Balancing Analysis Time and Bug Detection: Daily Development-friendly Bug Detection in Linux

Keita Suzuki*, Kenta Ishiguro+, Kenji Kono*

*Keio University, +Hosei University

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

sslab system software

Static bug detection in Linux is important

- Many tools has been proposed and proved to be useful
  - **Recommended to use these tools**

Usable from Linux Makefile

- Clang Static Analyzer (CSA)
- Coccinelle [Padioleau+ EuroSys'08]
- CppCheck

- PATA [Li+ ASPLOS '22]
- Saber [Xie+ FSE '05]

And More...

Static bug detection in Linux is important

- Many tools has been proposed and proved to be useful
  - **Recommended to use these tools**

Usable from Linux Makefile

Clang S…
Analyzer

[Padioleau+ EuroSys'08]

CppCheck

…OS '22]

Saber
[Xie+ FSE '05]

And More…

Are these tools used enough in practice?

Are bug detection tools used in practice?

- 40 patches **did not mention any use of tools**
  - Customary to credit the tool if a bug is found with tools
  - Suggests these bugs are found using other methods
    e.g.) Manual inspection by developers

Type	Tool	# of Patches
Not Specified		**40**
Static Analysis	Compiler	8
	Coverity	3
	Clang Static Analyzer (CSA)	1
Dynamic Analaysis	Syzkaller	11
	Abaci Fuzz	1
Total		64

Keyword-based sampling for 6 bug patterns
- Out of bounds, Double free, Use-before-initialization, Integer overflow,
  Nullptr dereference, Reference Counter error
- Patches for Linux v5.9 ~ 5.11

Are bug detection tools used in practice?

- 40 patches **did not mention any use of tools**
  - Customary to credit the tool if a bug is found with tools
  - Suggests these bugs are found using other methods

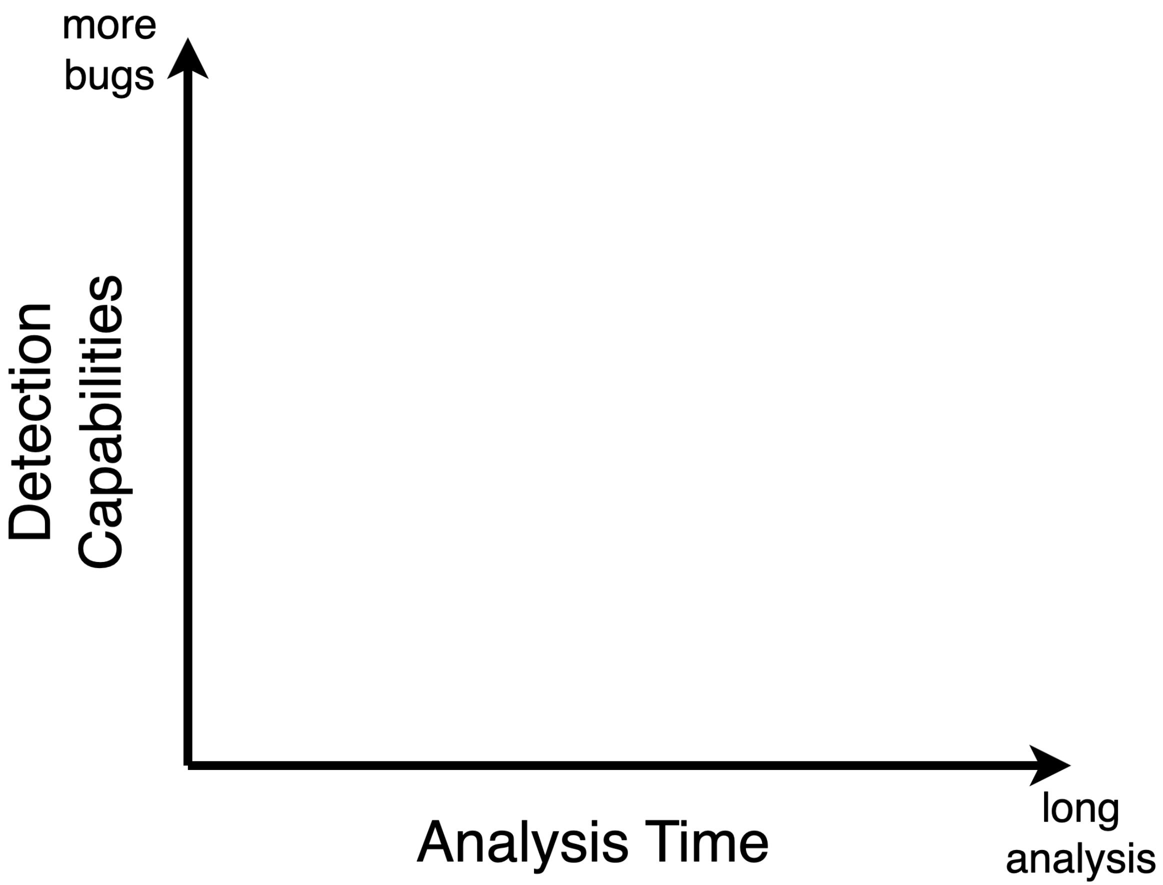
Static bug detection tools
are not used much in daily development!

Static Analysis	Compiler	
	Coverity	3
	Clang Static Analyzer (CSA)	1
Dynamic Analaysis	Syzkaller	11
	Abaci Fuzz	1
Total		64

Keyword-based sampling for 6 bug patterns
- Out of bounds, Double free, Use-before-initialization, Integer overflow, Nullptr dereference, Reference Counter error
- Patches for Linux v5.9 ~ 5.11

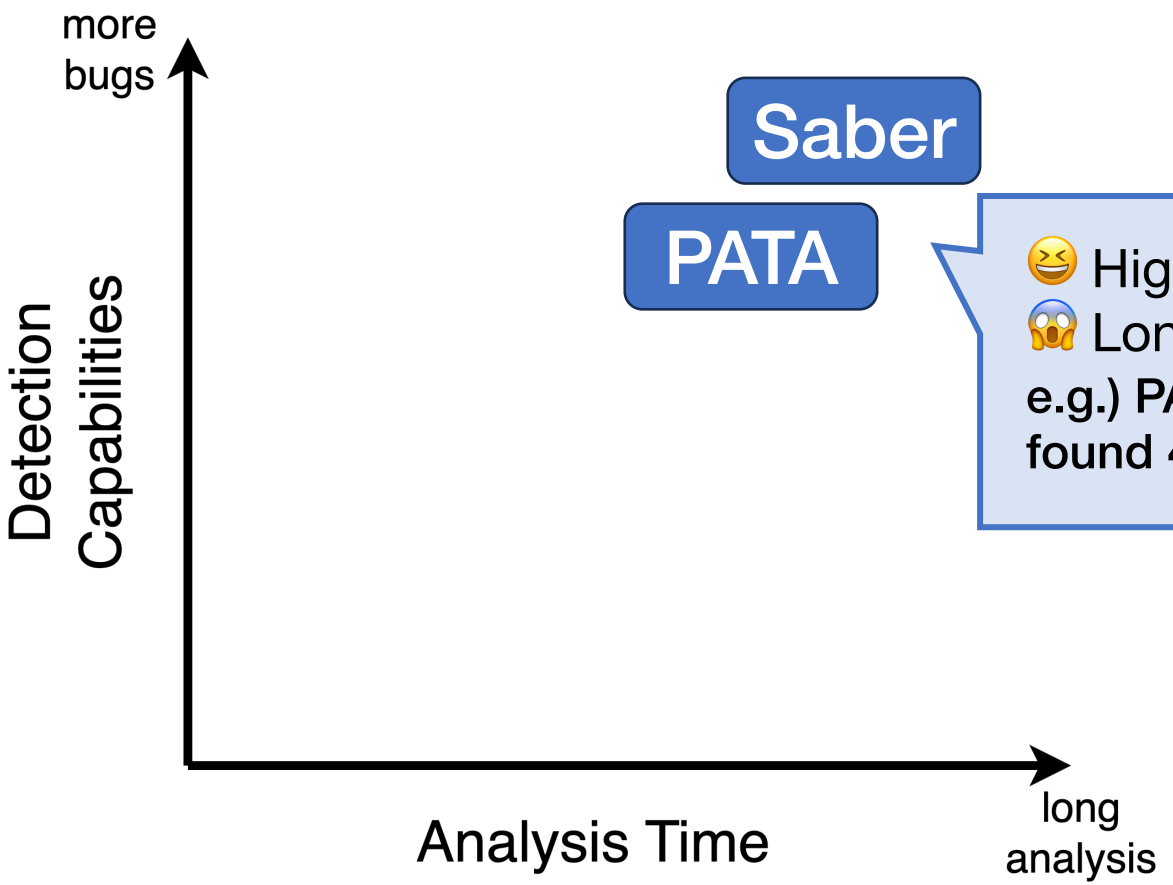
Tradeoff: Analysis time or Detection Capability

