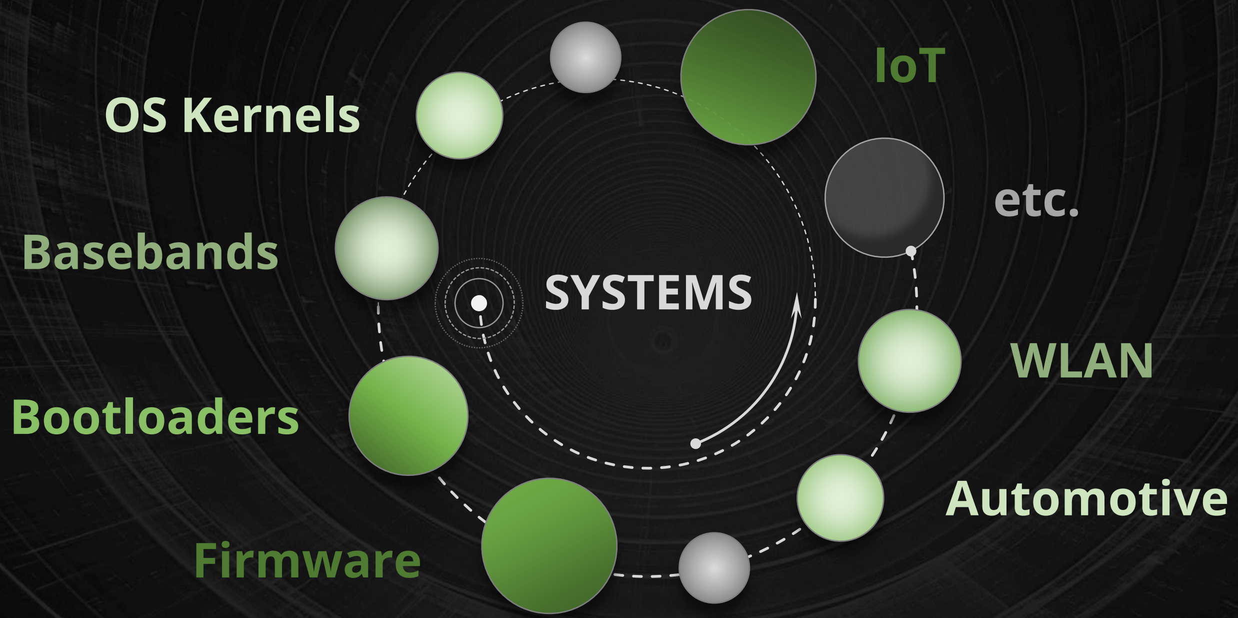
CAS & AoT

Enabling Symbolic Execution on Complex System Code via Automatic Test Harness Generation

KLEE Workshop 2022

Motivation:
Systems software is complex

Many crucial systems are built with unsafe languages such as C/C++



These systems build the foundation of infrastructure we rely on
Thorough and systematic testing is paramount

Motivation: Systems software is complex

Working with such systems poses significant challenges, e.g.:

- Code base size
- Variety of product configurations

Testing is also 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

Motivation: Systems complexity

Example: a modern smartphone

~74M lines of code

~thousands of developers

~330k source files [.c, .h, .cpp, .cxx, .hpp]

Many software layers

Bootloader, Linux kernel, modem, native framework

Engineers cannot handle S/W stack without supporting tools
Multiple potential vectors of attack

Our contributions

New approach to handle the complexity of large scale S/W systems

We propose 2 new tools aimed at complex software systems

CAS: Code Aware Services

- Provides insight into how a S/W product is made
- Automates source code related operations
- Collects build and source information
- Exposes the information to external applications

AoT: Auto Off-Target

- Built on top of CAS
- Automatically generates testable off-target code

CAS: Code Aware Services

GitHub: https://github.com/Samsung/CAS

Workshop 2022

Introduction

Overview of the CAS system

A system that provides insight into how a S/W product is made and automates source code related operations

CAS

Code Aware Services

BAS

Build Awareness Service

Provides information gathered during the build process

FTDB

Function/Type database

Transforms selected features from relevant source files into easily accessible format

Useful in various S/W engineering jobs: code search, test automation, fuzzing, code analysis etc.

Main focus on automation of vulnerability detection

BAS: Build Awareness Service

Main building blocks of the CAS system

CAS

Code Aware Services

BAS

Build Awareness Service

FTDB

Function/Type database

BAS overview

Build Awareness Service operation

There is a lot of information we can mine from the build process, e.g.:

- Configuration
- Structure
- Dependencies
- Tools used
- Commands

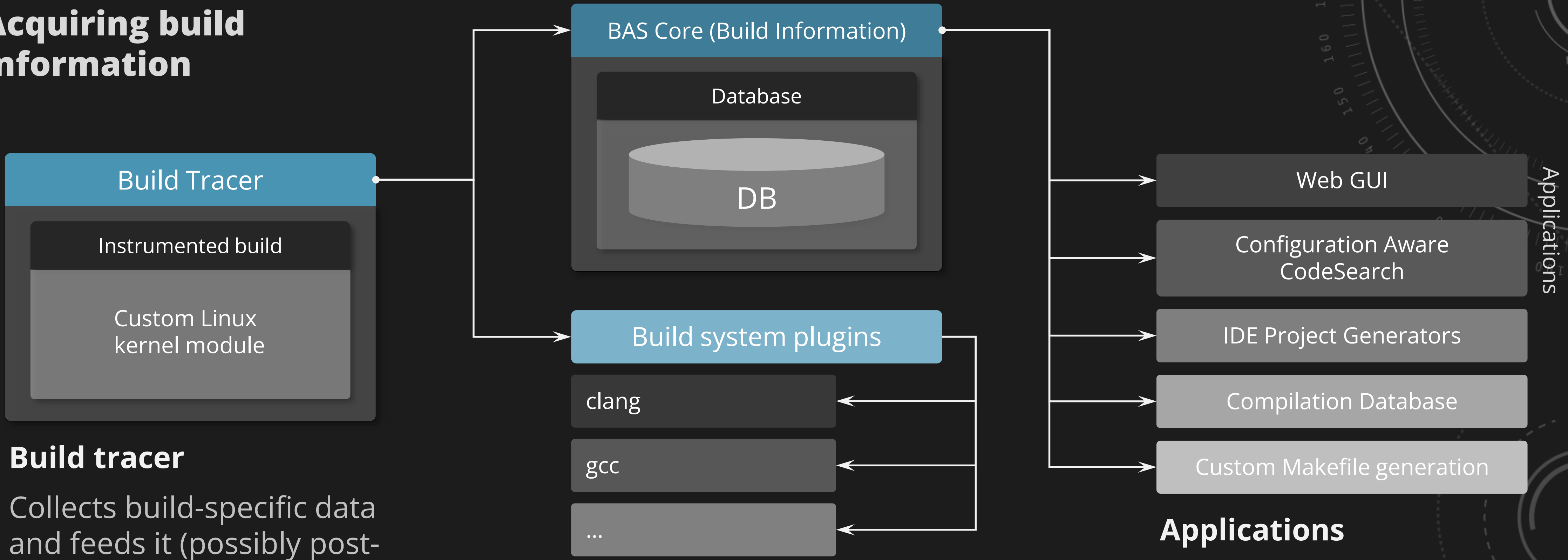
What is provided by BAS

1. Traces the build process with low overhead: ~5% for AOSP
2. Collects information on interconnections and dependencies between source code components
3. Makes the information available to other applications
4. Works at scale, e.g., AOSP Android with 1.1 million files