- Recent tools typically focus on one end of the tradeoff



Tradeoff: Analysis time or Detection Capability

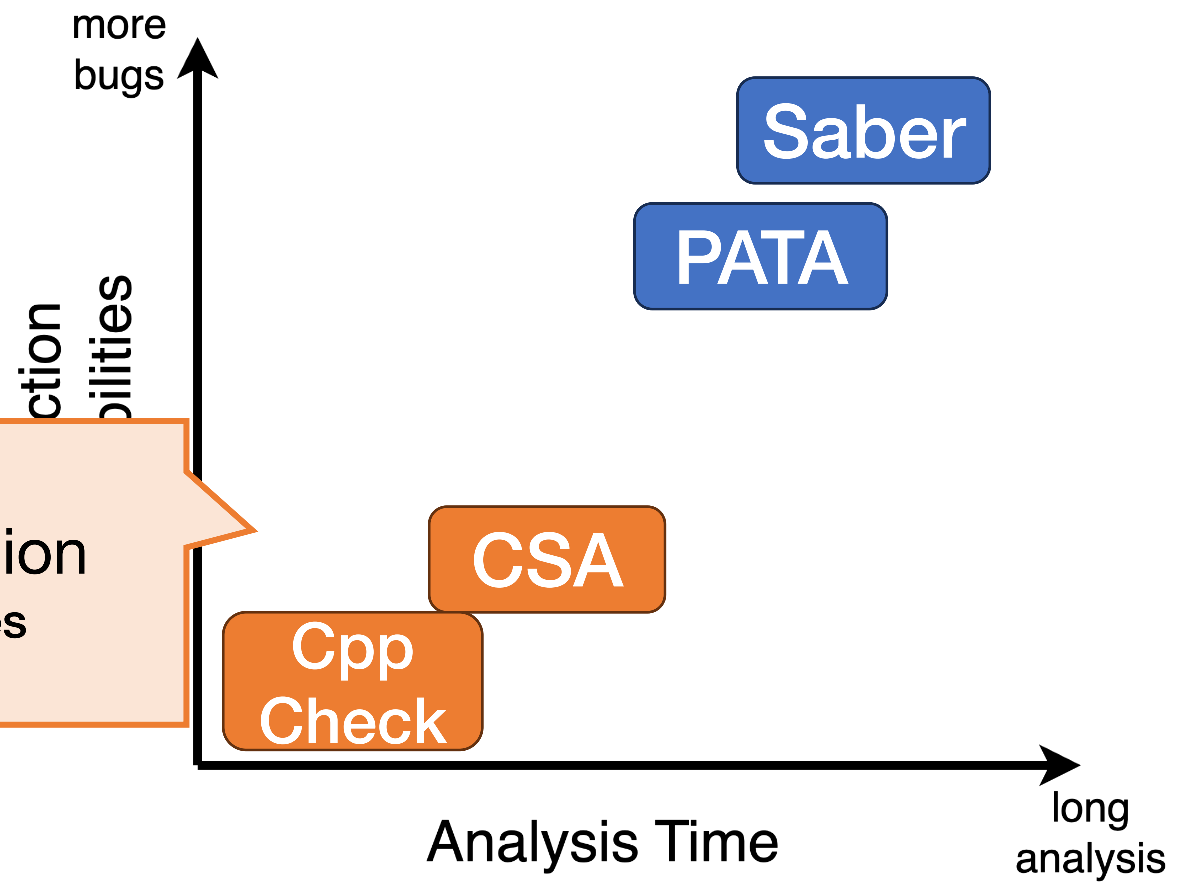
- Recent tools typically focus on one end of the tradeoff



😆 High bug detection capability
😱 Long analysis time
e.g.) PATA [Li+ ASPLOS '22]
found 454 new bugs in 33+ hours

Tradeoff: Analysis time or Detection Capability

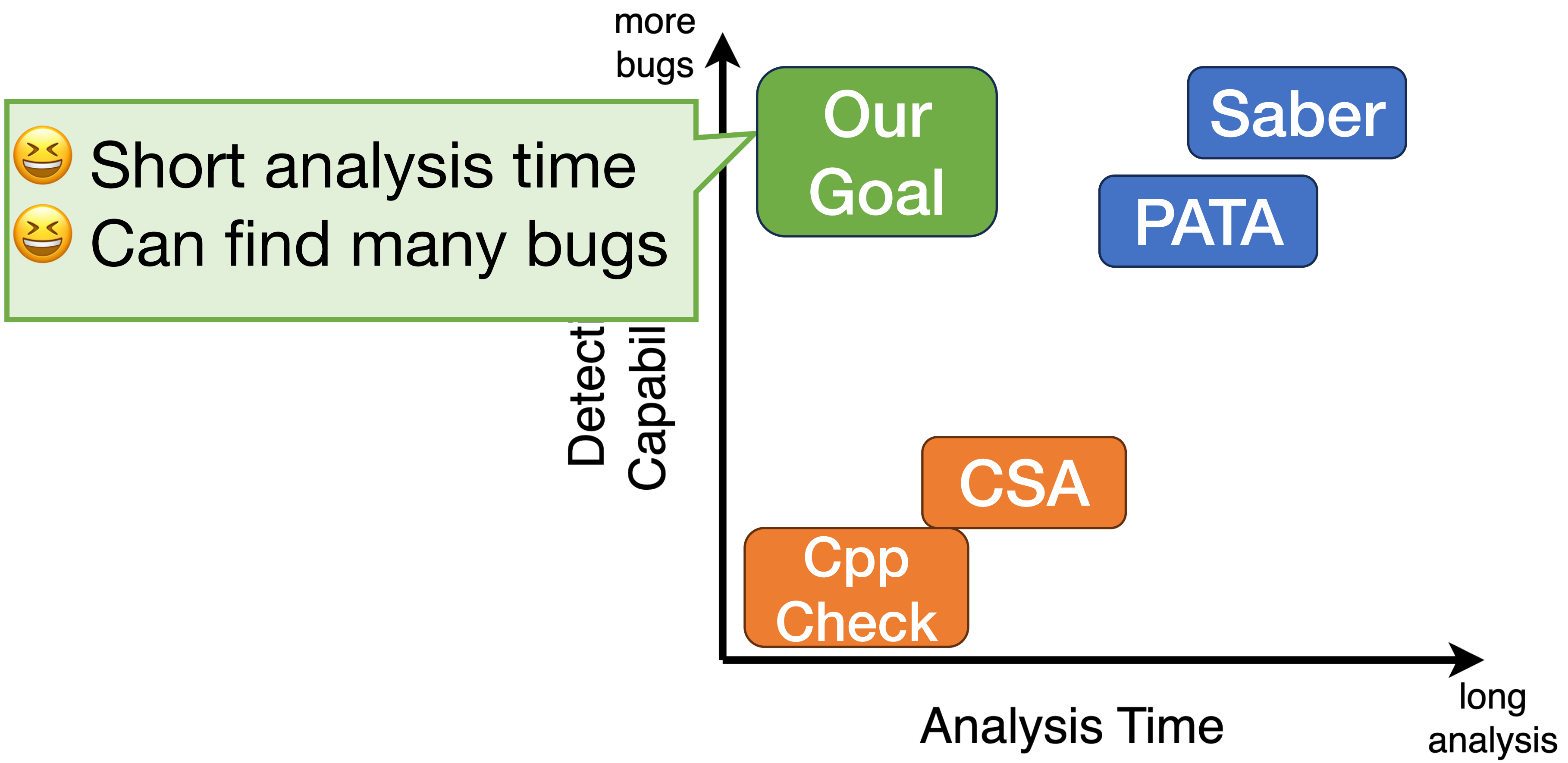
- Recent tools typically focus on one end of the tradeoff