BAS architecture

High-level components of BAS

Acquiring build information



Build tracer

Collects build-specific data and feeds it (possibly post-processed) to the database for later retrieval

Core engine (service)

Service that reads data from database and serves it to connected clients

Applications

Specific applications/tools that use build information to operate efficiently

Workshop 2022

FTDB (Code DB)

Main building blocks of the CAS system

CAS

Code Aware Services

BAS

Build Awareness Service

FTDB

Function/Type database

Workshop 2022

FTDB overview

Introduction of Code Database

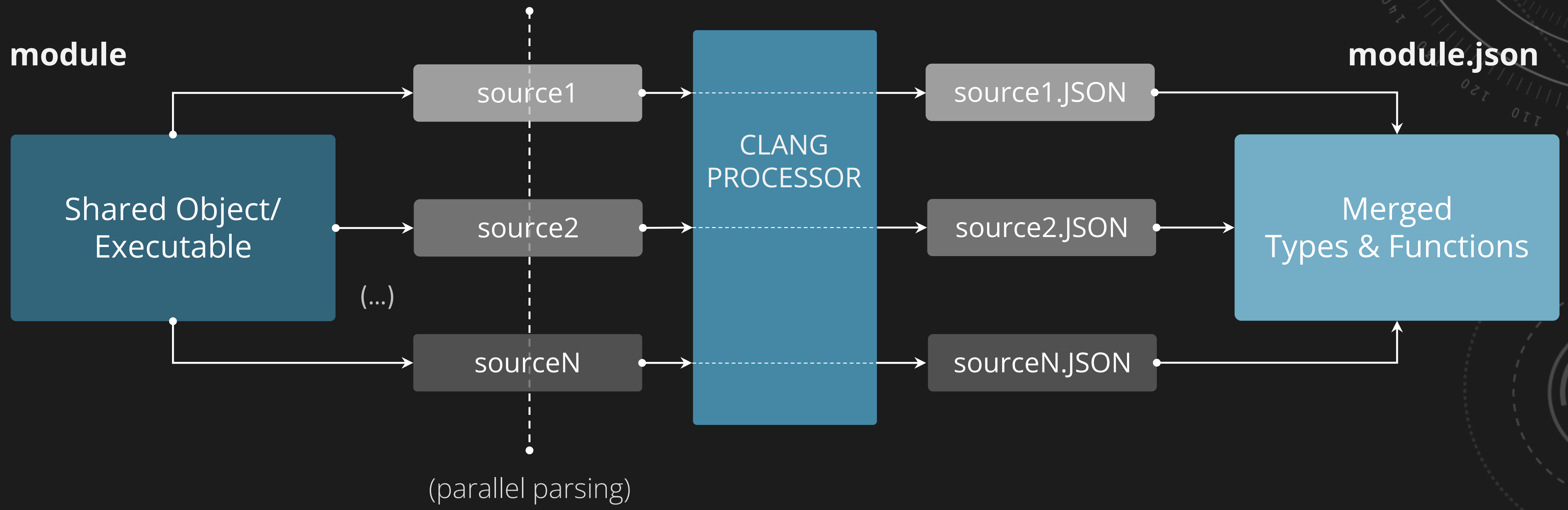
1. **Database of selected code features easily accessible** to external applications
2. **Uses a libclang-based code processor** to parse **AST** in-memory
3. **Mines information on:**
   - All defined types, functions and globals
   - Dependencies, function calls (including via pointers)
   - Initialization and assignment of function pointers
   - Source: type definitions, global initializers, function bodies (pre and post processed)
4. **Makes the information available to other applications: JSON** or **DB**

Creation of the FTDB

JSON database of selected code features easily accessible to external applications

Function/Type DB on module basis

Multiple source files merged into a final JSON representation



JSON database is useful for custom applications that operate on source code

FTDB Examples

Workshop 2022

Function information

in JSON

Extracting various function attributes, e.g.: name, source attributes, argument information, call information, referenced types, body, source literals, argument taints, selected expressions, referenced variables, etc.

.c

```c
static long v4l2_ioctl(struct file *filp, unsigned int cmd,
                                            unsigned long arg)
{
        struct video_device *vdev = video_devdata(filp);
        int ret = -ENODEV;

        if (vdev->fops->unlocked_ioctl) {
                if (video_is_registered(vdev))
                        ret = vdev->fops->unlocked_ioctl(filp,
                                                    cmd, arg);
        } else
                ret = -ENOTTY;

        return ret;
}
```

.json

```
{
        "name": "v4l2_ioctl",
        "id": 3,
        "fid": 0,
        "fids": [ 0 ],
        "nargs": 3,
        "variadic": false,
        "linkage": "internal",
        "attributes": [ ],
        "body": "static long v4l2_ioctl...",
        "location": "drivers/media/v4l2-core/v4l2-dev.c:353",
        "literals":{
                "integer": [ 19, 25 ],
        },
        "calls": [ 2, 1 ],
        "funrefs": [  1, 2 ],
        "refs": [  12, 1, 4, 10, 13, 14, 16 ],
        "types": [ 9, 10, 1, 4 ],
        "locals": [
                {
                        "id": 0,
                        "name": "filp",
                        "parm": true,
                        "type": 10,
                        "static": false,
                        "used": true,
                        "location":
                                "drivers/media/v4l2-core/v4l2-dev.c:37:"
                },
        ],
}
// (...)]
```

Workshop 2022

Function information

in JSON (cont.)

Extracting variable information used in various expressions inside function body, i.e., indirection operator, array expression, member expression, offsetof, cast, if, switch, loop conditions, return statement, etc.