😆Short analysis time
😱 Limited bug detection
e.g.) CppCheck only requires 2.5 hours to analyze

Goal: Daily-development friendly bug detection

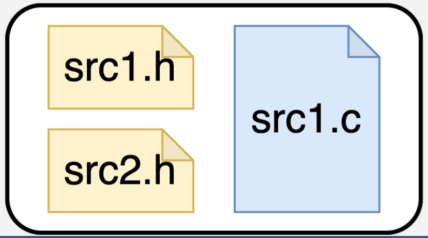
- Explore approach that finds bug while achieving short analysis time
  - Maintains **developer's daily development throughput**



Proposal: Finger Traceable Analysis (FiT Analysis)

- Combination of computationally less complex analysis
  - Focuses on four analysis techniques

1. Analysis of **Single Compilation Unit**



2. Field offset **statically determinable**

```
struct test*s;
func(s->mem);
```

```
tgt = names[1];
```

3. Only requires **simple alias analysis**

```
int* val = s->mem;
...;
free(val);
```

4. **No indirect** function calls

```
struct test *s;
read(s);
```

Are targeting FiT Analysis Bugs impactful?

- Conduct a simple check of Linux bug fixing patches
  - Target 105 patches
  - Investigate its analytical characteristics

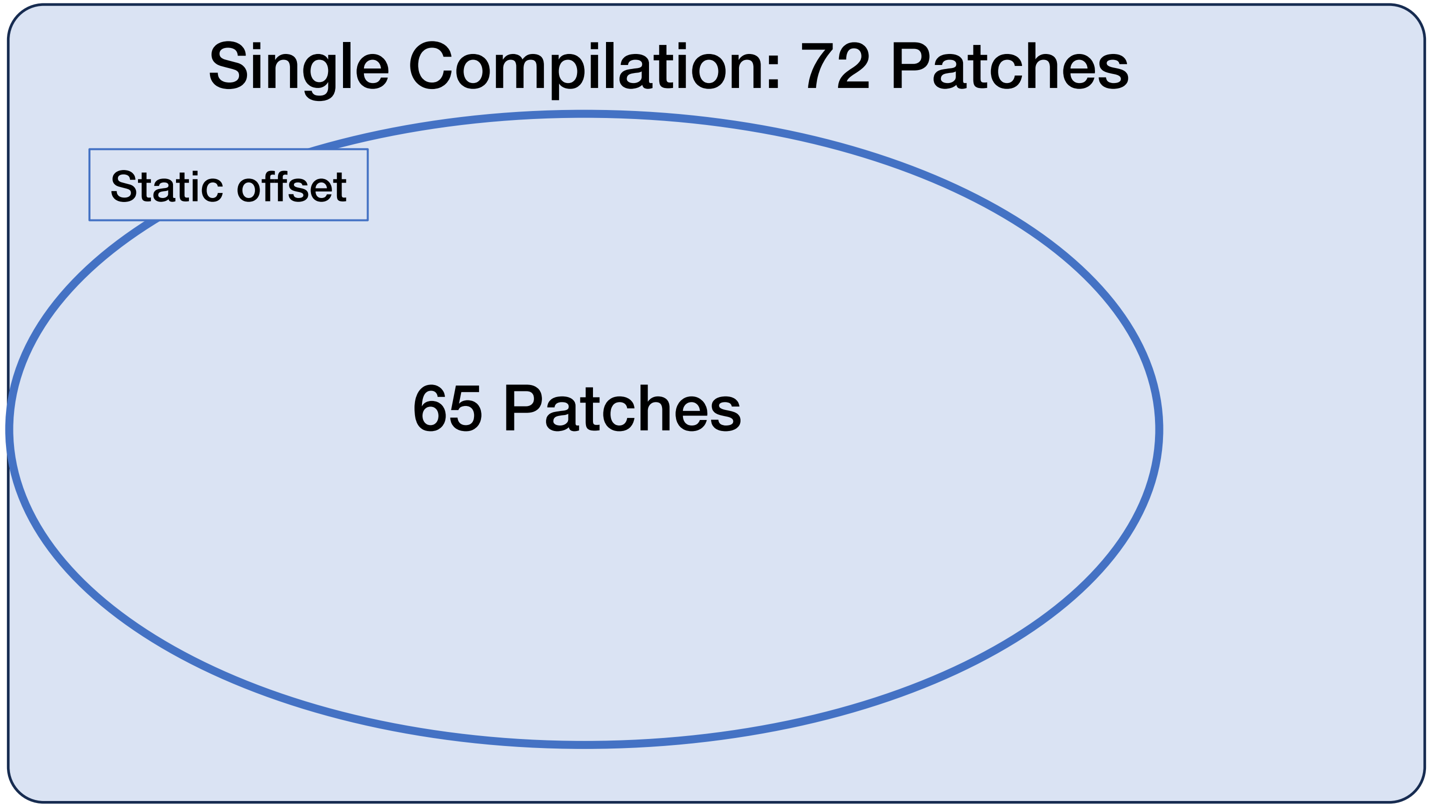
Q1. Single compilation unit?
Q2. Offset calculation static?
Q3. Alias analysis intraprocedural?
Q4. Indirect call involved?

Single Compilation: 72 Patches

Are targeting FiT Analysis Bugs impactful?