.c

```
int param_set_charp(const char *val,
        const struct kernel_param *kp) {
        if (strlen(val) > 1024) {
                return -ENOSPC;
        }
        maybe_kfree_parameter(*(char **)kp->arg);

        if (slab_is_available()) {
                *(char **)kp->arg =
                        kmalloc_parameter(strlen(val)+1);
                if (!*(char **)kp->arg) return -ENOMEM;
                strcpy(*(char **)kp->arg, val);
        } else
                *(const char **)kp->arg = val;
        return 0;
}
```

.json

```
[ // (...)
  {
    "member": [ 6, 0 ],
    "kind": "member",
    "offsetrefs": [
      {
        "kind": "parm",
        "mi": 0,
        "id": 1
      }
    ],
    "shift": [ 0, 0 ],
    "access": [ 1, 0 ],
    "csid": 0,
    "expr": "[kernel/params.c:253:34]: kp->arg",
    "ord": [ 29616 ],
    "type": [ 3006, 3017 ]
  },
  // (...)
  {
    "kind": "unary",
    "offsetrefs": [
      {
        "cast": 229,
        "kind": "member",
        "id": 1
      }
    ],
    "expr": "[kernel/params.c:253:24]: *(char **)kp->arg“,
    "csid": 0,
    "offset": 0,
    "ord": [ 29615 ]
  },
  // (...)]
```

Function information

in JSON (cont.)

Extracting variable information used in various expressions inside function body, i.e., indirection operator, array expression, member expression, offsetof, cast, if, switch, loop conditions, return statement, etc.

.c

```c
static int param_array_get(char *buffer, const struct kernel_param *kp)
{
        int i, off, ret;
        const struct kparam_array *arr = kp->arr;
        struct kernel_param p = *kp;

        for (i = off = 0; i < (arr->num ? *arr->num : arr->max); i++)
        {
                /* Replace \n with comma */
                if (i)
                        buffer[off - 1] = ',';
                p.arg = arr->elem + arr->elemsize * i;
                check_kparam_locked(p.mod);
                ret = arr->ops->get(buffer + off, &p);
                if (ret < 0)
                        return ret;
                off += ret;
        }
        buffer[off] = '\0';
        return off;
}
```

.json


```
{
  "kind": "array",
  "offsetrefs":
    [
      { "kind": "parm", "id": 0 },
      { "kind": "local", "id": 3  }
    ],
  "basecnt": 1,
  "expr": "[kernel/params.c:463:4]: buffer[off - 1]",
  "csid": 2,
  "offset": -1,
  "ord":
    [ 31783 ]
},
...
{
  "kind": "logic",
  "offsetrefs":
    [
      { "kind": "local", "id": 4 },
      { "kind": "integer", "id": 0 }
    ],
  "basecnt": 1,
  "expr": "[kernel/params.c:467:7]: ret < 0",
  "csid": 1,
  "offset": 10,
  "ord":
    [ 31797 ]
}
```


Workshop 2022

Function information

in JSON (cont.)

Extracting variable information used in various expressions inside function body, i.e., indirection operator, array expression, member expression, offsetof, cast, if, switch, loop conditions, return statement, etc.

.c

```c
static int param_array_get(char *buffer, const struct kernel_param *kp)
{
	int i, off, ret;
	const struct kparam_array *arr = kp->arr;
	struct kernel_param p = *kp;

	for (i = off = 0; i < (arr->num ? *arr->num : arr->max); i++)
	{
		/* Replace \n with comma */
		if (i)
			buffer[off - 1] = ',';
		p.arg = arr->elem + arr->elemsize * i;
		check_kparam_locked(p.mod);
		ret = arr->ops->get(buffer + off, &p);
		if (ret < 0)
			return ret;
		off += ret;
	}
	buffer[off] = '\0';
	return off;
}
```

.json

```
{
  "kind": "cond",
  "offsetrefs":
   [
    {
      "kind": "logic",
      "id": 30
    }
   ],
  "expr": "[kernel/params.c:460:20]:
                        i < (arr->num ? *arr->num : arr->max)",
...
{
  "kind": "logic",
  "offsetrefs":
   [
    {
      "kind": "local",
      "id": 4
    },
    {
      "kind": "integer",
      "id": 0
    }
   ],
  "expr": "[kernel/params.c:467:7]: ret < 0",
}
```

Workshop 2022

Type information

in JSON

Extracting type attributes that allow to fully identify and reconstruct the original type, i.e., type class, size, attributes, member information, function, global and other types references, etc.

.c

```
struct v4l2_input {
    unsigned int    index;
    unsigned char   name[32];
    unsigned int    type;
    unsigned int    audioset;
    unsigned int    tuner;
    unsigned long   std;
    unsigned int    status;
    unsigned int    capabilities;
    unsigned int    reserved[3];
};
```

.json


```
{
        "id": 0,
        "fid": 0,
        "class": "record",
        "qualifiers": "",
        "size": 640,
        "location": "include/uapi/linux/videodev2.h:1644",
        "union": false,
        "str": "v4l2_input",
        "refs": [ 1, 3, 1, 1, 1, 4, 1, 1, 5 ],
        "refnames": [
        "index","name","type","audioset","tuner",
            "std","status","capabilities","reserved" ],
        "memberoffsets": [ 0,32,288,320,352,384,448,480,512 ],
        "globalrefs": [ ],
        "funrefs": [ ],
}

{
        "id": 1,
        "fid": 0,
        "class": "builtin",
        "qualifiers": "",
        "size": 32,
        "str": "unsigned int",
        "refs": [ ],
}
```

Function ptr initialization

in JSON

Extracting initialization information (function references) of function pointer members of structure types used by the global variables

.c

```c
struct v4l2_file_operations {
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
};

long int video_ioctl2(struct file *file,
              unsigned int cmd, unsigned long int arg);

static const struct v4l2_file_operations dev_fops = {
    .unlocked_ioctl = video_ioctl2,
};
```

.json

```json
{
  "vars": [
    {
      "type": "v4l2_file_operations",
      "name": "uvc_v4l2_fops",
      "members": {
        "0": "video_ioctl2"
      },
      "location": "drivers/usb/gadget/function/uvc_v4l2.c:356"
    }
  ]
}
```

Generating dictionaries for fuzzers

.c

```
enum regex_type filter_parse_regex(char *buff, int len, char **search, int *not) {
	int type = MATCH_FULL;
	int i;

	if (buff[0] == '!') {
		*not = 1; buff++; len--;
	} else { *not = 0; }
	*search = buff;
	if (isdigit(buff[0])) { return MATCH_INDEX; }
	for (i = 0; i < len; i++) {
		if (buff[i] == '*') {
			if (!i) {
				type = MATCH_END_ONLY;
			} else if (i == len - 1) {
				if (type == MATCH_END_ONLY) { type =
						MATCH_MIDDLE_ONLY; }
				else { type = MATCH_FRONT_ONLY; }
				buff[i] = 0;
				break;
			} else { return MATCH_GLOB;	}
		} else if (strchr("[?\\", buff[i])) { return MATCH_GLOB; }
	}
	if (buff[0] == '*') { *search = buff + 1; }
	return type;
}
```

.txt

```
0
1
'!'
'*'
"[?\\"
```

Workshop 2022

Other usage examples

Using CAS in S/W engineering jobs

Examples

Automatically generate structure-aware fuzzing harness for libfuzzer [4]

Overcome the mostly manual work of preparing protobuf descriptions for C structure types

Automatically generate syzkaller definitions

Quick preparation of fuzzing for custom Linux kernel drivers

Finally,

Automatically create test harness code in AoT

AoT: Auto Off Target

GitHub: https://github.com/Samsung/auto_off_target

Workshop 2022

Testing On-Target

An approach to testing S/W of complex embedded systems

Example: testing a message parser in the modem

Testing On-Target

Setup a testing mobile network

Send test messages over the air to the device

When the device crashes capture logs from the device (if any), start offline code analysis

Reboot the device and **repeat**

Difficult test setup for numerous complex embedded systems

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?