- Conduct a simple check of Linux bug fixing patches
  - Target 105 patches
  - Investigate its analytical characteristics

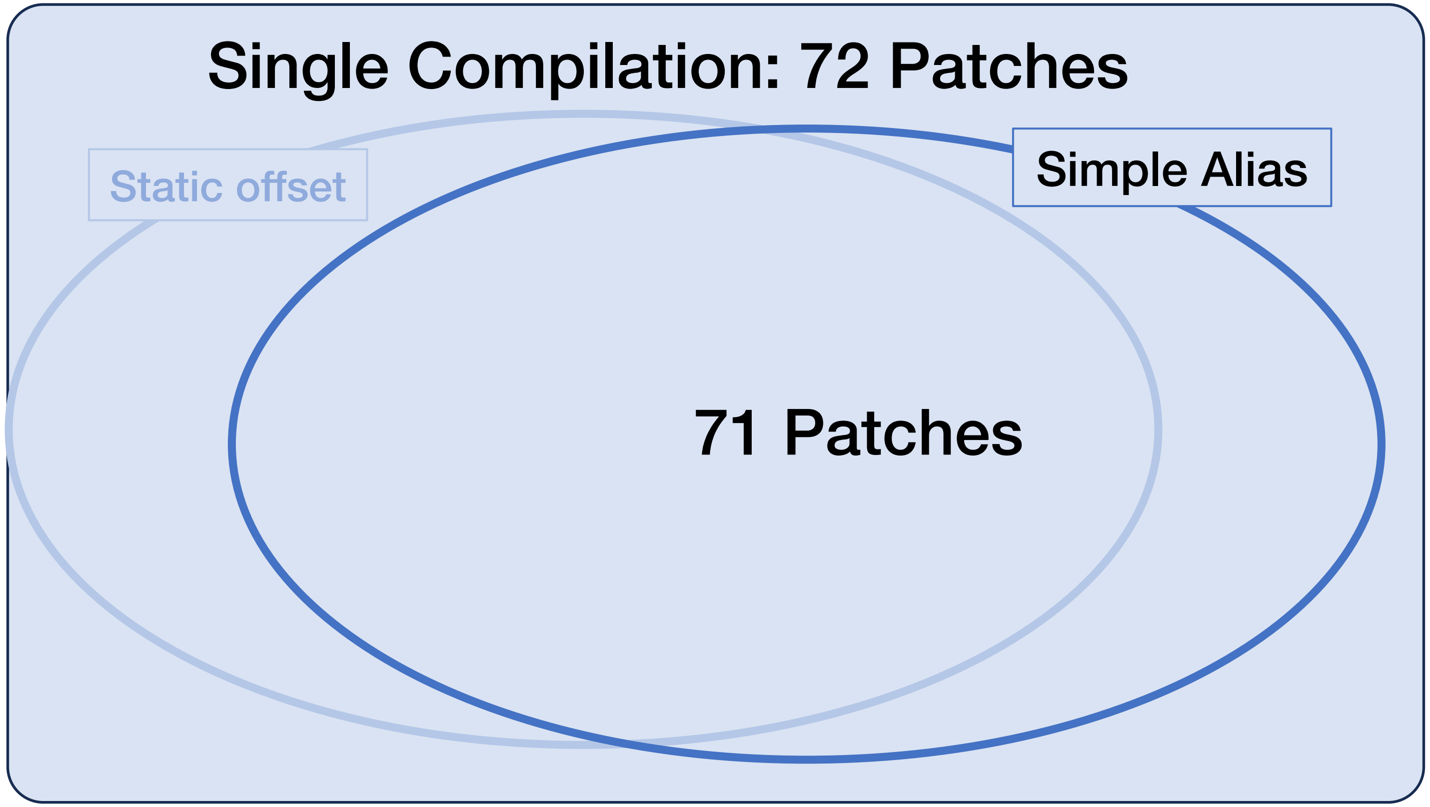
Q1. Single compilation unit?
Q2. Offset calculation static?
Q3. Alias analysis intraprocedural?
Q4. Indirect call involved?



Are targeting FiT Analysis Bugs impactful?

- Conduct a simple check of Linux bug fixing patches
  - Target 105 patches
  - Investigate its analytical characteristics

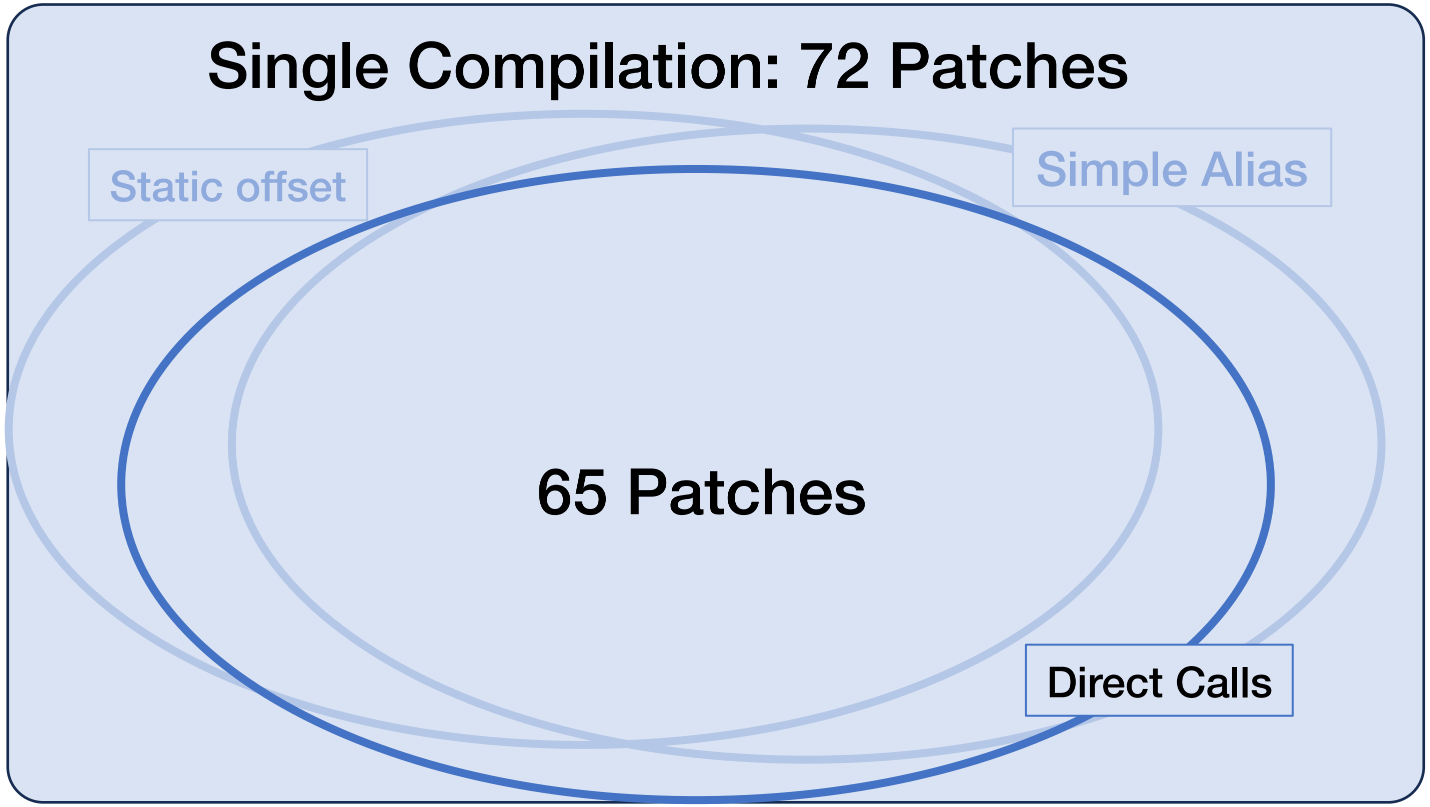
Q1. Single compilation unit?
Q2. Offset calculation static?
Q3. Alias analysis intraprocedural?
Q4. Indirect call involved?