Testing On-Target vs Off-Target

Approach to testing S/W of complex embedded systems

Extracting a part of the code to test in on a different host

Testing Off-Target

Prepare the test harness for the parser function

Fuzz the harness on a powerful development machine

Use the available toolchain: gdb, coverage, etc.

BUG FOUND

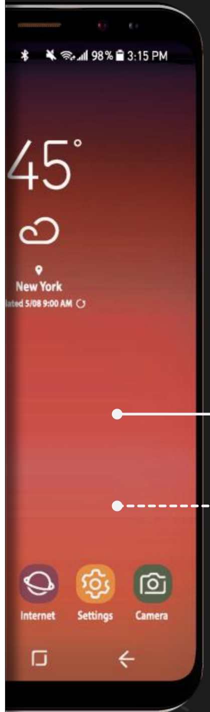
Easier and faster testing in a native development environment

Off-Target testing process

Preparation of the testable Off-Target (OT)

Example: testing a message parser in the modem

On-Target



Off-Target

Step 1:
Extract from mobile

Interesting parts of source code with all required dependencies

Step 2:
Compile and run on a server

Source code testing MANY TIMES FASTER on large servers than on a mobile device

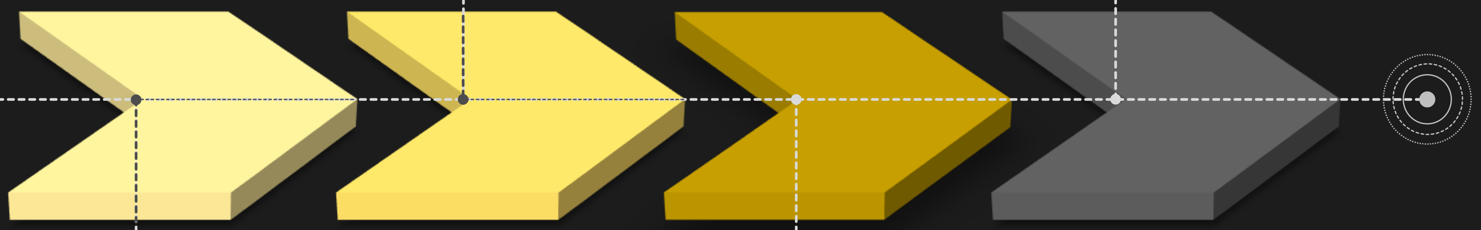
TARGET SOURCE

From a mobile device to a dev machine: might take days of manual labor

Mostly manual process until NOW!

AoT overview

Overview



Automation

Based on the information provided by CAS, AoT automatically creates off-target programs (OTs) in C

Independent

OTs are fully independent from the original build and source tree

Initialization

Additionally, AoT helps to recreate the program state in OTs

Test support

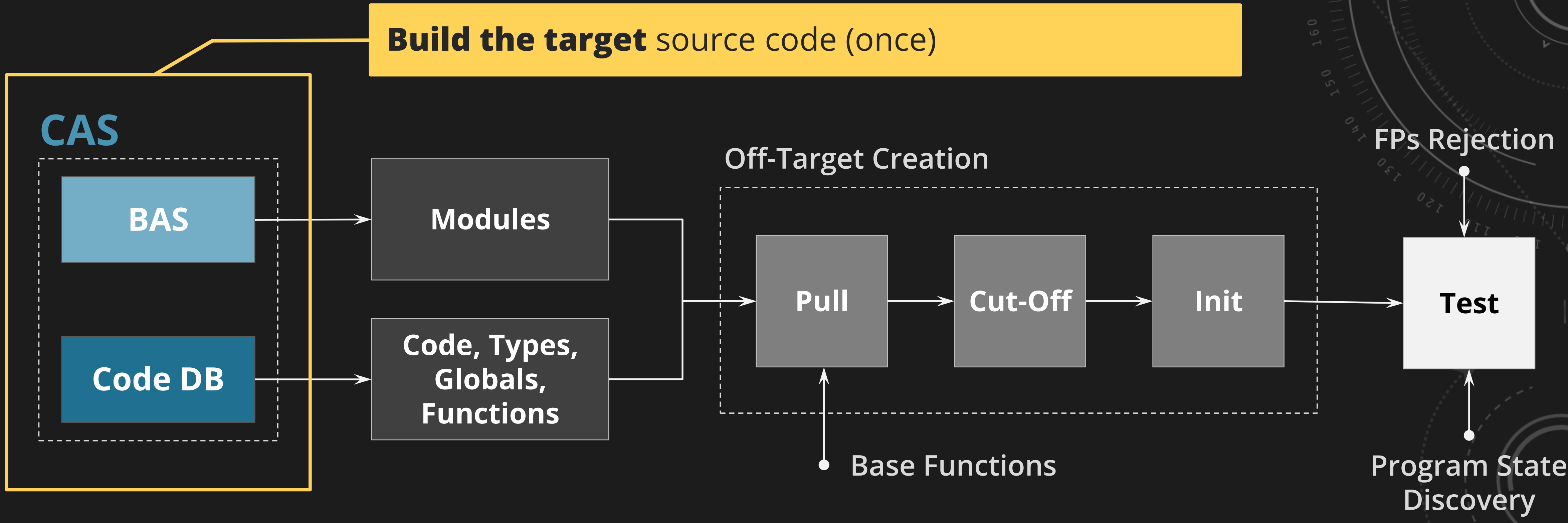
AoT provides support for fuzzing (afl++) and symbolic execution (KLEE)