Are targeting FiT Analysis Bugs impactful?

- Conduct a simple check of Linux bug fixing patches
  - Target 105 patches
  - Investigate its analytical characteristics

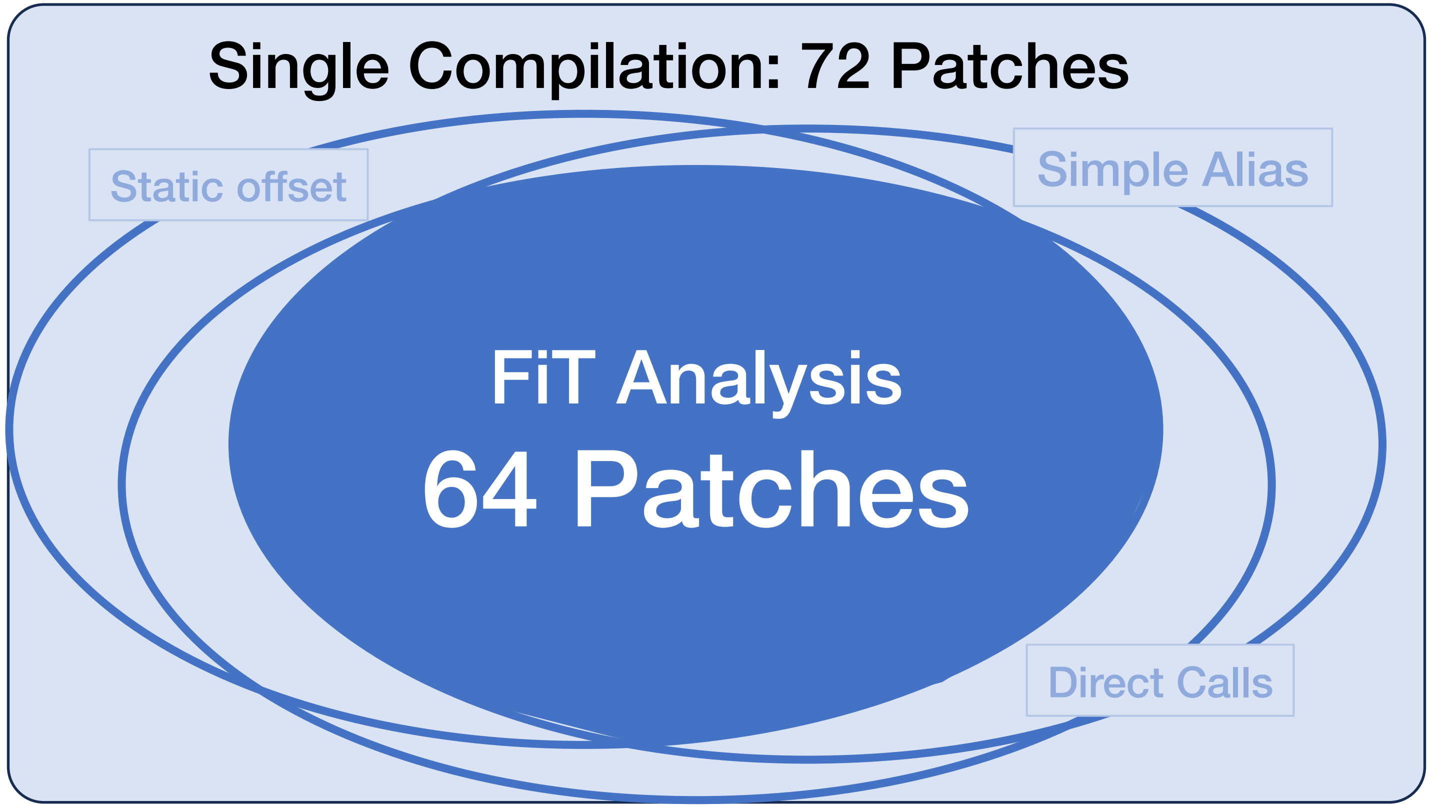
Q1. Single compilation unit?
Q2. Offset calculation static?
Q3. Alias analysis intraprocedural?
Q4. Indirect call involved?



Are targeting FiT Analysis Bugs impactful?

- Conduct a simple check of Linux bug fixing patches
  - Target 105 patches
  - Investigate its analytical characteristics

Q1. Single compilation unit?
Q2. Offset calculation static?
Q3. Alias analysis intraprocedural?
Q4. Indirect call involved?

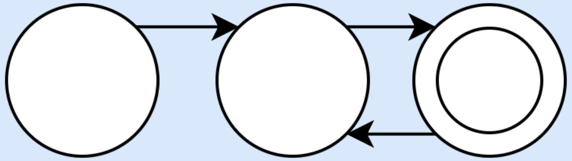


FiTx: FiT Bug Specialized Framework

- Design / Implement a proof-of-concept framework
  - Conduct computationally low-cost dataflow analysis using FiT analysis
  - Define bugs as typestate properties [Strom+ TSE 1986]

Computationally low cost dataflow analysis

✔ Path-insensitive
✔ Field-based
✔ Summary based interprocedural analysis



Defined states of a bug

FiT Analysis

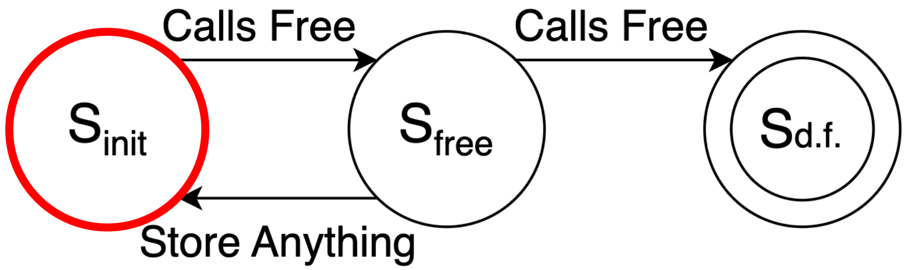
✔ Single compilation unit
✔ Statically determinable fields
✔ Simple alias analysis
✔ No indirect function calls

Introduce
Return code aware interprocedural state propagation

Defining the bug to be detected

- Leverage typestate property analysis [Strom+ TSE 1986]
  - Express each bug using finite state machine
- Collect the state transition per compilation unit

```
kfree(tbl->val);
...;
kfree(tbl->val);
```