RESULT

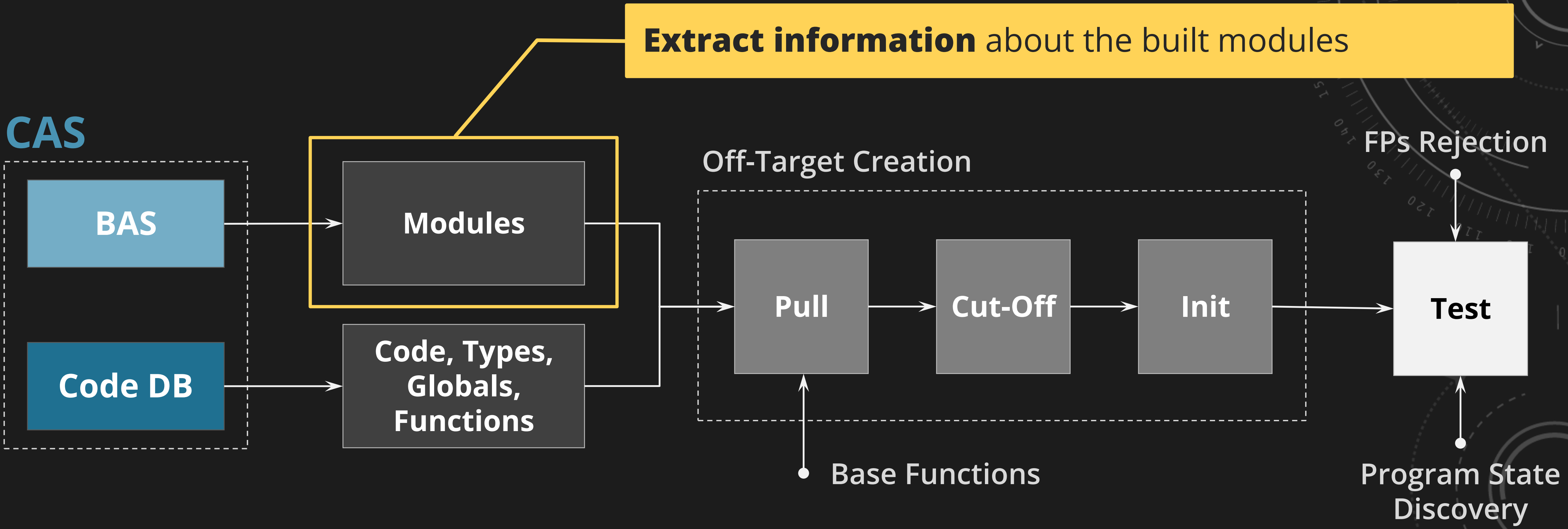
As a result, pieces of complex systems can be thoroughly tested, debugged and analyzed