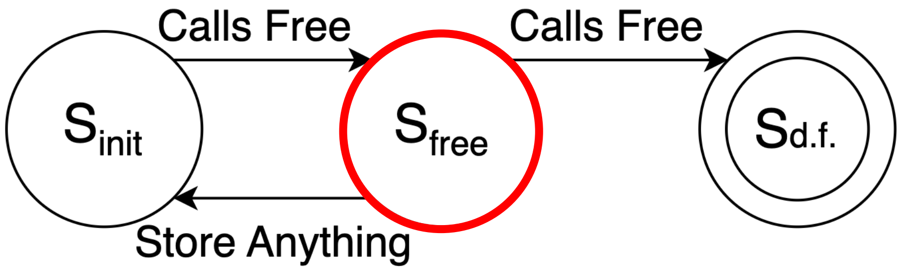


FSM of Double Free

Defining the bug to be detected

- Leverage typestate property analysis [Strom+ TSE 1986]
  - Express each bug using finite state machine
- Collect the state transition per compilation unit

```
kfree(tbl->val);
...;
kfree(tbl->val);
```

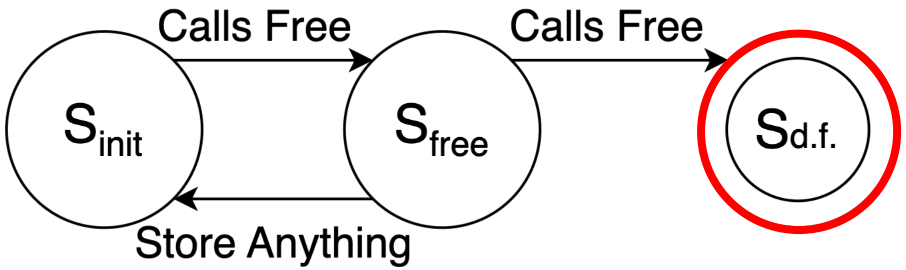


FSM of Double Free

Defining the bug to be detected

- Leverage typestate property analysis [Strom+ TSE 1986]
  - Express each bug using finite state machine
- Collect the state transition per compilation unit

```
kfree(tbl->val);
...;
kfree(tbl->val);
```



FSM of Double Free

Return Code Aware State Propagation (1/2)

- Caller may expect a certain state from callee
  - No consideration **leads to false positive**

```c
int allocate_tbl_fields(struct table *tbl) {
  tbl->val = kmalloc();
  ...;
  if (err) {
    kfree(tbl->val);
    return -ERROR;
  }
  return 0;
}
```

	Return State	
	0	-ERROR
tbl->val	S_{init}	S_{free}

```c
struct table* init_driver() {
  struct table *tbl = kmalloc(...);
  int err = allocate_tbl_fields(tbl);
  if (err) {
    kfree(tbl->val);
    return NULL;
  }
  ...;
  kfree(tbl->val);
}
```

	State
tbl->val	

Return Code Aware State Propagation (1/2)

- Caller may expect a certain state from callee
  - No consideration **leads to false positive**

```
int allocate_tbl_fields(struct table *tbl) {
  tbl->val = kmalloc();
  ...;
  if (err) {
    kfree(tbl->val);
    return -ERROR;
  }
  return 0;
}
```

	Return State	
	0	-ERROR
tbl->val	S_{init}	S_{free}

```
struct table* init_driver() {
  struct table *tbl = kmalloc(...);
  int err = allocate_tbl_fields(tbl);
  if (err) {
    kfree(tbl->val);
    return NULL;
  }
  ...;
  kfree(tbl->val);
}
```

Expects success state

	State
tbl->val	

Return Code Aware State Propagation (1/2)

- Caller may expect a certain state from callee
  - No consideration **leads to false positive**

```
int allocate_tbl_fields(struct table *tbl) {
  tbl->val = kmalloc();
  ...;
  if (err) {
    kfree(tbl->val);
    return -ERROR;
  }
  return 0;
}
```

Pass error state

```
struct table* init_driver() {
  struct table *tbl = kmalloc(...);
  int err = allocate_tbl_fields(tbl);
  if (err) {
    kfree(tbl->val);
    return NULL;
  }
  ...;
  kfree(tbl->val);
}
```

Expects success state

	Return State	
	0	-ERROR
tbl->val	S_{init}	S_{free}

	State
tbl->val	S_{free}

Return Code Aware State Propagation (1/2)

- Caller may expect a certain state from callee
  - No consideration **leads to false positive**

```
int allocate_tbl_fields(struct table *tbl) {
  tbl->val = kmalloc();
  ...;
  if (err) {
    kfree(tbl->val);
    return -ERROR;
  }
  return 0;
}
```

	Return State	
	0	-ERROR
tbl->val	S_{init}	S_{free}

```
struct table* init_driver() {
  struct table *tbl = kmalloc(...);
  int err = allocate_tbl_fields(tbl);
  if (err) {
    kfree(tbl->val);
    return NULL;
  }
  ...;
  kfree(tbl->val);
}
```

Expects success state

	State
tbl->val	$S_{d.f.}$

False Positive!

Return Code Aware State Propagation (2/2)

- Propagate states together **with the return code**
  - **Focus on constant return codes** such as error codes
    - Use Linux return convention

```
int allocate_tbl_fields(struct table *tbl) {
  tbl->val = kmalloc();
  ...;
  if (err) {
    kfree(tbl->val);
    return -ERROR;
  }
  return 0;
}
```

	Return State	
	0	-ERROR
tbl->val	S_{init}	S_{free}

```
 1 struct table* init_driver() {
 2   struct table *tbl = kmalloc(...);
 3   int err = allocate_tb
 4   if (err) {
 5     kfree(tbl->val);
 6     return NULL;
 7   }
 8   ...;
 9   kfree(tbl->val);
10 }
```

Check error code usage

		State
tbl->val	line 4	
	line 8	

Return Code Aware State Propagation (2/2)

- Propagate states together **with the return code**
  - **Focus on constant return codes** such as error codes
    - Use Linux return convention

```c
int allocate_tbl_fields(struct table *tbl) {
  tbl->val = kmalloc();
  ...;
  if (err) {
    kfree(tbl->val);
    return -ERROR;
  }
  return 0;
}
```

	Return State	
	0	-ERROR
tbl->val	S_{init}	S_{free}

```c
1  struct table* init_driver() {
2    struct table *tbl = kmalloc(...);
3    int err = allocate_tbl_fields(tbl);
4    if (err) {
5      kfree(tbl->val);
6      return NULL;
7    }
8    ...;
9    kfree(tbl->val);
10 }
```

Expects error state

		State
tbl->val	line 4	S_{free}
	line 8	

Return Code Aware State Propagation (2/2)

- Propagate states together **with the return code**
  - **Focus on constant return codes** such as error codes
    - Use Linux return convention

```
int allocate_tbl_fields(struct table *tbl) {
  tbl->val = kmalloc();
  ...;
  if (err) {
    kfree(tbl->val);
    return -ERROR;
  }
  return 0;
}
```