Importantly, we can now easily run KLEE on these complex targets

AoT implementation

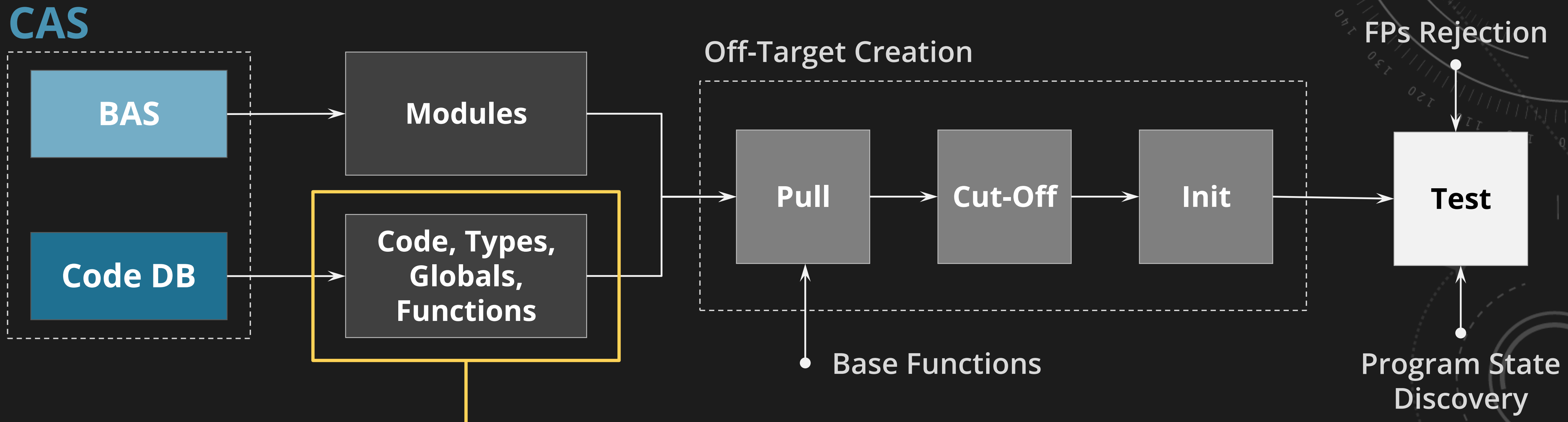


AoT implementation



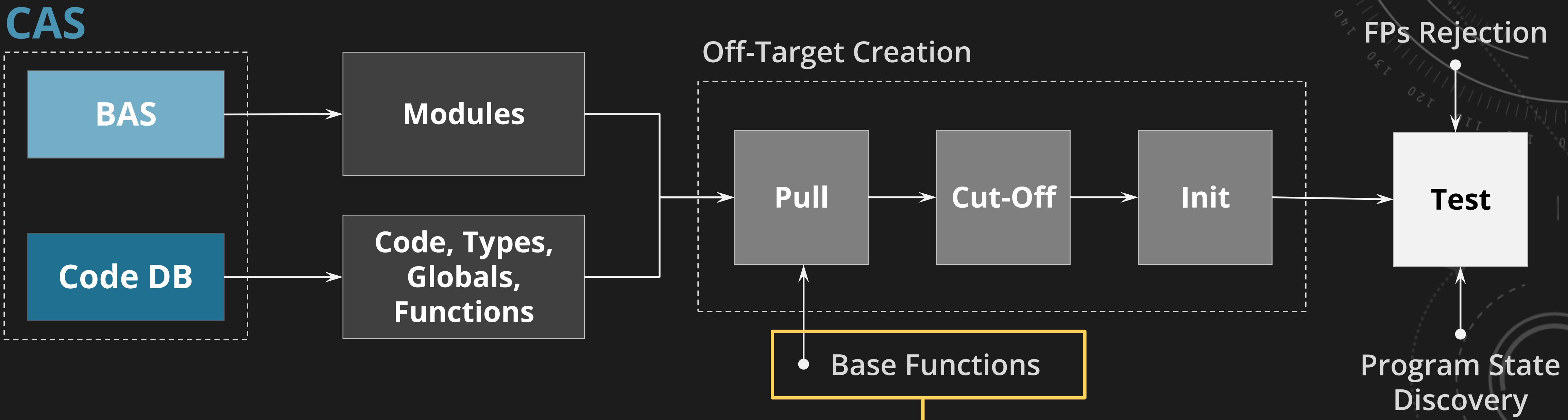
Workshop 2022

AoT implementation



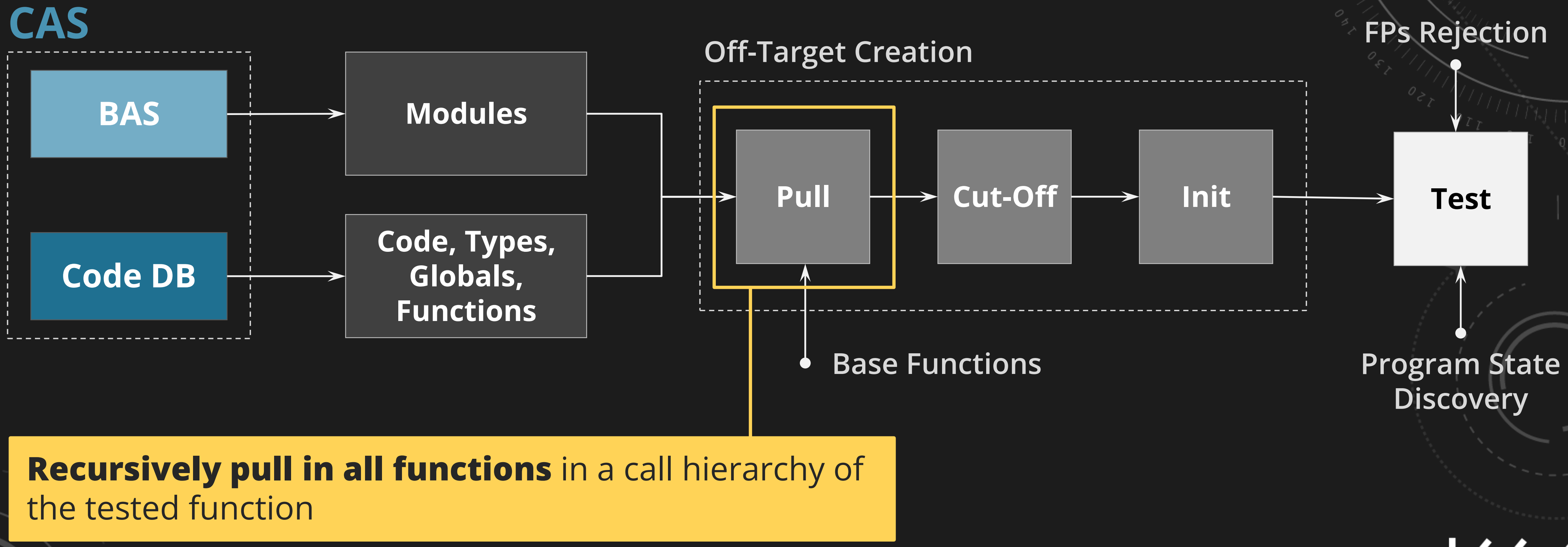
Extract source information, including types, globals, functions, and their dependencies

AoT implementation



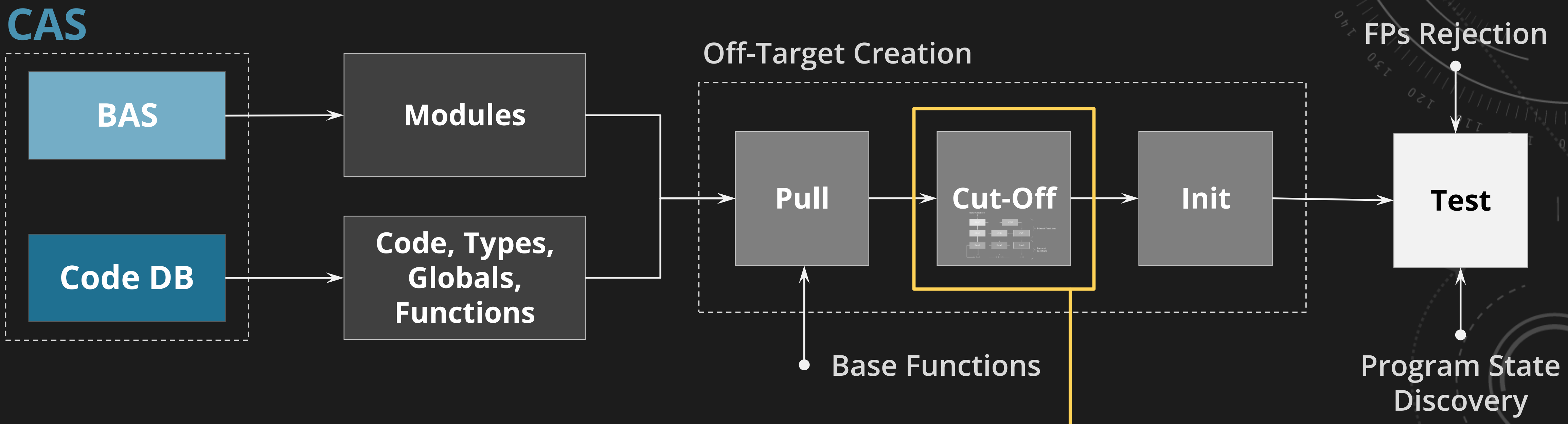
Base functions are the functions we **want to test**