	Return State	
	0	-ERROR
tbl->val	S_{init}	S_{free}

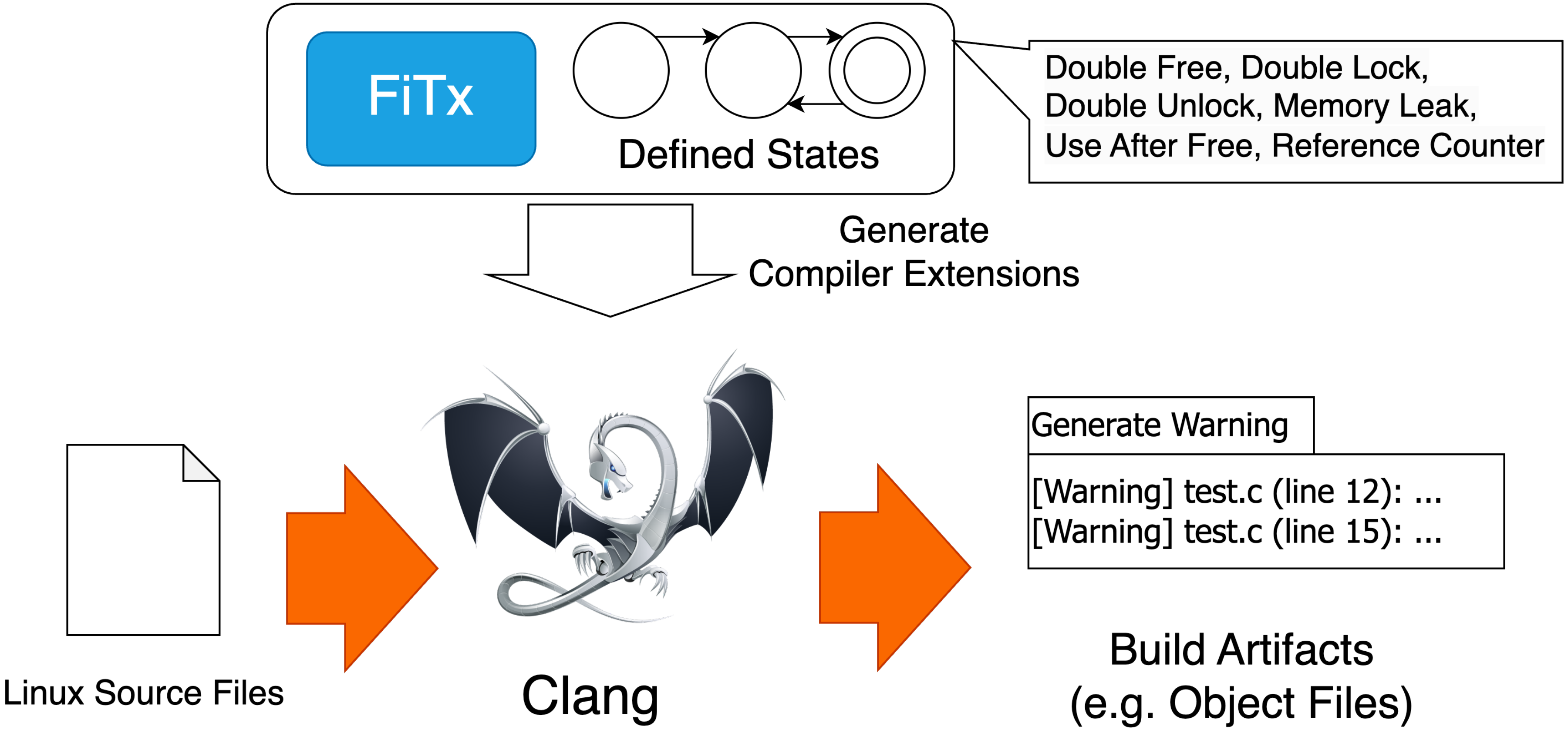
```
1  struct table* init_driver() {
2    struct table *tbl = kmalloc(...);
3    int err = allocate_tbl_fields(tbl);
4    if (err) {
5      kfree(tbl->val);
6      return NULL;
7    }
8    ...;
9    kfree(tbl->val);
10 }
```

Expects success state

		State
tbl->val	line 4	S_{free}
	line 8	S_{init}

Implementation

- Use LLVM compiler framework to implement the framework
  - Implemented checkers for 6 bug patterns



Evaluation

1. What is FiTx's bug detection capabilities?
2. How long does FiTx take to analyze the Linux?
3. How does FiTx perform compared to other tools?
    - Compare with Clang Static Analyzer (CSA) and CppCheck

OS	Ubuntu 20.04
CPU	16 Core Intel Xeon CPU E5-2620
RAM	96 GB (limited to 32 GB)
LLVM	10.0.1
Target Kernel	v5.15
Config	**allyesconfig**

Evaluation Environment

What is FiTx's bug detection capabilities?

- FiTx was able to find 47 new bugs
  - 13 of them confirmed / fixed by developers

Bug Type	Warnings	TPs
Double Free	41	21
Use After Free	31	9
Double Lock	16	7
Double Unlock	13	5
Memory Leak	15	3
Reference Counter	5	2
Total	**121**	**47**

Data updated from original paper

Found Bug: Double free in AMD GPU driver

- Confirmed / fixed by developers
  - **Existed from 2016**

drivers/gpu/drm/amd/pm/legacy-dpm/si_dpm.c

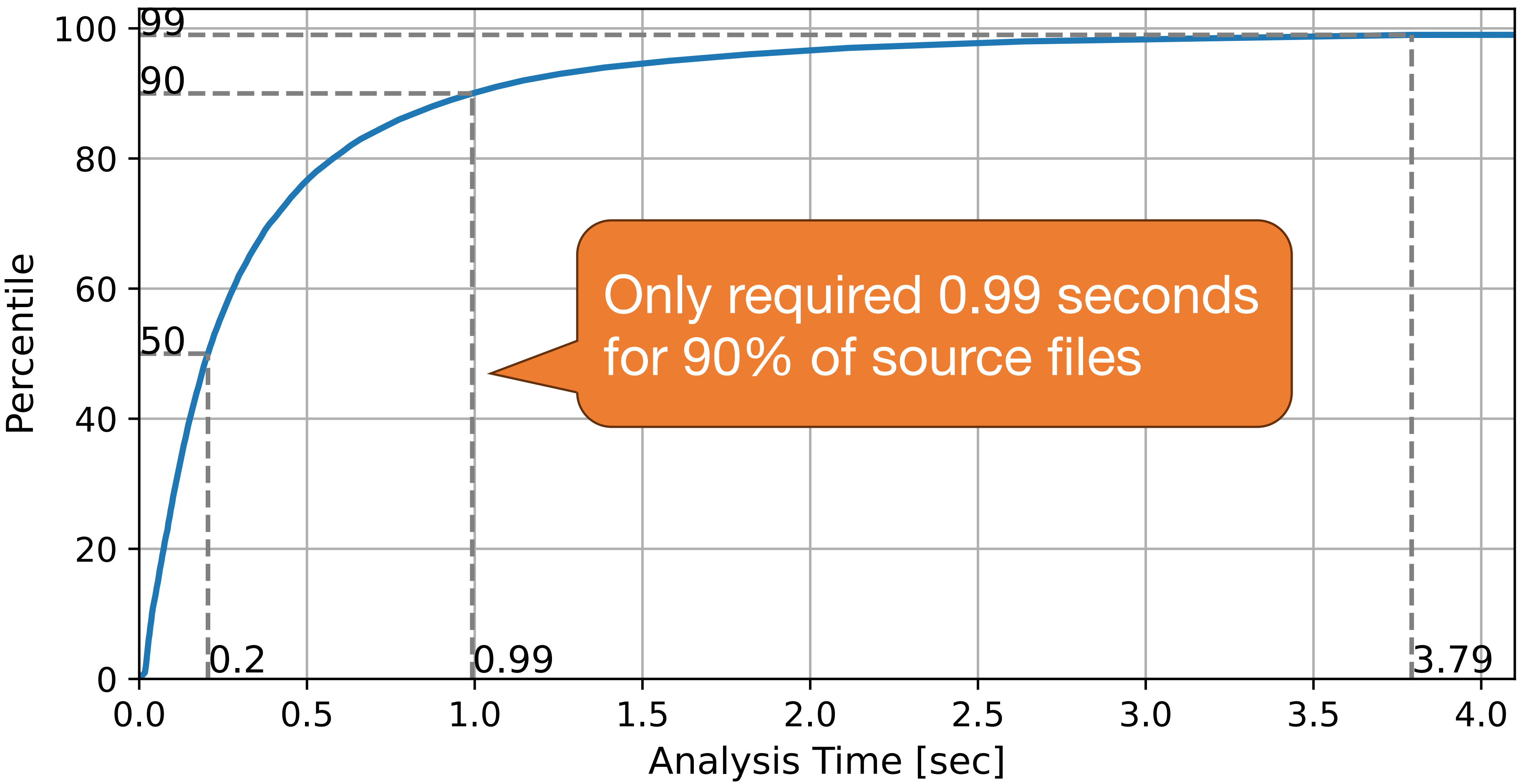
```
int si_dpm_sw_init(void *handle) {
  ...;
  ret = si_parse_power_table(adev);
  if (ret)
    si_dpm_fini(adev);
}
```

```
int si_parse_power_table(struct amdgpu_device *adev) {
  for (int i = 0; i < num_entries; i++) {
    ps = kzalloc(...);
    if (ps == NULL) {
      kfree(adev->ps);
      return -ENOMEM;
    }
    adev->ps[i].ps_priv = ps;
  }
  return 0;
}
```

```
void si_dpm_fini(struct amdgpu_device *adev) {
  ...;
  kfree(adev->ps);
}
```