AoT implementation



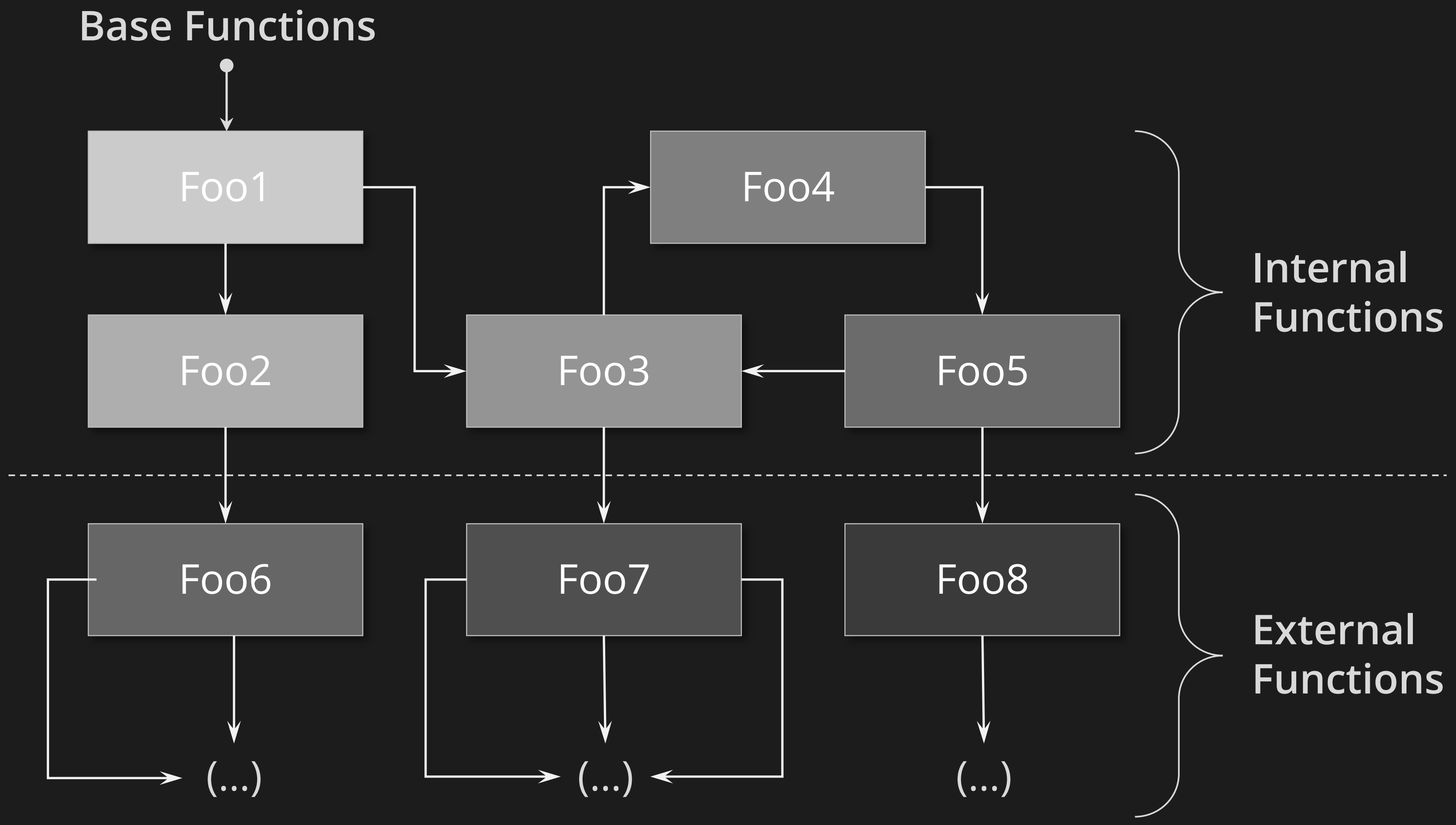
Workshop 2022

AoT implementation

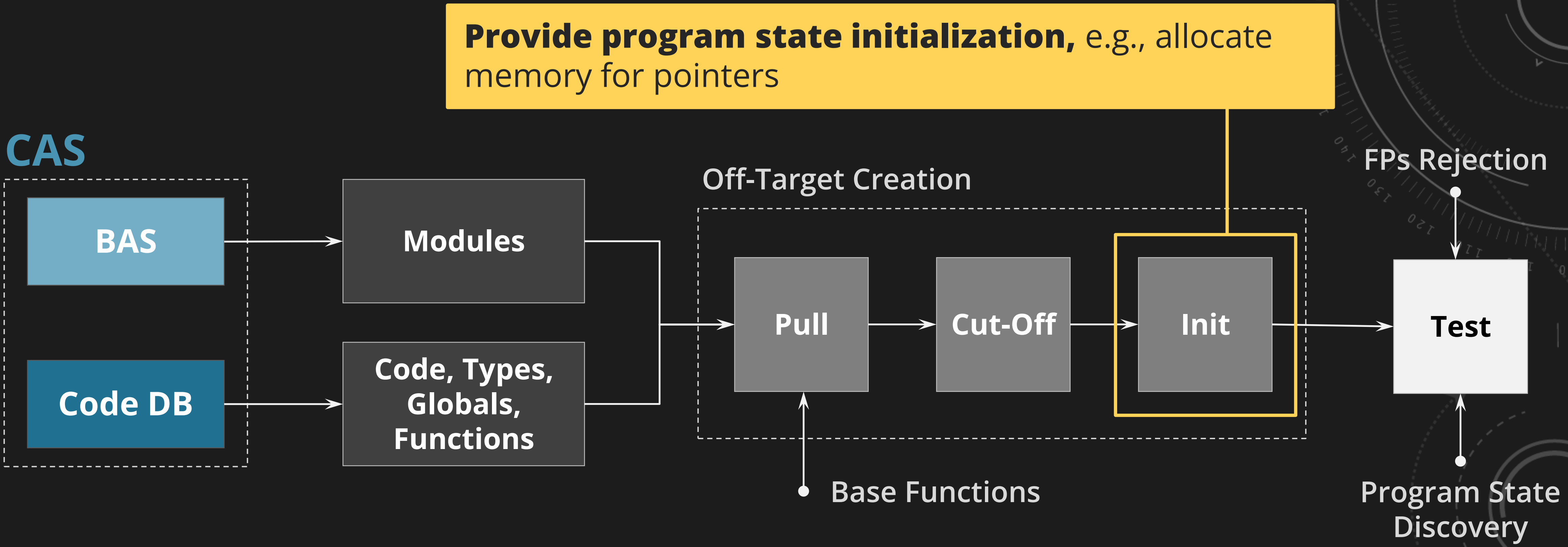


Cut off the code that is outside of the current module
+ generate stubs

AoT implementation: cut-off



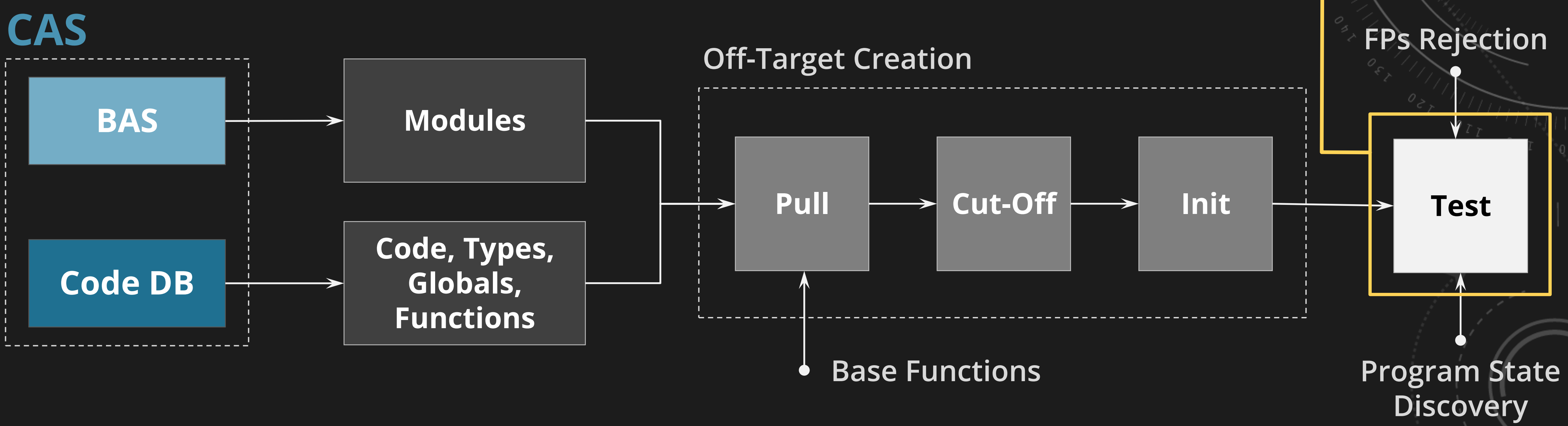
AoT implementation



Workshop 2022

AoT implementation

Apply fuzzing, symbolic execution or other techniques **to test the off-target**



AoT: Smart Init

Initialization of the OT internal state

Failure to allocate memory for “y” results in a crash on line 12

.c

```c
01 #define SIZE 10
02 #define SIZE_A 15
03 globtype_t g[SIZE];
04 rettype_t arr[SIZE_A];
05
06 rettype_t foo ( size_t x, struct B* y ) {
07   if ( x != SIZE ) return 0;
08   for ( int i = 1; i < x; ++i ) {
09     int selector = bar(i) + g[i];
10     if ( selector < SIZE_A ) return arr[selector];
11   }
12   return y->member;
13 }
```

AoT: Smart Init

Initialization of the OT internal state (cont.)

Often, the type to allocate is not known from the function arguments

.c

```c
int foo ( void* x ) {
   struct A* a = ( struct A* )x;
   struct B* b = ( struct B* )a->member1 ;
   struct C* c = b - offsetof( struct C*, member2 );
```

```c
struct A{
   …
   T* member1;
   …
} x;
```

```c
struct C{
   …
   struct B* member2;
   …
} c;
```

Correct allocation involves creating 2 objects and setting a pointer

AoT: Smart Init

Initialization of the OT internal state (cont.)

To discover the types to use, **AoT implements a heuristic based on static analysis** of the information provided by **FTDB**

AoT parses a trace created by 3 code events:

- Casts
- Member accesses
- Use of offsetof

AoT checks in **FTDB** which struct members are used in the OT and initializes only them

AoT: Smart Init

Initialization of the OT internal state (cont.)

The memory allocation is not enough, we need to know the right values

.c

```
#define SIZE 10
#define SIZE_A 15
globtype_t g[SIZE];
rettype_t arr[SIZE_A];

rettype_t foo ( size_t x, struct B* y ) {
  if ( x != SIZE ) return 0;
  for ( int i = 1; i < x; ++i ) {
    int selector = bar(i) + g[i];
    if ( selector < SIZE_A ) return arr[selector];
  }
  return y->member;
}
```

AoT: Program State Discovery

Setting the OT internal state to proper values

- In order to discover usable values of variables, **AoT** uses hybrid fuzzing
- First **AoT** runs **KLEE**, then **AFL++**
- Entire initialized program state in OT is treated as symbolic
- Thanks to symbex, OT code can be run without or with little knowledge on the original system state
- We over approximate: some values will not be allowed on-target
- That results in FPs
- **AoT** implements a mechanism to reject FPs based on **DFSAN**
- We apply taints to user inputs and look for tainted data accesses

  when an issue is detected

AoT evaluation

1. We evaluated AoT on 4 targets:

T1: AOSP kernel for oriole (Pixel 6) -> 50k functions

T2: The Little Kernel Embedded OS -> 1k functions

T3: Das U-Boot bootloader -> 2k functions

T4: The IUH module from Osmocom -> 2k functions

2. Cut-off based on the same compiled module

3. We excluded OT entry points with assembly

Target	*Sample Size*	*No asm*	*Created OTs*	T_{Create}	*LOC*		*Files*	*Types*	*Struct Types*	*Globals*	*Funcs*		*Builds*	T_{Build}
					Pulled	*Total*					*Int*	*Ext*		
T1	50k	48,835	47,970	85.7s	6,041	7,425	15	1,736	347	13	20	14	45,132 (94%)	21s
T2	1k	806	806	0.6s	235	1268	4	35	4	1	6	2	692 (86%)	3s
T3	2k	1,927	1,871	2.3s	410	1469	4	61	10	2	8	2	1,777 (94%)	4s
T4	2k	2,000	2,000	0.9s	284	1605	6	96	8	10	3	14	1,713 (85%)	3s

AoT evaluation

Automated run results

The aim: see on how many OTs we can run symbex and fuzzing out of the box

Target	*Testable* OTs	*Coverage*		*Test cases*	
		Funcs	*Lines*	KLEE	afl++
T1	43,622 (96%)	53.1%	46.4%	2.4	4.5
T2	659 (95%)	75.9%	66.9%	3.8	7.2
T3	1,665 (93%)	73.3%	66.4%	3.5	7.4
T4	1,710 (99%)	98.9%	92.2%	4.5	3.4

AoT evaluation

Human in the loop exercise: a bug finding campaign with AoT

- **AoT** run on **9,396 functions** – kernel entry points from Pixel phones
- After FPs rejection, **312 OTs left** for manual inspection
- Results: **7 bugs found**
- Discovered **3 new security issues**, two were already assigned CVEs
- Rediscovered **CVE-2020-13143**, which was fixed in a newer version
- **Found 3 non-security** out of bound reads

AoT limitations

Where AoT is still imperfect

- Currently **AoT** works with **C code only**
- By default, **code with assembly** is **not included** in OTs
- **Incomplete modelling** of program state can cause FPs
- **Program state initialization** in OTs is an **open** and orthogonal **research problem**

AoT future work

How to make AoT even better

- Implement **support for C++** in **CAS** and **AoT**
- **Improve program state initialization**, e.g., with the use of on-device dumps performed with our tool kflat [3]

Summary

Testing complex embedded systems is both necessary and hard

We presented new projects **CAS** and **AoT** aimed at complex systems:

CAS provides information extracted from the build and from the code
AoT automatically generates usable test harnesses in **C**

CAS provides data useful in dynamic testing research
AoT can be used to run **KLEE** on complex embedded systems code

We demonstrate that AoT is a viable bug finding approach

Summary

Core engines of CAS and AoT and our memory dump tool kflat are open source

[1] **CAS:** https://github.com/samsung/cas

[2] **AoT:** https://github.com/Samsung/auto_off_target

[3] **Kflat:** https://github.com/Samsung/kflat

[4] Bartosz gave a talk on **CAS** at **Linux Security Summit NA 2022**
https://youtu.be/M7gl7MFU_Bc?t=648

[5] We have an upcoming **ASE 2022 paper** on **CAS** & **AoT**:
"Auto Off-Target: Enabling Thorough and Scalable Testing for Complex Software Systems",
https://samsung.github.io/auto_off_target/paper

We welcome contributions,
bug reports and feedback!