How long does FiTx take to analyze Linux?

- Took 2hr 33 min in total to analyze Linux



CDF of Analysis Time per source file

How does FiTx perform compared to other tools?

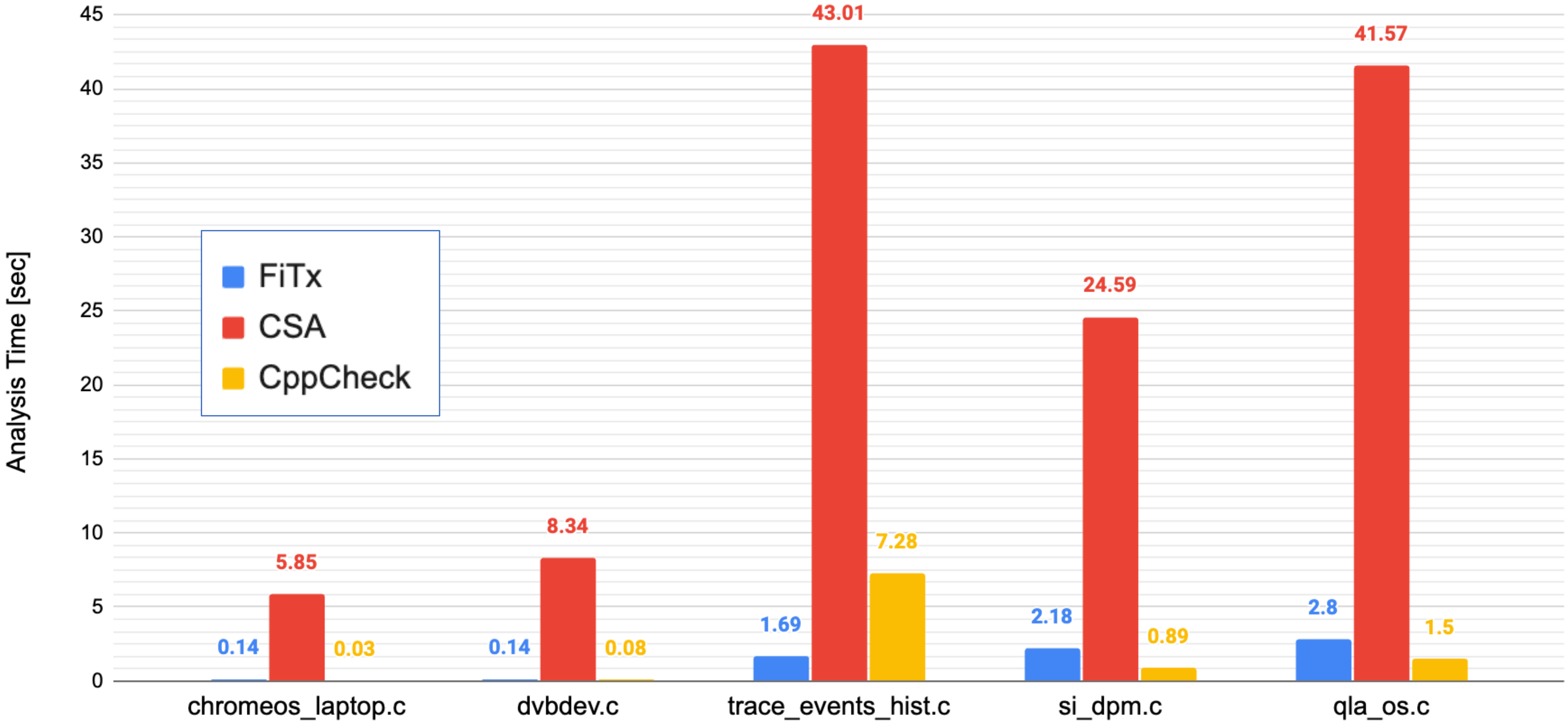
- Compare with Clang Static Analyzer (10.0.1) and CppCheck (1.9)
  - **What is the analysis time?**
  - **Can the tools find the bugs?**
  - **What is the false positive rate?**

Analyze developer confirmed bugs (analysis time; finding the bugs)

Analyze entire Linux (false positive rate)

Source File	LoC	# of Bugs
drivers/platform/ chrome/chromeos_laptop.c	958	2
drivers/media/dvb-core/dvbdev.c	1,084	1
kernel/trace/trace_events_hist.c	6,113	6
drivers/gpu/drm/amd/pm/ powerplay/si_dpm.c	7,127	2
drivers/scsi/qla2xxx/qla_os.c	8,216	2
Total		**13**

Comparison: Analysis Time



Comparison: Bug detection capabilities

- CSA and CppCheck did not find the bugs

Source File	FiTx	CSA	CppCheck
drivers/platform/ chrome/chromeos_laptop.c	2	0	0
drivers/media/dvb-core/dvbdev.c	1	0	0
kernel/trace/trace_events_hist.c	6	0	0
drivers/gpu/drm/amd/pm/ powerplay/si_dpm.c	2	0	0
drivers/scsi/qla2xxx/qla_os.c	2	0	0
Total	**13**	**0**	**0**

Comparison: False Positives

- Compare the false positive rate when analyzing entire Linux
- FiTx generates less false positive compared to CSA and CppCheck

	FP Rate
FiTx	61.2 %
CppCheck	83.4%*
CSA	83.0%*

FP rate when analyzing entire Linux

*Reported in [Li+ ASPLOS '22]

Summary

- FiTx's Goal: Daily development friendly bug detection
  - **Combination of four low computational analysis**
- Found 47 new bugs in Linux kernel version 5.15
  - **13 bugs confirmed by developers**
  - **0.99 sec of analysis time for 90% of source file**
  - **Outperformed CSA / CppCheck**

Thank you!

Artifact here!
https://github.com/sslab-keio/FiTx

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED