



# **QSYM: A PRACTICAL CONCOLIC EXECUTION ENGINE TAILORED FOR HYBRID FUZZING**

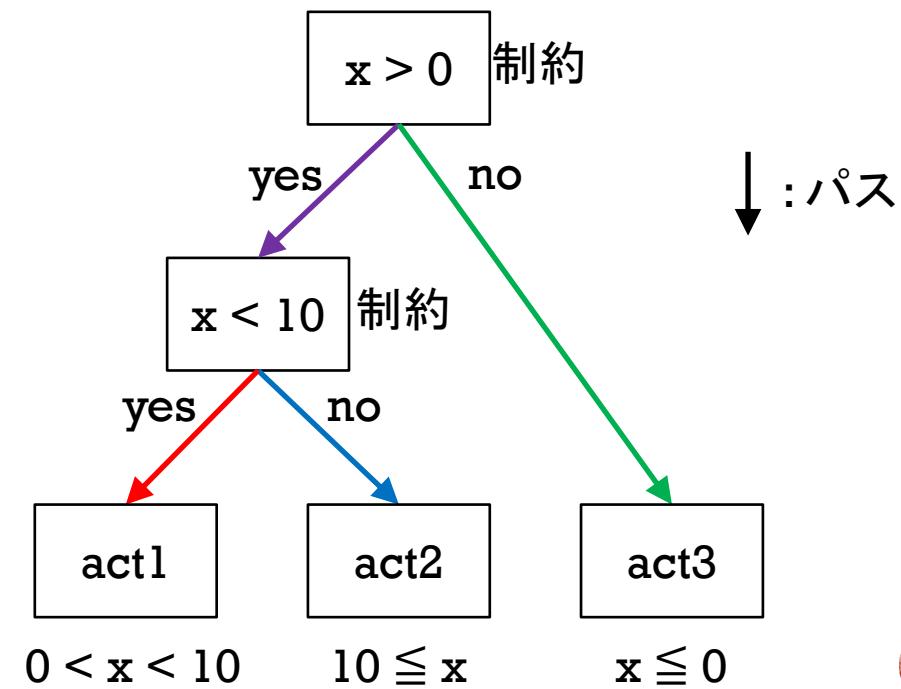
**INSU YUN ET. AL.  
USENIX SECURITY 2018**

# 予備知識（用語定義）

- 実行パス(path)
  - プログラムが一回の実行で通る路のこと
- 制約(constraint)
  - (本論文上では)特定のパスを通るための条件のこと

```
int main(){
    int x;
    x = input();

    if(x > 0){
        if(x < 10){ action1(); }
        else{ action2(); }
    }else{
        action3();
    }
}
```



# ABSTRACT

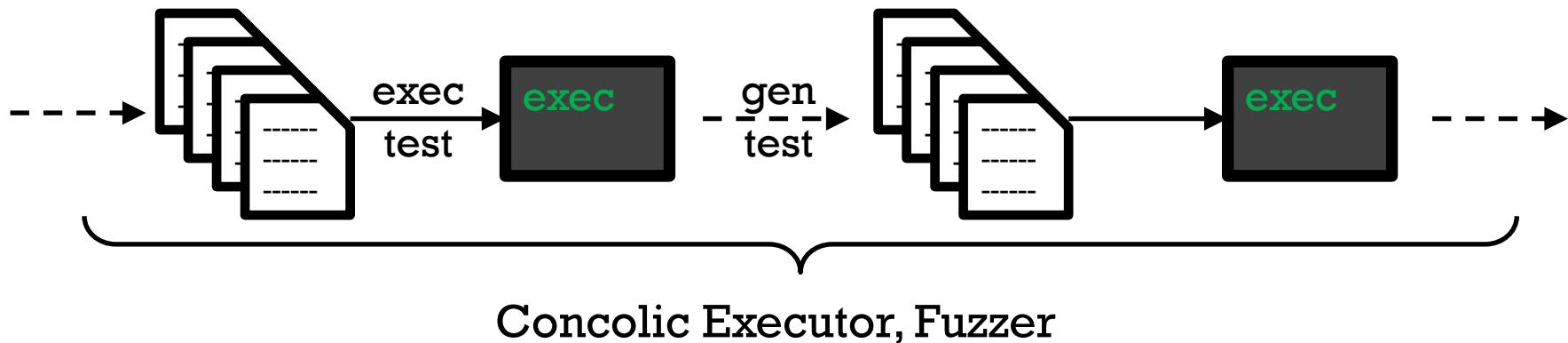
---

- **Hybrid Fuzzing**は現実の複雑なプログラムの検査が困難
  - Hybrid Fuzzing = **Concolic Execution + Fuzzing**
    - Concolic = Concrete + Symbolic
    - 理由①: 遅い
    - 理由②: システムコール／外部環境をサポートできない
    - 理由③: 複雑な制約を解決できない
- 提案手法: **QSYM** (Concolic Executor) with AFL
  - ① → IR(中間表現)の排除、命令粒度の最適化、制約解決の最小化
  - ② → 再実行の高速化、必要最低限のモデル化、ブラックボックス化
  - ③ → ヒューリスティクス(Optimistic Solving, BBL pruning)
- 実験結果
  - 既存技術では発見できなかった**未知のバグを13件発見**
  - 既存手法 (Driller, VUzzerなど) よりも**高いカバレッジを達成**

# INTRODUCTION: そもそも

- **Concolic Execution**とか**Fuzzing**って何？

- プログラムを **大量にテストして** 脆弱性を見つける手法
- バグを引き起こしそうなテストを自動生成、実行するフレームワーク



自動テスト生成 & 実行  
バグが見つかれば、その入力と合わせて報告

# INTRODUCTION: FUZZING

- **Fuzzing:**

- 脆弱性を引き起こしそうな入力(fuzz)に対する挙動を監視して脆弱性を発見を試みる手法 (by IPA\*)



- 研究分野としてのFuzzing

- 入力を自動生成し、より多くのパスを実行/テストする
  - Pure (Random) Fuzzing, Coverage-guided Fuzzing, ...
- オープンソースで有名なのはAFL[1]とかOSS-Fuzz[3]
- 利点: 緩い制約や、一般的なパターンは探索できる (かつ早い)
- 欠点: 厳しい制約 (key == 0xFDなど) は突破が困難

[1] “american fuzzy lop,” <http://lcamtuf.coredump.cx/afl/> [2015]

[3] “OSS-Fuzz - continuous fuzzing of open source software,” <https://github.com/google/oss-fuzz> [2016]

\* IPA “脆弱性対策: ファジング” <https://www.ipa.go.jp/security/vuln/fuzzing.html>

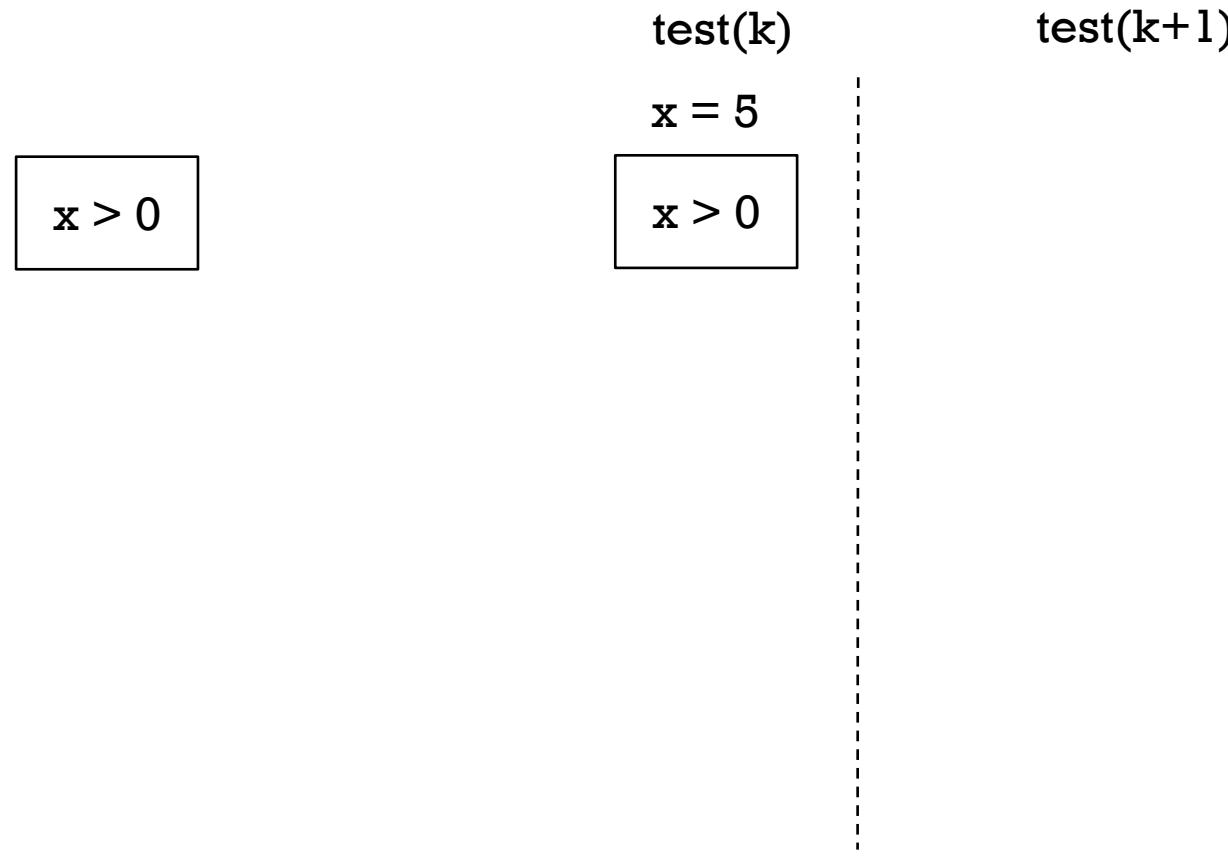
# INTRODUCTION: CONCOLIC EXECUTION

---

- Concolic = Concrete(具体的な) + Symbolic(シンボリック)
- **Symbolic Execution**(記号実行)
  - 外部入力を記号(シンボル)として扱い、記号の持つ制約と到達するパスをシミュレート
  - 結果から別の制約を持つパスを探し、再シミュレート
- **Concolic Execution**(コンコリック実行)
  - 外部入力を**実際に**プログラムに入力し 入力が持つ制約と到達するパスを検査
  - 検査結果から別の制約を持つパスを探し、再実行 (or ロールバック)
- 利点: 狹い制約も突破可能  
実値を入れることで、Symbolicよりも制約が解きやすくなる
- 欠点: パスが爆発する(Symbolicも同様)  
結果、重くなりがち

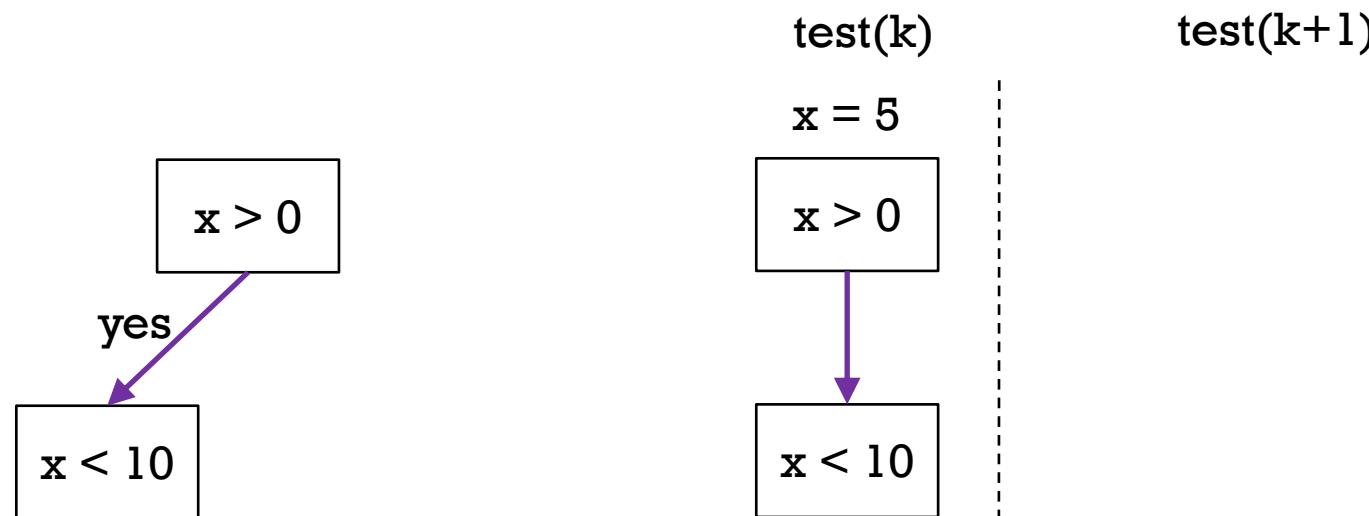
# INTRODUCTION: CONCOLIC EXECUTION

- こんなイメージ



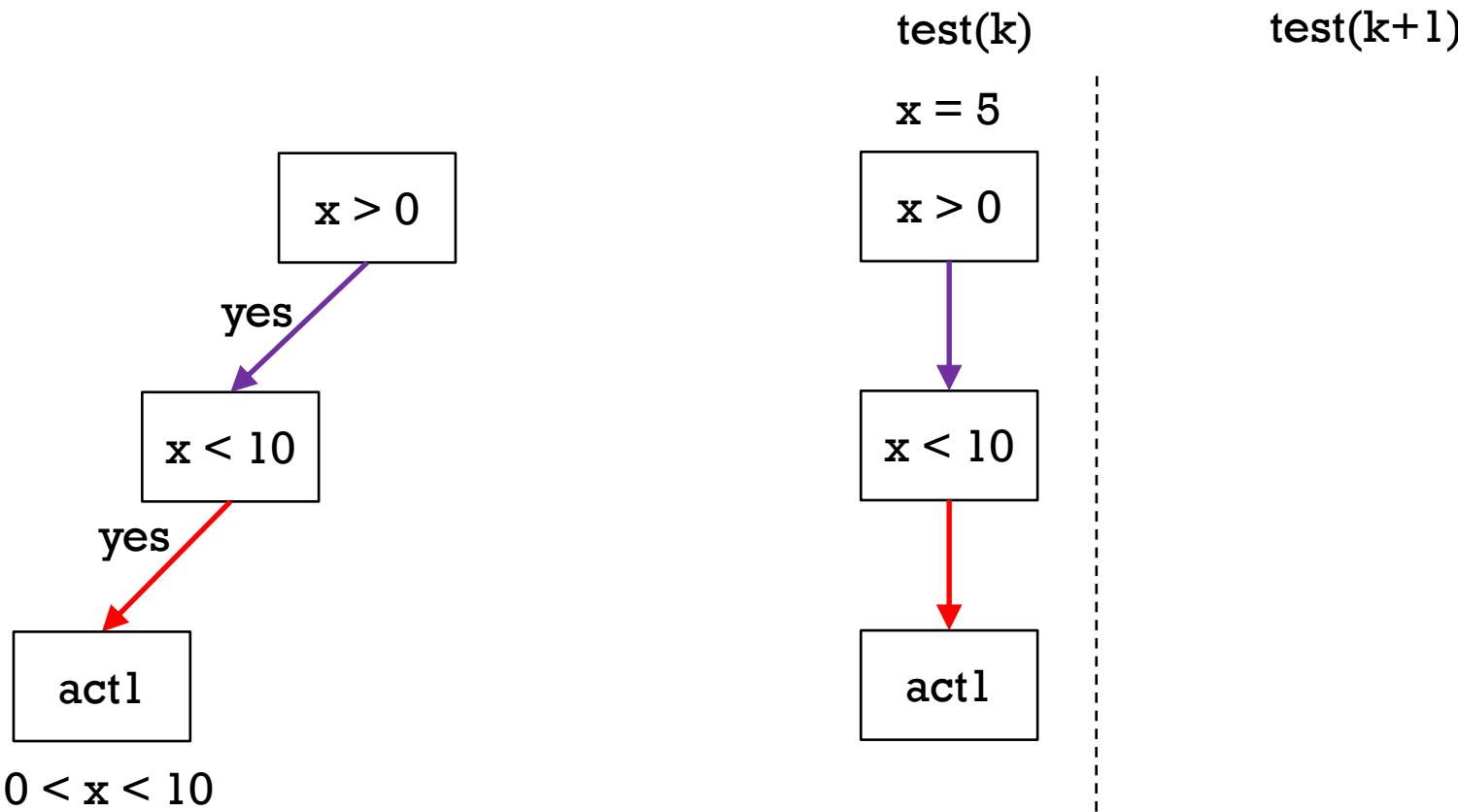
# INTRODUCTION: CONCOLIC EXECUTION

- こんなイメージ



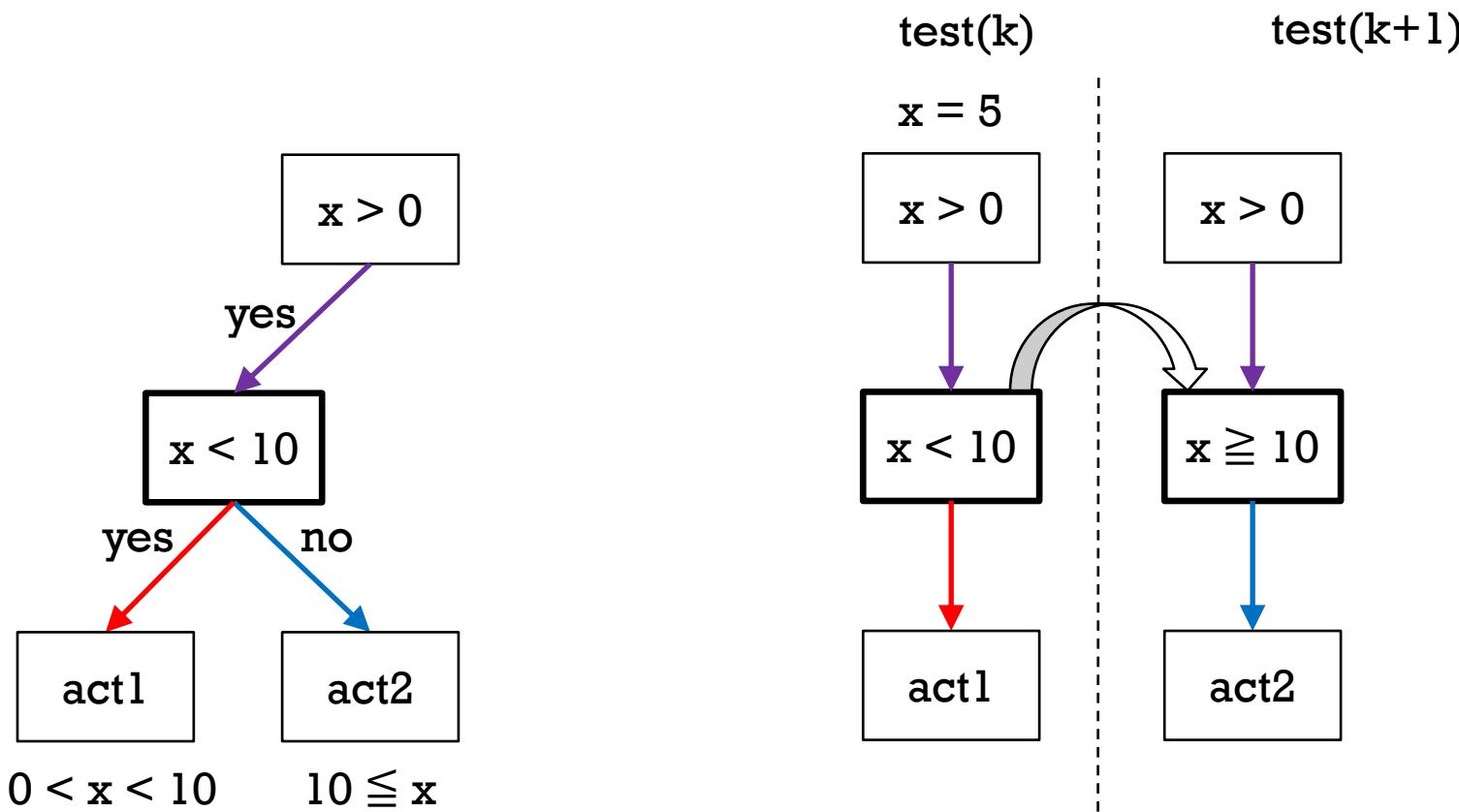
# INTRODUCTION: CONCOLIC EXECUTION

- こんなイメージ



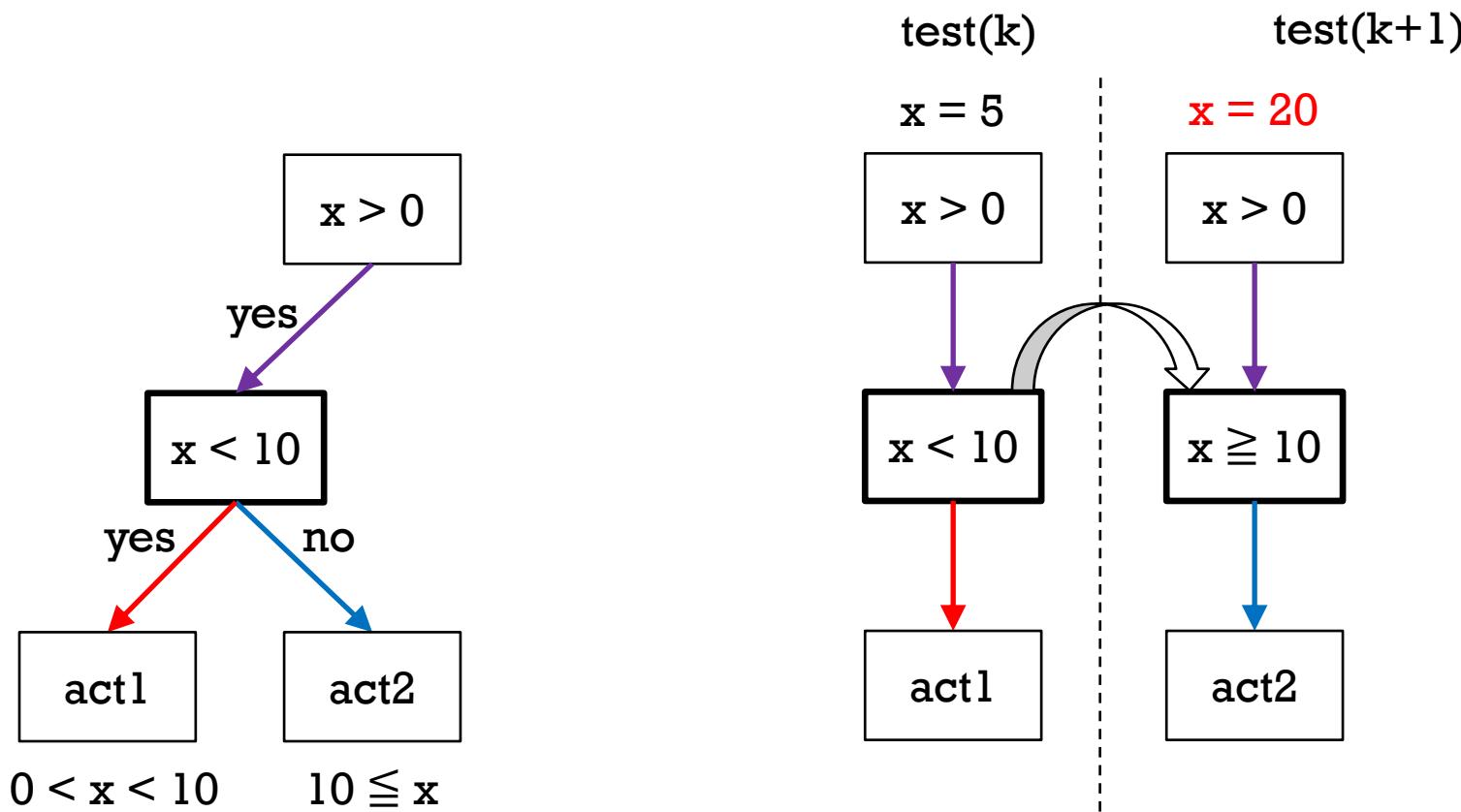
# INTRODUCTION: CONCOLIC EXECUTION

- こんなイメージ



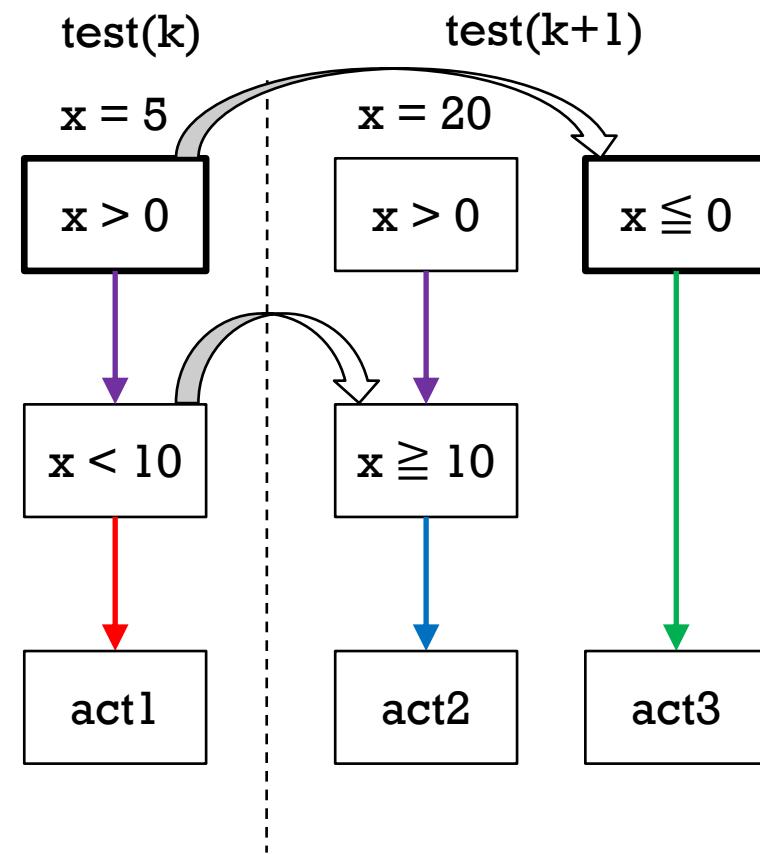
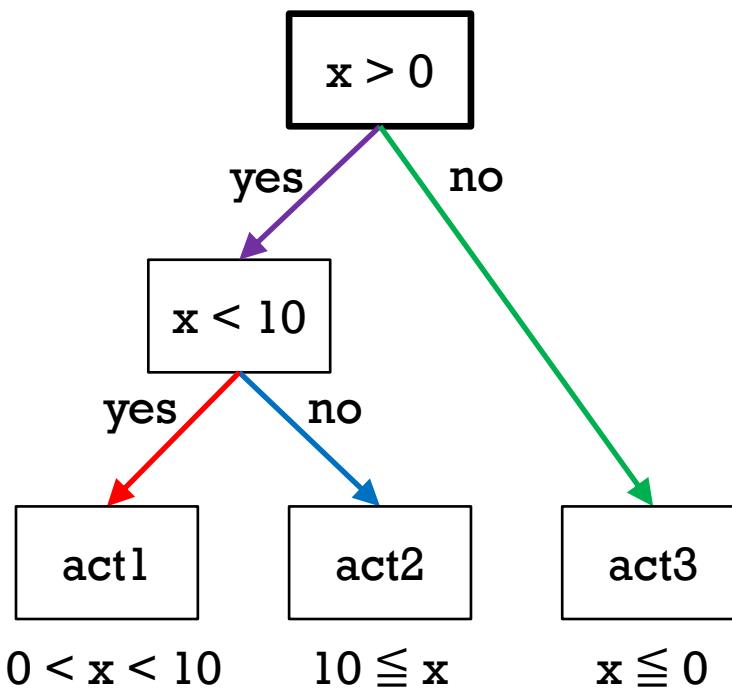
# INTRODUCTION: CONCOLIC EXECUTION

- こんなイメージ



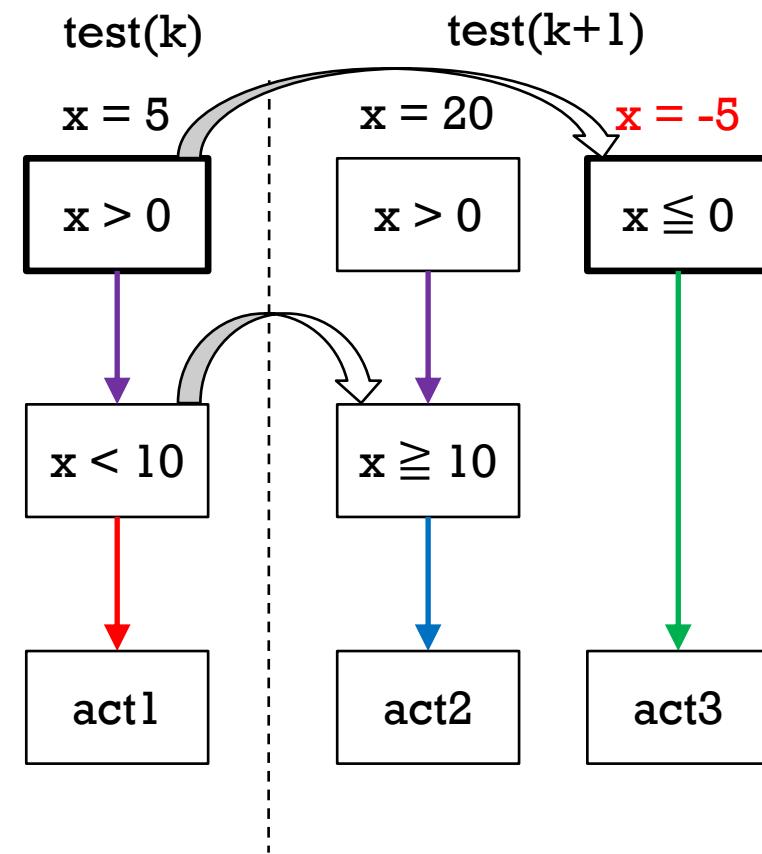
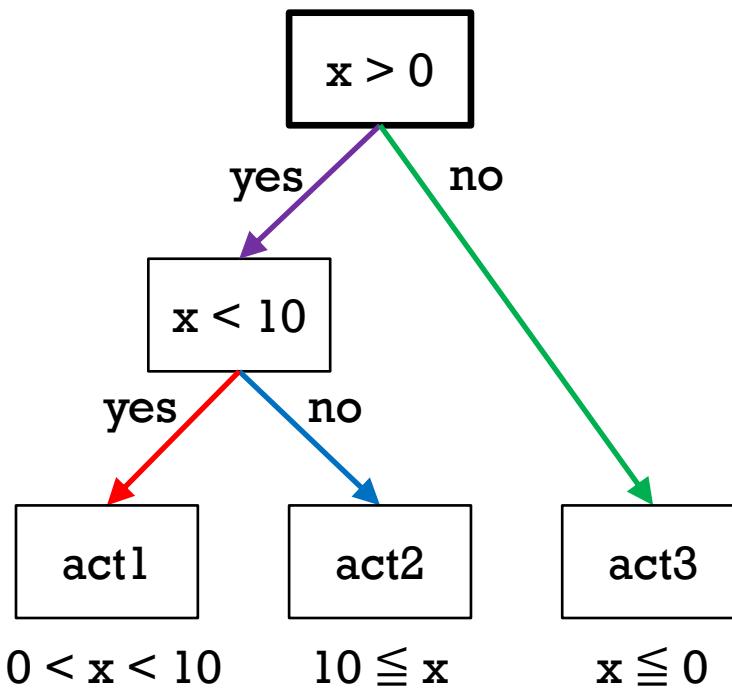
# INTRODUCTION: CONCOLIC EXECUTION

- こんなイメージ



# INTRODUCTION: CONCOLIC EXECUTION

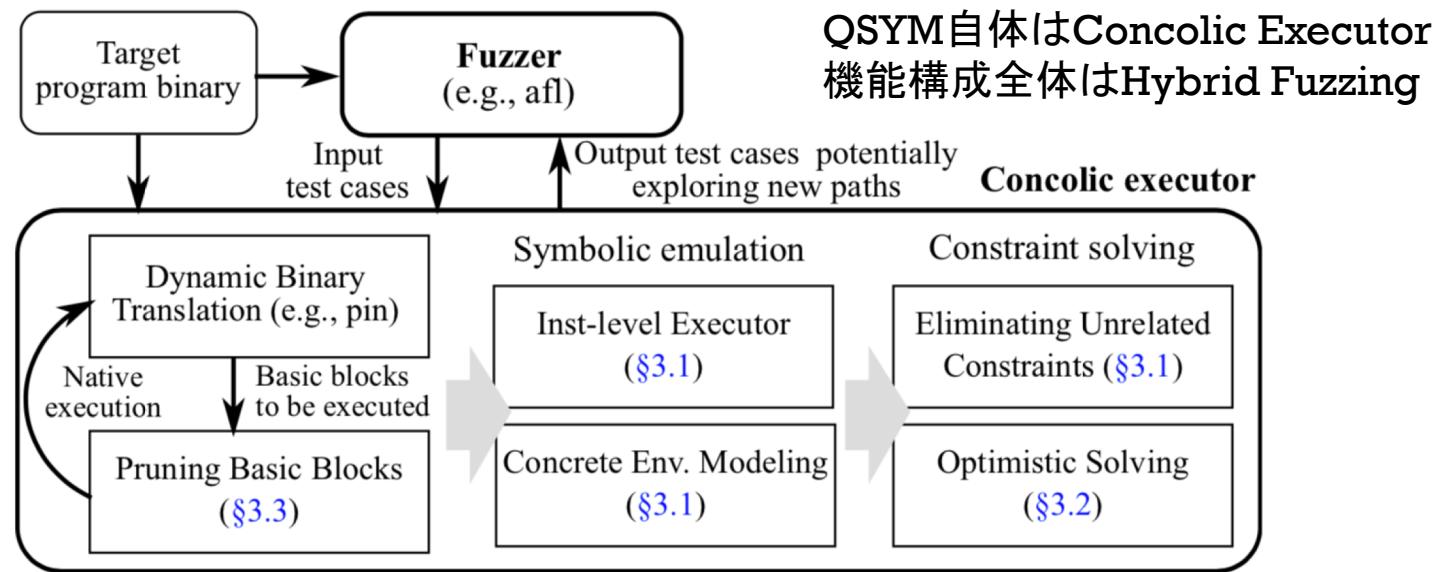
- こんなイメージ



# INTRODUCTION: HYBRID FUZZING

- **Hybrid Fuzzing** = Fuzzing + Concolic Execution

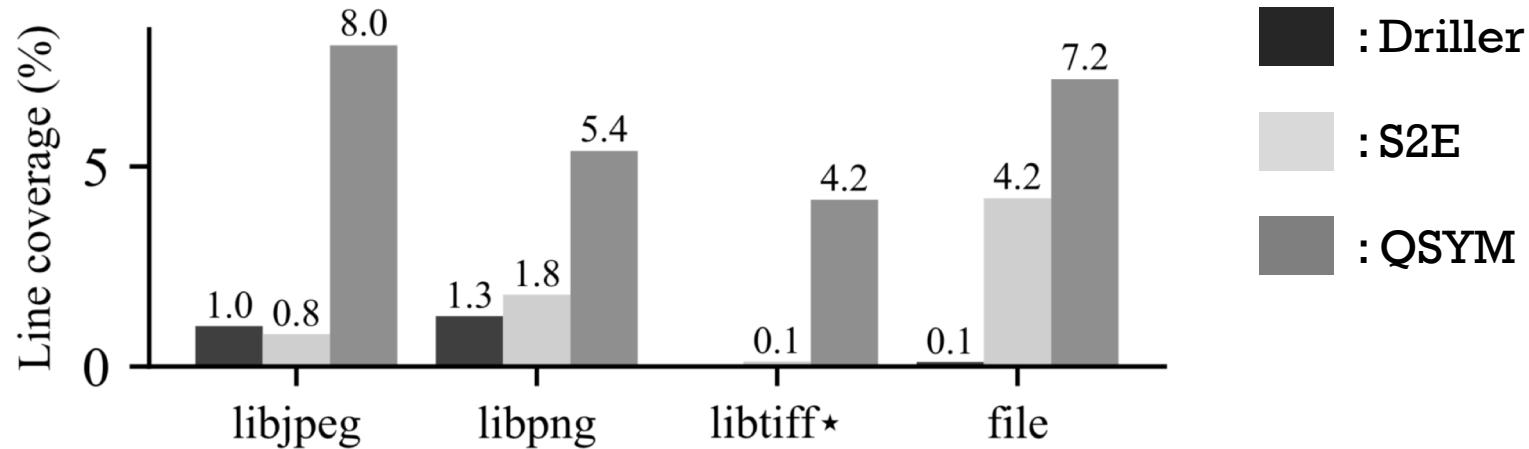
- 制約が厳しい箇所はConcolic Executionで突破
- Fuzzingによる効率的なパス探索でパス爆発を抑制
- といういいところを期待



# MOTIVATION

---

- 既存のHybrid Fuzzingは現実のアプリに対応できていない！
  - 理由①：（主にConcolic実行の部分が）遅い
  - 理由②：システムコール／外部環境をサポートできない
  - 理由③：複雑な制約を解消できない



- ex. ②) QSYMで生成したテストケース
  - Driller[8]やS2E[5]に食わせてもカバレッジが上がらない

[8] Driller: Augmenting fuzzing through selective symbolic execution [NDSS'16]

[5] S2E: A platform for in-vivo multi-path analysis of software systems [ASPIOS'11]

# MOTIVATION ①

---

- 何故既存のConcolic Executorは遅いのか？

- 中間表現（IR）を使っているから

- - 
  - 
  - 
  - 
  - 
  - 
  -

# MOTIVATION ①

---

- 何故既存のConcolic Executorは遅いのか？
  - 中間表現（IR）を使っているから
    - 命令種類が減るためIRを使うと実装がラク
      - 例) KLEE[42]: amd64の1,795命令 → LLVM IRの62命令
    - しかしIRへの変換オーバーヘッドと、実行命令数の増加が生じる
      - 例) x86 → VEX IR では一つのx86命令を表現するのに平均4.7IR命令
  - 
  - 
  -

# MOTIVATION ①

---

- 何故既存のConcolic Executorは遅いのか？
  - 中間表現（IR）を使っているから
    - 命令種類が減るためIRを使うと実装がラク
      - 例) KLEE[42]: amd64の1,795命令 → LLVM IRの62命令
    - しかしIRへの変換オーバーヘッドと、実行命令数の増加が生じる
      - 例) x86 → VEX IR では一つのx86命令を表現するのに平均4.7IR命令
  - さらに、最適化の粒度が粗くなる
    - キャッシングの関係でBBL単位での最適化しかできない
    - →命令単位で見ればもっと最適化できる

# MOTIVATION ①

- 何故既存のConcolic Executorは遅いのか？

Bad!

- 中間表現 (IR) を使っているから

- 命令種類が減るためIRを使うと実装がラク
  - 例) KLEE[42]: amd64の1,795命令 → LLVM IRの62命令
- しかしIRへの変換オーバーヘッドと、実行命令数の増加が生じる
  - 例) x86 → VEX IR では一つのx86命令を表現するのに平均4.7IR命令
- さらに、最適化の粒度が粗くなる
  - キャッシングの関係でBBL単位での最適化しかできない
  - →命令単位で見ればもっと最適化できる

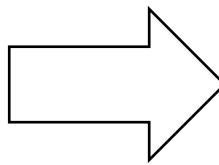
# MOTIVATION ①

- 何故既存のConcolic Executorは遅いのか？

Bad!

- 中間表現 (IR) を使っているから

- 命令種類が減るためIRを使うと実装がラク
  - 例) KLEE[42]: amd64の1,795命令 → LLVM IRの62命令
- しかしIRへの変換オーバーヘッドと、実行命令数の増加が生じる
  - 例) x86 → VEX IR では一つのx86命令を表現するのに平均4.7IR命令
- さらに、最適化の粒度が粗くなる
  - キャッシングの関係でBBL単位での最適化しかできない
  - →命令単位で見ればもっと最適化できる



QSYM: 実装に手間かけ、IRを排除する

# MOTIVATION ②

---

- システムコールや外部環境をサポートできないとは?
  - Symbolic Executionでの実行形式
    - **Forking-based** (セーブ & ロード)
      - Concolic Executionだけならオーバーヘッド面の利点有り
      - Hybrid Fuzzingではこの利点が薄れる
    - 外部環境（ファイル、ヒープメモリ）の影響まで考慮できない
  - 
  - 
  -

# MOTIVATION ②

---

- システムコールや外部環境をサポートできないとは?
  - Symbolic Executionでの実行形式
    - **Forking-based** (セーブ & ロード)
      - Concolic Executionだけならオーバーヘッド面の利点有り
      - Hybrid Fuzzingではこの利点が薄れる
      - 外部環境（ファイル、ヒープメモリ）の影響まで考慮できない
    - **Re-execution-based** (最初から)
      - 再実行するので外部環境の影響を受けない
      - 一般にはConcolic Executionでの再実行は重い

# MOTIVATION ②

- システムコールや外部環境をサポートできないとは？

- Symbolic Executionでの実行形式

Bad!

- Forking-based (セーブ & ロード)

- Concolic Executionだけならオーバーヘッド面の利点有り
    - Hybrid Fuzzingではこの利点が薄れる
    - 外部環境（ファイル、ヒープメモリ）の影響まで考慮できない

Good!

- Re-execution-based (最初から)

- 再実行するので外部環境の影響を受けない
    - 一般にはConcolic Executionでの再実行は重い

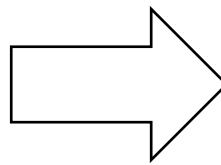
# MOTIVATION ②

- システムコールや外部環境をサポートできないとは？

- Symbolic Executionでの実行形式

- Bad!*
- **Forking-based** (セーブ & ロード)
      - Concolic Executionだけならオーバーヘッド面の利点有り
        - Hybrid Fuzzingではこの利点が薄れる
      - 外部環境（ファイル、ヒープメモリ）の影響まで考慮できない

- Good!*
- **Re-execution-based** (最初から)
      - 再実行するので外部環境の影響を受けない
      - 一般にはConcolic Executionでの再実行は重い



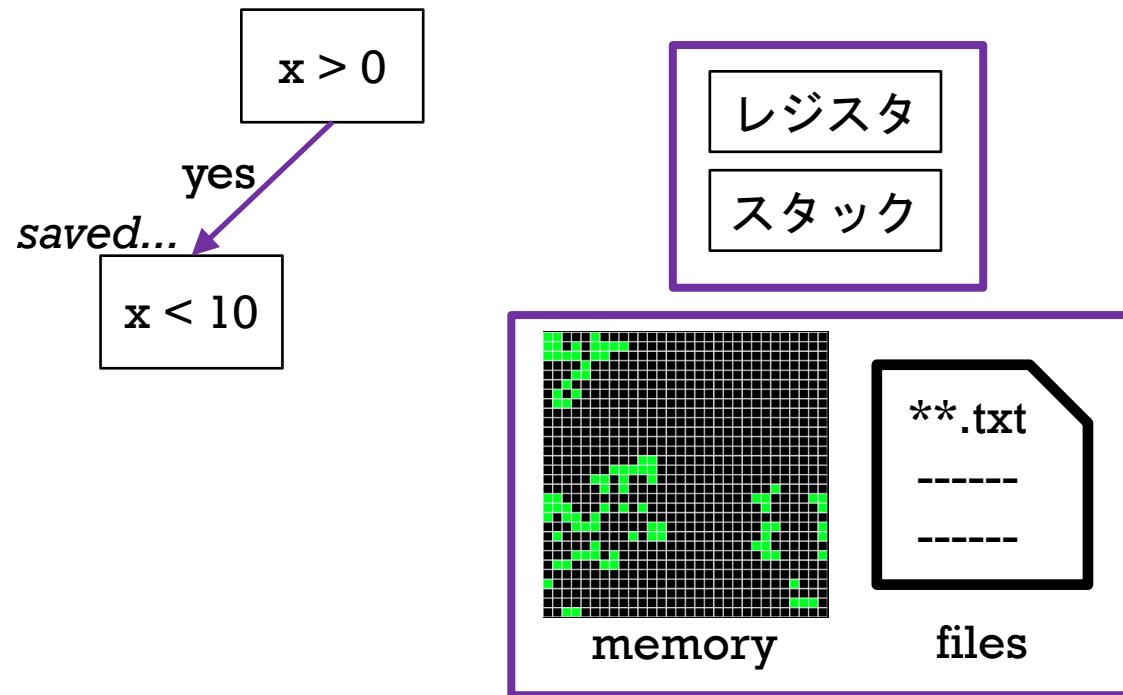
QSYM: 高パフォーマンス再実行

& 最低限のモデリング

& 外部環境のブラックボックス化

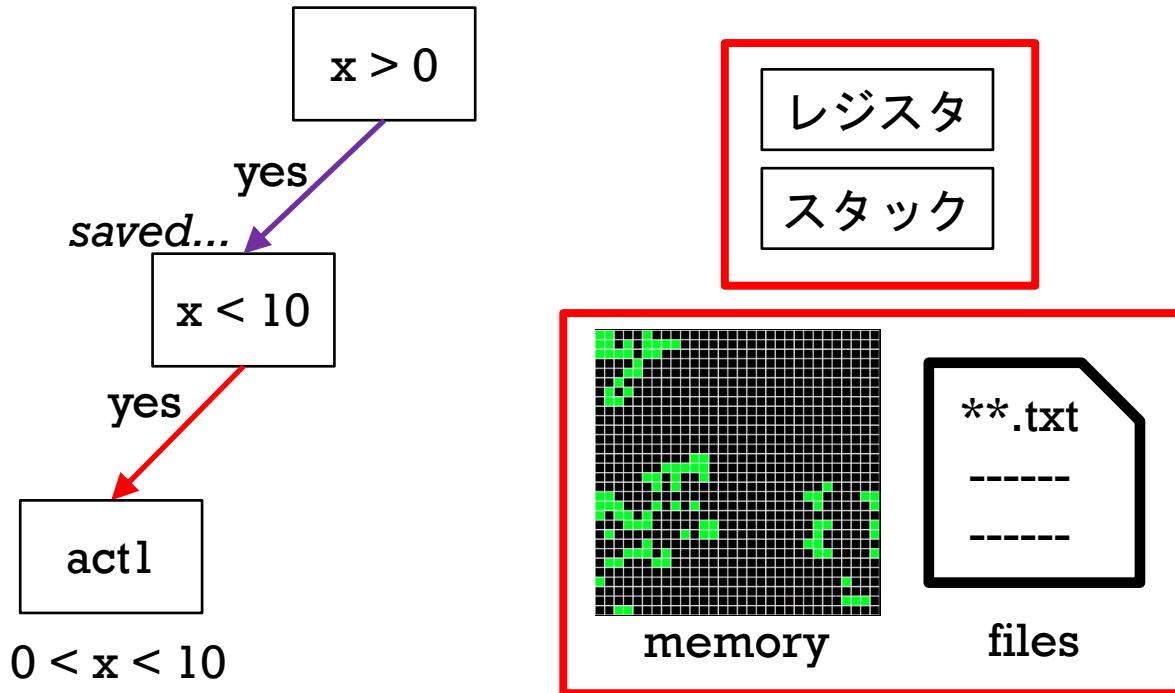
# MOTIVATION ② DETAIL

- Forking-based: 同一のConcolic実行内で複数のパスを実行する
  - 別のパスを実行するためにどこかの分岐でセーブ & ロード
  - 利点: Concolic実行の回数を減らすことで、再実行のオバヘが減少
  - 欠点:
    - プログラムの状態を記録するのは地味に高コスト（しかも不完全）
    - Hybrid FuzzingではFuzzingの段階で再実行を挟むのであんまり恩恵がない



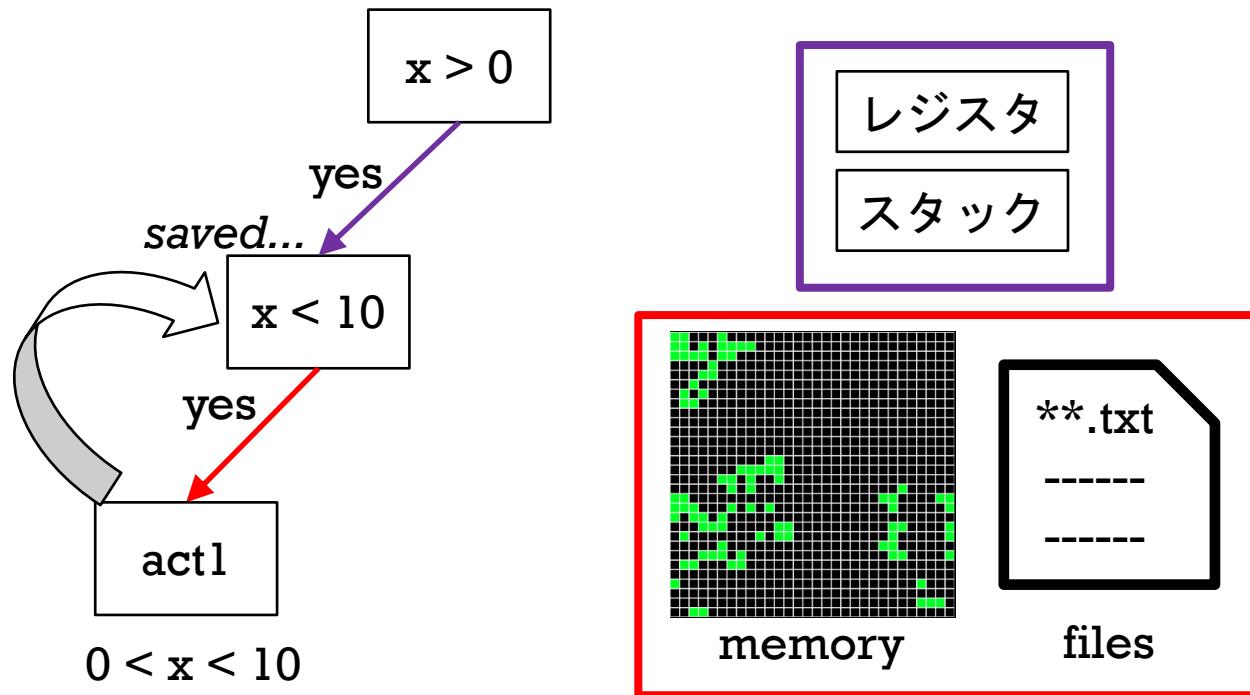
# MOTIVATION ② DETAIL

- Forking-based: 同一のConcolic実行内で複数のパスを実行する
  - 別のパスを実行するためにどこかの分岐でセーブ＆ロード
  - 利点: Concolic実行の回数を減らすことで、再実行のオバヘが減少
  - 欠点:
    - プログラムの状態を記録するのは地味に高コスト（しかも不完全）
    - Hybrid FuzzingではFuzzingの段階で再実行を挟むのであんまり恩恵がない



# MOTIVATION ② DETAIL

- Forking-based: 同一のConcolic実行内で複数のパスを実行する
  - 別のパスを実行するためにどこかの分岐でセーブ&ロード
  - 利点: Concolic実行の回数を減らすことで、再実行のオバヘが減少
  - 欠点:
    - プログラムの状態を記録するのは地味に高コスト（しかも不完全）
    - Hybrid FuzzingではFuzzingの段階で再実行を挟むのであんまり恩恵がない



# MOTIVATION ③

---

- Concolic Executionでは制約を厳密に求めることで健全性を保証
  - 健全性: 起こりえない制約を持つテストケースは生成されないこと
  - 
  -

# MOTIVATION ③

- Concolic Executionでは制約を厳密に求めることで健全性を保証
  - 健全性: 起こりえない制約を持つテストケースは生成されないこと
- しかし、健全性の担保が高速化を阻害することがある
  - →過制約 (over-constraint)

```
type = int(input());
...
if type == 1
    parse_type1();
...
if type == 2
    parse_type2();
```

type == 1

type == 2

という制約が残り続ける限り、  
のような制約は評価されない  
→制約全体が常に偽になるため  
(unsatisfied)

# MOTIVATION ③

- Concolic Executionでは制約を厳密に求めることで健全性を保証
  - 健全性: 起こりえない制約を持つテストケースは生成されないこと
- しかし、健全性の担保が高速化を阻害することがある

Bad! →過制約 (over-constraint)

```
type = int(input());
...
if type == 1
    parse_type1();
...
if type == 2
    parse_type2();
```

type == 1

type == 2

という制約が残り続ける限り、  
のような制約は評価されない  
→制約全体が常に偽になるため  
(unsatisfied)

# MOTIVATION ③

- Concolic Executionでは制約を厳密に求めることで健全性を保証
  - 健全性: 起こりえない制約を持つテストケースは生成されないこと
- しかし、健全性の担保が高速化を阻害することがある

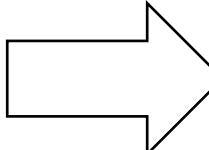
Bad! →過制約 (over-constraint)

```
type = int(input());
...
if type == 1
    parse_type1();
...
if type == 2
    parse_type2();
```

type == 1

type == 2

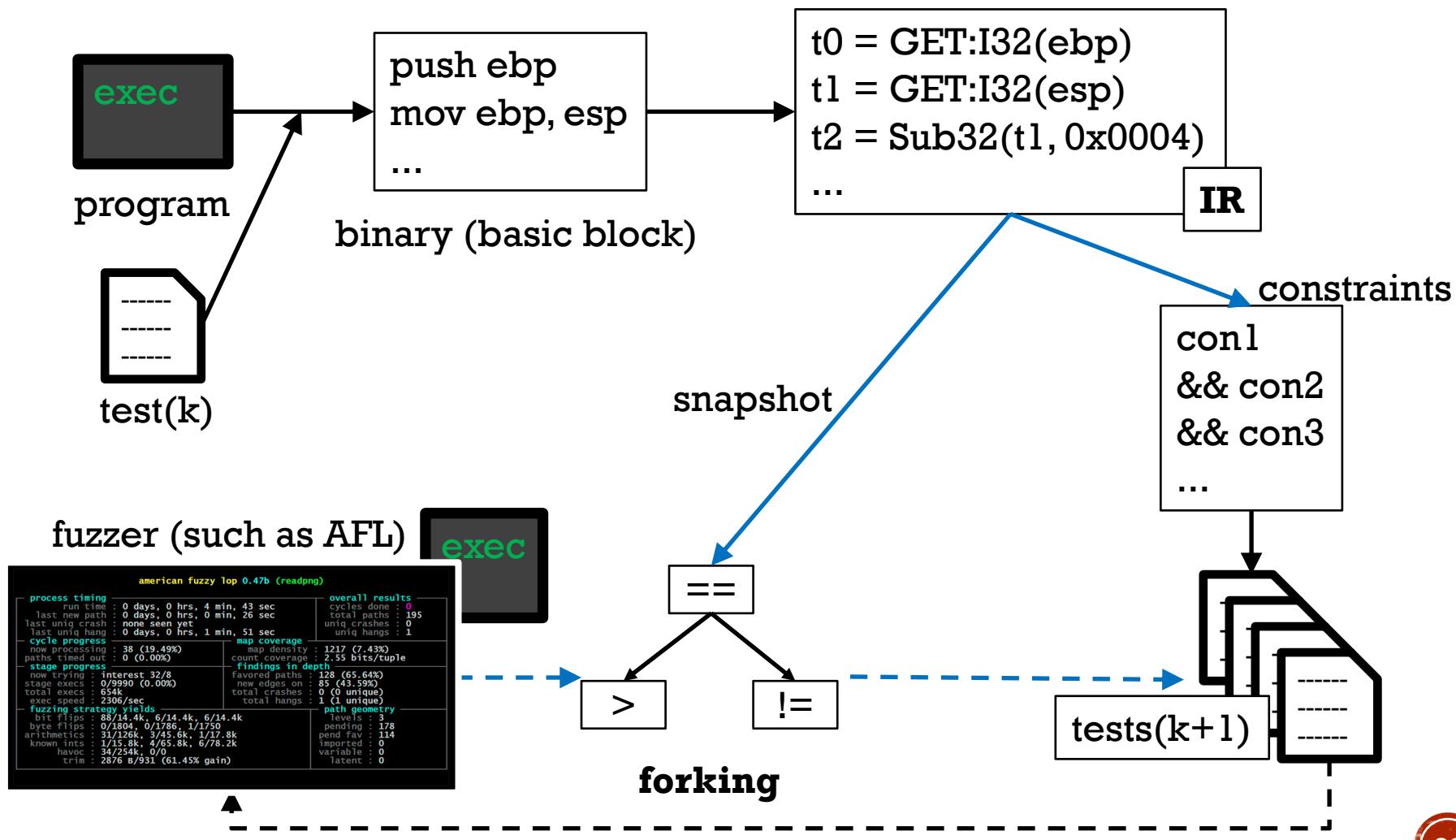
という制約が残り続ける限り、  
のような制約は評価されない  
→制約全体が常に偽になるため  
(unsatisfied)



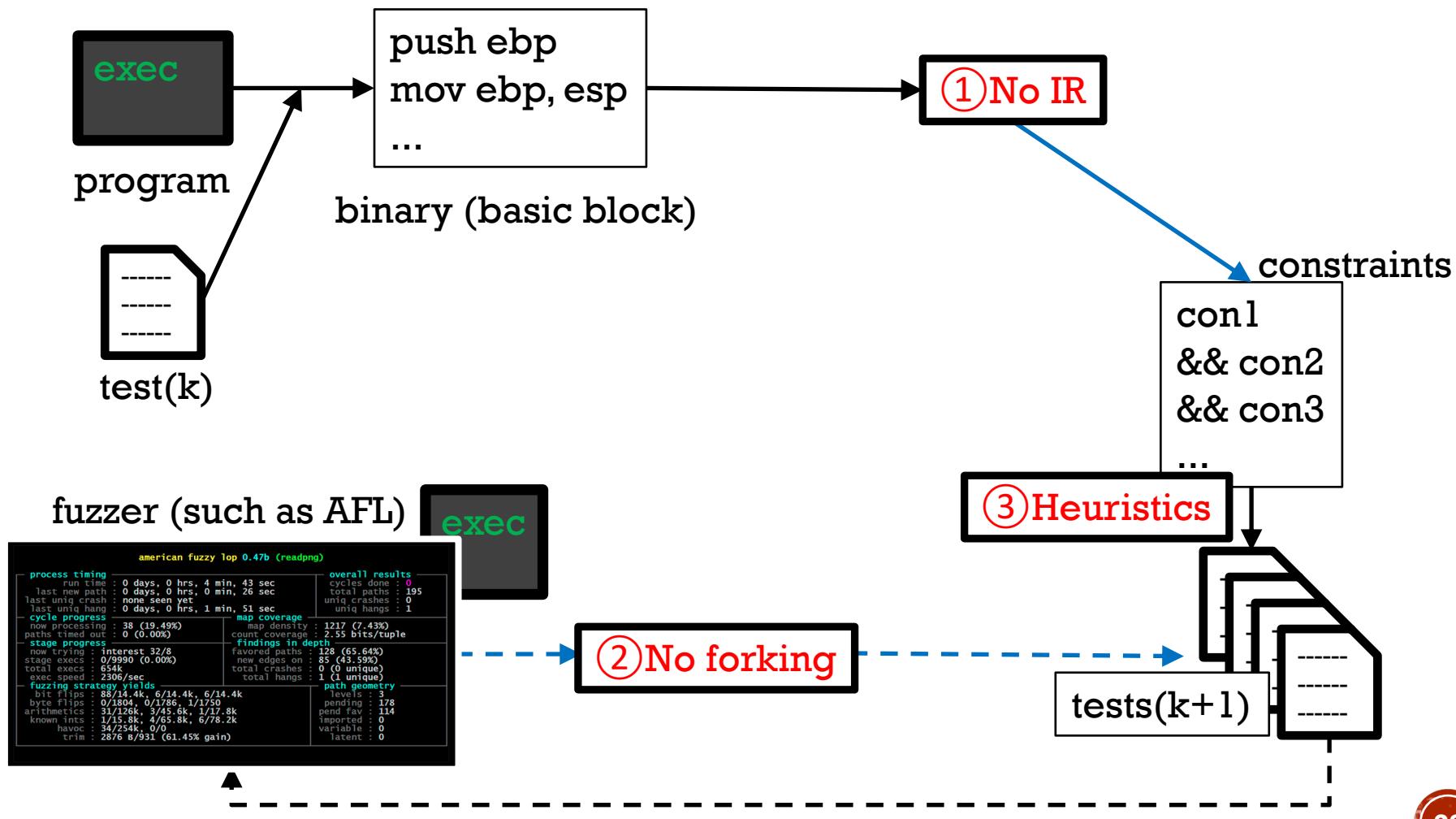
QSYM: 各種ヒューリスティクスを導入

Optimistic Solving (& BBL pruning)

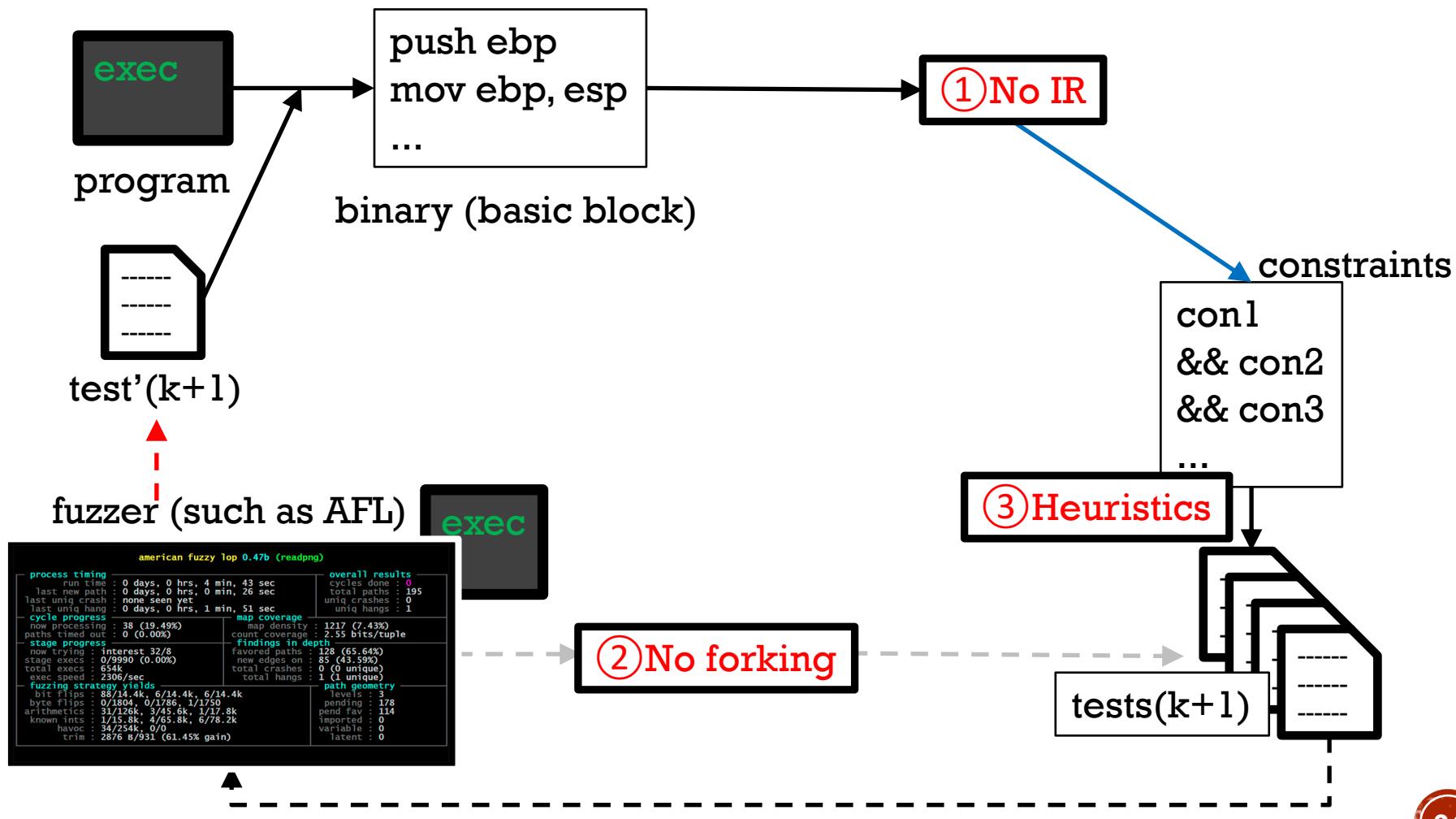
# MOTIVATION (1 CONCOLIC EXEC IN FORKING)



# MOTIVATION (1 CONCOLIC EXEC IN QSYM)



# MOTIVATION (1 CONCOLIC EXEC IN QSYM)



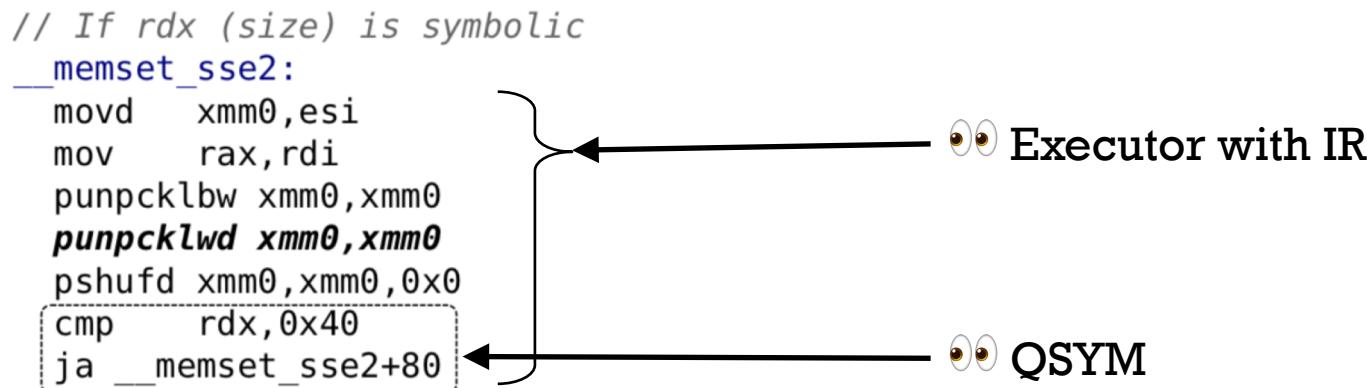
# APPROACH FOR ①: REMOVE IR

- IRを使うと遅くなる(命令数増加、変換の必要性...)
  - QSYM: 直のアーキテクチャ命令(x86\_64)で制約を収集

- 
- 
- 
- 
- 
-

# APPROACH FOR ①: REMOVE IR

- IRを使うと遅くなる(命令数増加、変換の必要性...)
  - QSYM: 直のアーキテクチャ命令(x86\_64)で制約を収集
- QSYM: DBTで命令単位の汚染解析を行い要エミュな箇所を特定
  - DBT: Dynamic Binary Translation (Pin[24]とかValgrind[16]とか)
    - 実行中のバイナリに対して介入できる
  - 利点: IRに比べて軽量な解析が可能
  - 利点: 命令単位で最適化することで、複雑な命令の追跡を除外可能
    - punpcklwdなど

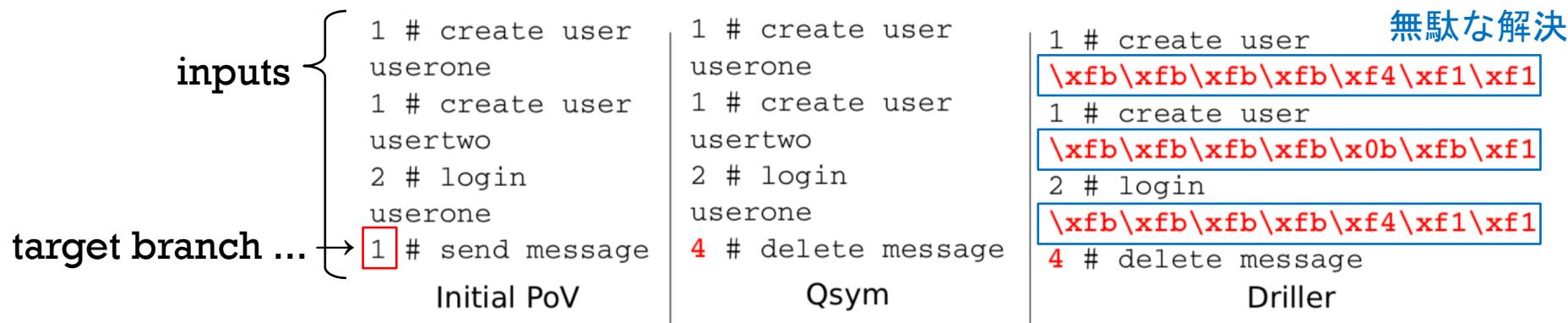


[24] Pin: building customized program analysis tools with dynamic instrumentation [PLDI'05]

[16] Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation [PLDI'07]

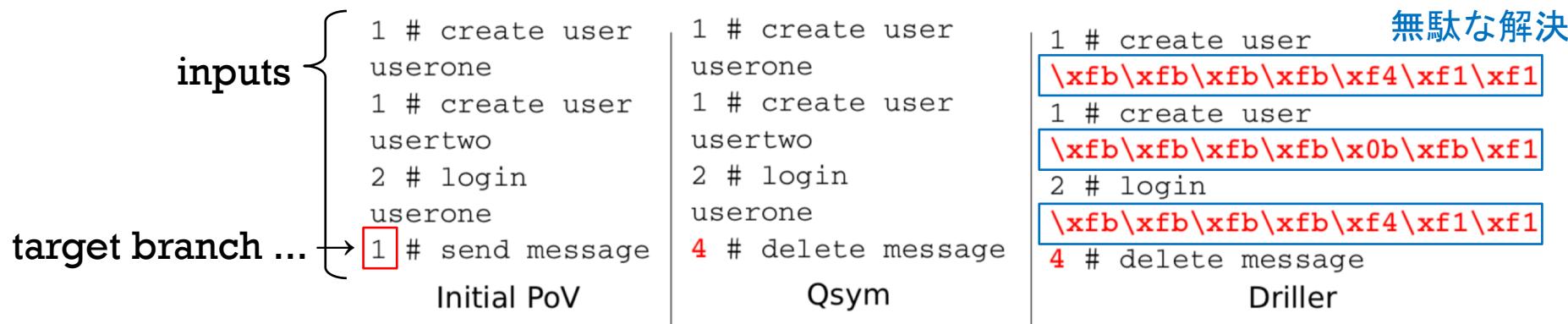
# APPROACH FOR ①: RELEVANT CONSTRAINTS

- 既存手法(such as Driller):
  - 更新された箇所全部を解決しようとする...
  - 意訳: k-1回目とk回目の差分内を可能な限り更新してk+1回目のテストを生成
  - 複雑なプログラムでは制約を解決できない場合多 (@ Hybrid Fuzzing)
  - 無駄な時間



# APPROACH FOR ①: RELEVANT CONSTRAINTS

- 既存手法(such as Driller):
  - 更新された箇所全部を解決しようとする...
    - 意訳: k-1回目とk回目の差分内を可能な限り更新してk+1回目のテストを生成
    - 複雑なプログラムでは制約を解決できない場合多 (@ Hybrid Fuzzing)
    - 無駄な時間



- QSYM: 無関係な制約の排除
  - トグルしたい制約の解決は最小限に抑える

# APPROACH FOR ②: FAST RE-EXECUTION

- QSYM: **re-execution-based** の手法を採択
  - Approach for ① から再実行自体を高速化
    - → 不完全な結果になる forking-based からの脱却
- QSYM: Symbolicに係る（入力）システムコールはモデリング
  - read()とか
- QSYM: その他システムコール及び外部環境はBlack-box化[4, 19]
  - Concreteな値を渡して中の挙動はスルー

■

[4] Automated whitebox fuzz testing [NDSS'08]

[19] DART: Directed Automated Random Testing [PLDI'05]

# APPROACH FOR ②: FAST RE-EXECUTION

正確

- QSYM: **re-execution-based** の手法を採択
  - Approach for ① から再実行自体を高速化
    - → 不完全な結果になる forking-based からの脱却

手動

- QSYM: Symbolicに係る（入力）システムコールはモデリング
  - read()とか

正確

- QSYM: その他システムコール及び外部環境はBlack-box化[4, 19]
  - Concreteな値を渡して中の挙動はスルー

■

[4] Automated whitebox fuzz testing [NDSS'08]

[19] DART: Directed Automated Random Testing [PLDI'05]

# APPROACH FOR ②: FAST RE-EXECUTION

正確

- QSYM: **re-execution-based** の手法を採択
  - Approach for ① から再実行自体を高速化
    - → 不完全な結果になるforking-basedからの脱却

手動

- QSYM: Symbolicに係る（入力）システムコールはモデリング
  - read()とか

正確

- QSYM: その他システムコール及び外部環境はBlack-box化[4, 19]
  - Concreteな値を渡して中の挙動はスルー

- → QSYMの方針: 既存の正確な技術を高速化

[4] Automated whitebox fuzz testing [NDSS'08]

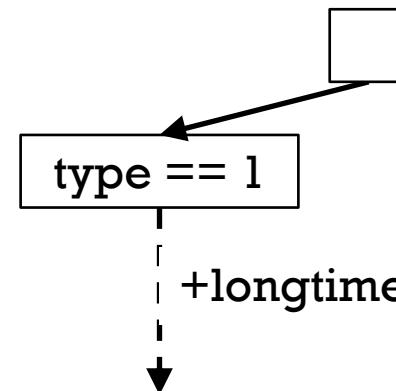
[19] DART: Directed Automated Random Testing [PLDI'05]

# APPROACH FOR ③: OPTIMISTIC SOLVING

- 既存のConcolic Executorは、収集済みの制約を絶対視しがち
  - 解決できない(un satisfied)制約は無視する
- 
- 
- 
- 

Exist  
Executor

```
type = int(input());
...
if type == 1
    parse_type1();
...
if type == 2
    parse_type2();
```

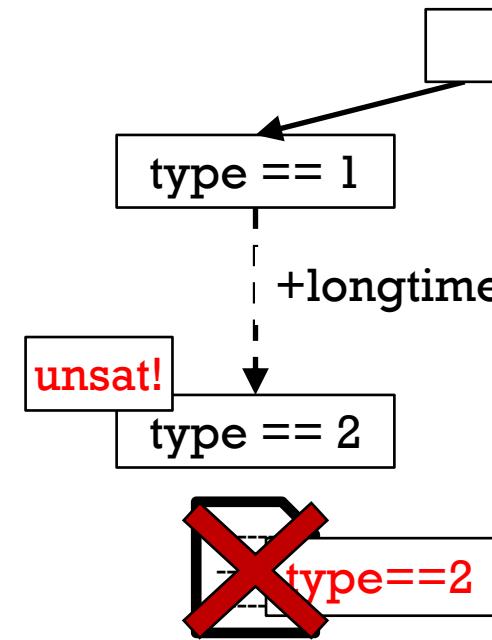


# APPROACH FOR ③: OPTIMISTIC SOLVING

- 既存のConcolic Executorは、収集済みの制約を絶対視しがち
  - 解決できない(un satisfied)制約は無視する
- 
- 
- 
- 

Exist  
Executor

```
type = int(input());  
...  
if type == 1  
    parse_type1();  
...  
if type == 2  
    parse_type2();
```

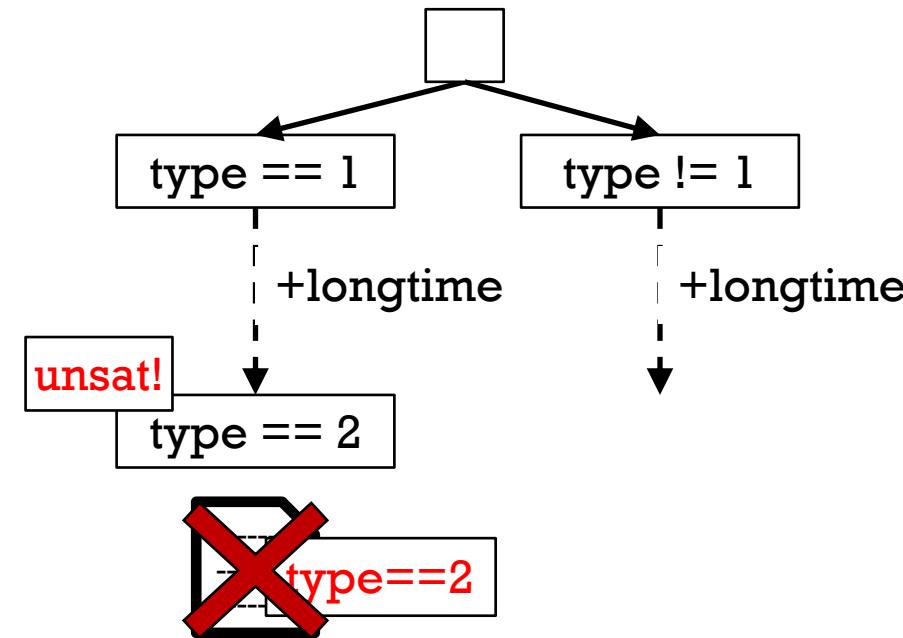


# APPROACH FOR ③: OPTIMISTIC SOLVING

- 既存のConcolic Executorは、収集済みの制約を絶対視しがち
  - 解決できない(un satisfied)制約は無視する
- 
- 
- 
- 

**Exist  
Executor**

```
type = int(input());
...
if type == 1
    parse_type1();
...
if type == 2
    parse_type2();
```

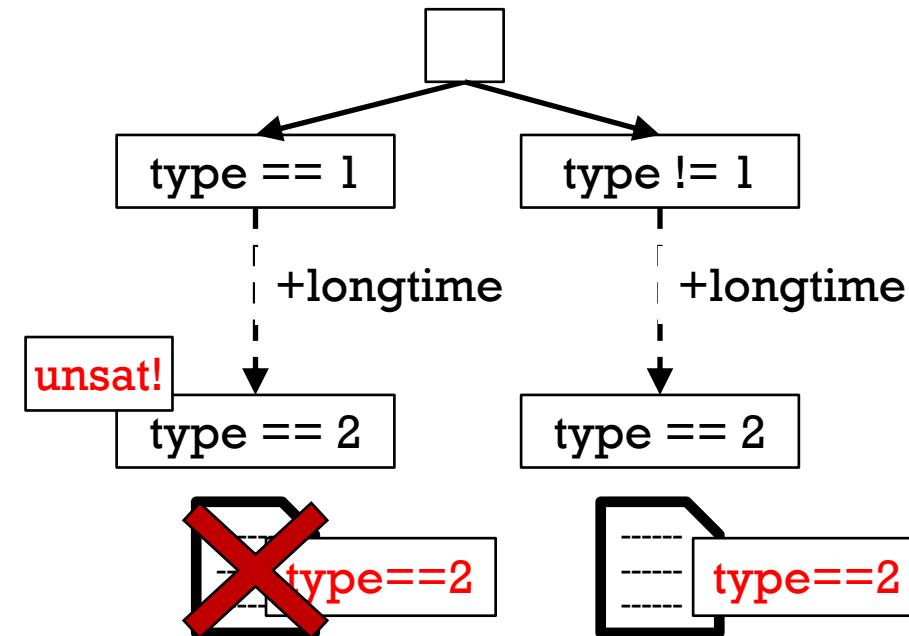


# APPROACH FOR ③: OPTIMISTIC SOLVING

- 既存のConcolic Executorは、収集済みの制約を絶対視しがち
  - 解決できない(un satisfied)制約は無視する
- 
- 
- 
- 

**Exist  
Executor**

```
type = int(input());
...
if type == 1
    parse_type1();
...
if type == 2
    parse_type2();
```



# APPROACH FOR ③: OPTIMISTIC SOLVING

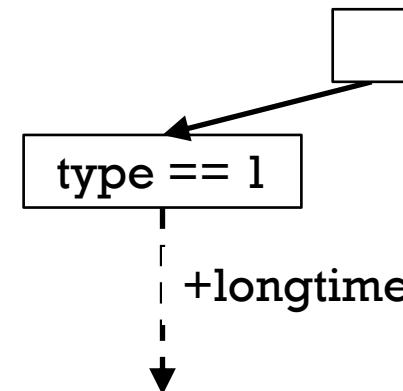
- 既存のConcolic Executorは、収集済みの制約を絶対視しがち
  - 解決できない(un satisfied)制約は無視する
- QSYM: **Optimistic Solving**
  - (特定パターンの) 解決できない制約が出現
    - 解決できるように収集済みの制約をスイッチ
    - ありえない制約を持つテストケースはFuzzingで弾ける

QSYM

```

type = int(input());
...
if type == 1
    parse_type1();
...
if type == 2
    parse_type2();

```



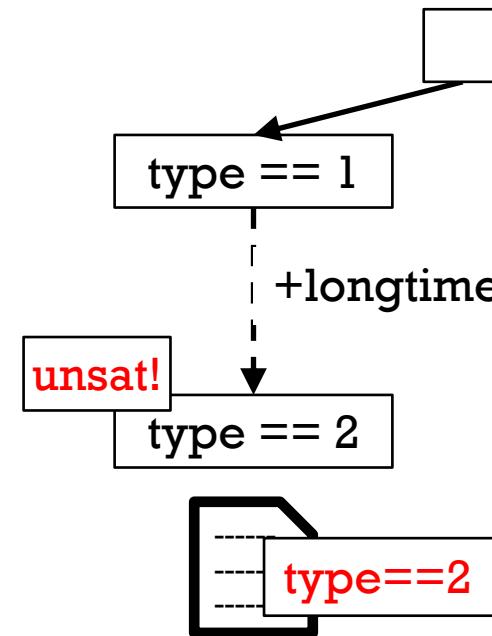
# APPROACH FOR ③: OPTIMISTIC SOLVING

- 既存のConcolic Executorは、収集済みの制約を絶対視しがち
  - 解決できない(un satisfied)制約は無視する
- QSYM: **Optimistic Solving**
  - (特定パターンの) 解決できない制約が出現
    - 解決できるように収集済みの制約をスイッチ
    - ありえない制約を持つテストケースはFuzzingで弾ける

QSYM

```

type = int(input());
...
if type == 1
    parse_type1();
...
if type == 2
    parse_type2();
  
```



# APPROACH FOR ③: BBL PRUNING

## ■ QSYM: BBL pruning

- BBL (Basic Block): 制御が変わらない連續する命令群...
  - 意訳: call/ret/jmpで区切られる命令群
- 頻繁に実行されるBBL → 制約の生成を停止
  - 経験則: 非線形な制約ができて解けない場合が多いから
- if  $Freq(BBL_i) == 2^j$ , then QSYM generates constraint from  $BBL_i$ 
  - ただ、間引きすぎるとパス探索に悪影響
    - → Grouping Multiple Executions & Context-Sensitivity で影響を抑制
- Grouping Multiple Executions :
  - if  $NumOfExec(BBL_i) == GroupSize$ , then  $Freq(BBL_i)++$
  - GroupSize: 明記されてないが、恐らくユーザ定義のパラメータ
- Context-Sensitivity :
  - 各BBLはcontextによって区別される、ということ

# IMPLEMENTATION & ENVIRONMENT

---

- 実装: Intel Pin[24]のツール（プラグイン）としてQSYMを実装
  - C++コード 約16K LoC
  - Python APIと共に公開済み\*
  
- 実験環境
  - OS: Ubuntu-14.04 LTS
  - CPU: Intel Xeon E7-4820 (2.0GHzコア × 8)
  - Mem: 256 GB RAM

[24] Pin: building customized program analysis tools with dynamic instrumentation [PLDI'05]

\* <https://github.com/sslab-gatech/qsym>

# EXPERIMENT (NEW BUGS)

- QSYM vs AFL[1], OSS-Fuzz[3] ( as a bug tester )
- 検体: AFL, OSS-Fuzzでテスト済みの検体11個
  - libjpeg, libpng, libtiff, lepton, openjpeg, tcpdump, file, libarchive, audiofile, ffmpeg, binutils
- 各検体について3時間QSYMで検査

Program	CVE	Bug Type	Fuzzer	Fail (Fuzzer)	Fail (Hybrid)
lepton	CVE-2017-8891	Out-of-bounds read	AFL	Meet complex constraints	Explore deep code paths
openjpeg	CVE-2017-12878	Heap overflow	OSS-Fuzz	Meet complex constraints	Support external environments
	Fixed by other patch	NULL dereference			
tcpdump	CVE-2017-11543*	Heap overflow	AFL	Find where to change*	Support external environments
file	CVE-2017-1000249*	Stack overflow	OSS-Fuzz	Meet complex constraints	Explore deep code paths
libarchive	Wait for patch	NULL dereference	OSS-Fuzz	Meet complex constraints	Support external environments
audiofile	CVE-2017-6836	Heap overflow	AFL	Multi-bytes magic values	Explore deep code paths
	Wait for patch	Heap overflow × 3			
	Wait for patch	Memory leak			
ffmpeg	CVE-2017-17081	Out-of-bounds read	OSS-Fuzz	Meet complex constraints	Support external environments
objdump	CVE-2017-17080	Out-of-bounds read	AFL	Meet complex constraints	Explore deep code paths

- 結果: 8検体で計13個の未知のバグを発見

[1] “american fuzzy lop,” <http://lcamtuf.coredump.cx/afl/> [2015]

[3] “OSS-Fuzz - continuous fuzzing of open source software,” <https://github.com/google/oss-fuzz> [2016]

# EXPERIMENT (SCALABLE)

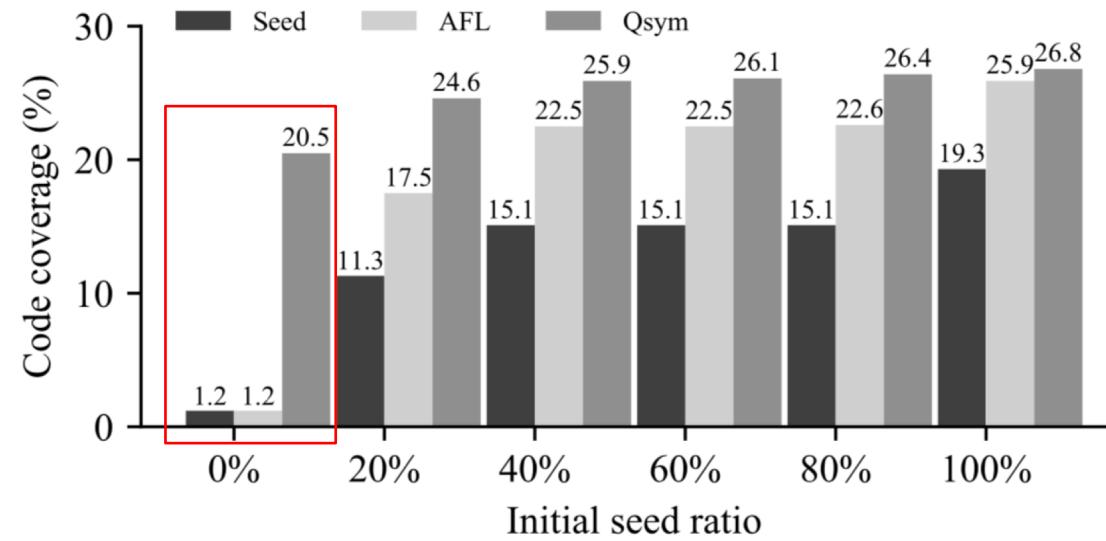
- QSYM vs Driller[8] (as a Concolic Executor)
- 検体: NEW BUGSと同じ11個 (libjpeg, ...)
- 各検体30分で生成したテストケース数を比較
- 結果:
  - Driller: 平均10個
  - QSYM: 平均数百個 → **Drillerの10倍以上**
- また、Drillerはモデリングの不完全さから、5個の検体に未対応

Program	Bug Type	Syscall
libtiff	Erroneous system calls	mmap
openjpeg	Unsupported system calls	set_robust_list
tcpdump	Erroneous system calls	mmap
libarchive	Unsupported system calls	fcntl
ffmpeg	Unsupported system calls	rt_sigaction

[8] Driller: Augmenting fuzzing through selective symbolic execution [NDSS'16]

# EXPERIMENT (CODE COVERAGE)

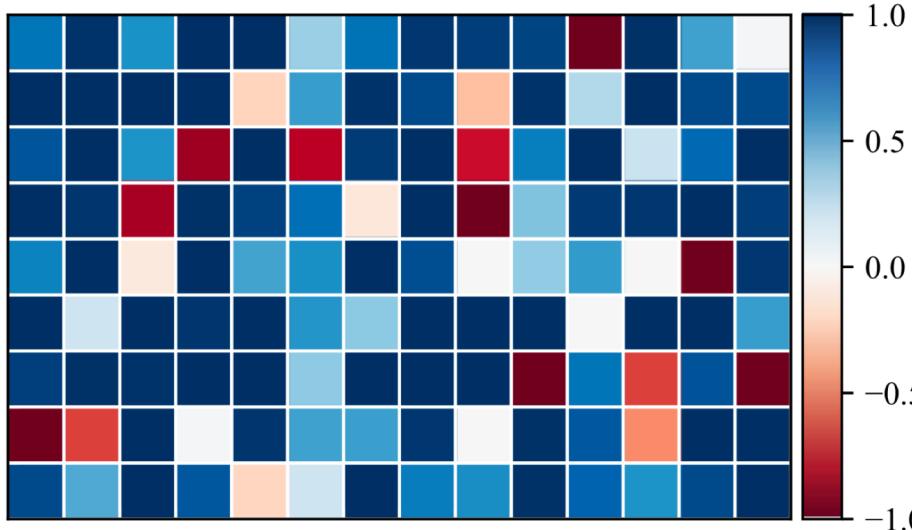
- QSYM vs AFL[1] (as a Fuzzer)
  - QSYMはFuzzing用にAFLを使用
- 検体: libpng (複雑なプログラム代表)
- 141個の高品質なPNG画像を入力、6時間の検査
  - 横軸: テストケース141個の利用率



- 結果: 総じてAFLよりカバレッジ大
  - 特に利用率が低いほど差が大きくなる傾向

# EXPERIMENT (CODE COVERAGE)

- QSYM vs Driller[8] (as a Hybrid Fuzzer)
- 検体: DARPA CGC dataset[30] のうち126個
- 各検体5分ずつ検査、コードカバレッジを比較



各セルが各検体  
青系統: **QSYM wins**  
赤系統: **Driller wins**

色が濃い = 上位互換性が高い  
(他方を網羅しつつ+ $\alpha$ を探索)

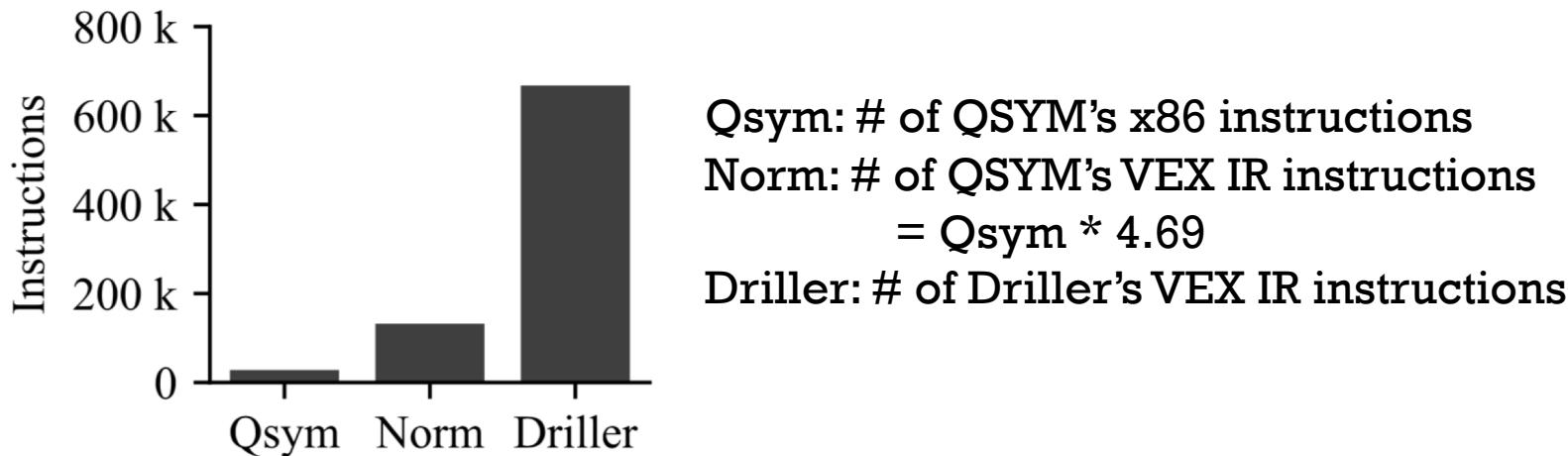
- 結果: **QSYM wins 104/126, Driller wins 18/126**
  - 多くの検体でQSYMはDrillerの上位互換になった

[8] Driller: Augmenting fuzzing through selective symbolic execution [NDSS'16]

[30] Cyber Grand Challenge, <https://www.cybergrandchallenge.com/> [DARPA'16]

# EXPERIMENT (INSTRUCTIONS)

- QSYM vs Driller[8] (as a Concolic Executor)
- 検体: DARPA CGC dataset[30] のうち126個
- 各検体5分ずつ検査、検査命令数を比較(平均値)



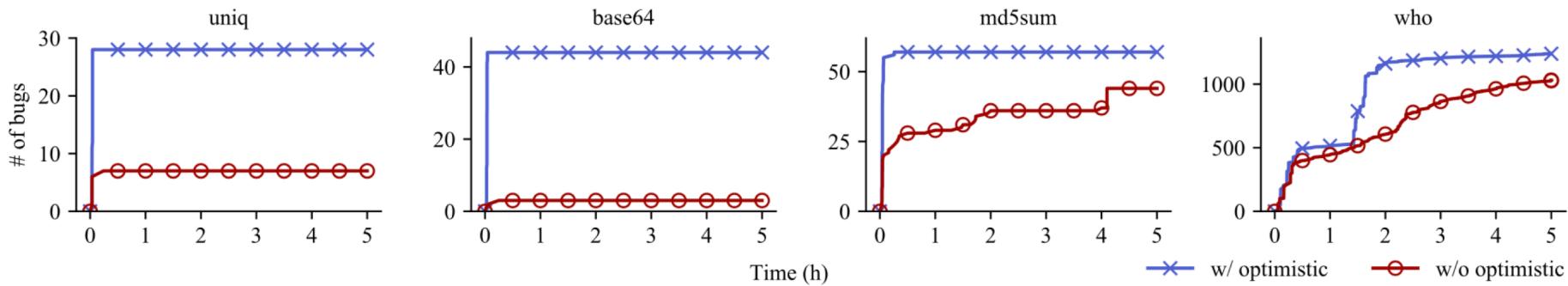
- 結果: QSYMの検査命令数はVEX IRへ近似してもDrillerの1/5程度
  - QSYM: オーバーヘッド削減は検査命令数が削減されたためである

[8] Driller: Augmenting fuzzing through selective symbolic execution [NDSS'16]

[30] Cyber Grand Challenge, <https://www.cybergrandchallenge.com/> [DARPA'16]

# EXPERIMENT (OPTIMISTIC SOLVING)

- QSYM “**with**” vs “**without**” Optimistic Solving
- 検体: LAVA dataset[10]
  - LAVA-1(injected one bug) & LAVA-M (injected multiple bugs)
- 各検体 5時間 の検査、報告されるバグ数を比較

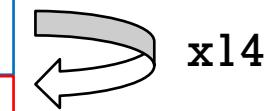


- 結果: Optimistic Solving有りの方が多く(効率的に)バグを発見
  - QSYM: ソルバが制約を解けない状態を解消したのでは

# EXPERIMENT (OPTIMISTIC SOLVING)

- QSYM vs VUzzer[9] (as a state-of-the-art system)
- 検体: LAVA-M dataset[10]
- 検査時間不明、報告されたバグ数を比較

	uniq	base64	md5sum	who
FUZZER	7 (25 %)	7 (16 %)	2 (4 %)	0 (0 %)
SES	0 (0 %)	9 (21 %)	0 (0 %)	18 (39 %)
VUzzer (R)	27 (96 %)	1 (2 %)	0 (0 %)	23 (1 %)
VUzzer (P)	27 (96 %)	17 (39 %)	0 (0 %)	50 (2 %)
QSYM	28 (100 %)	44 (100 %)	57 (100 %)	1,238 (58 %)
Total	28	44	57	2,136



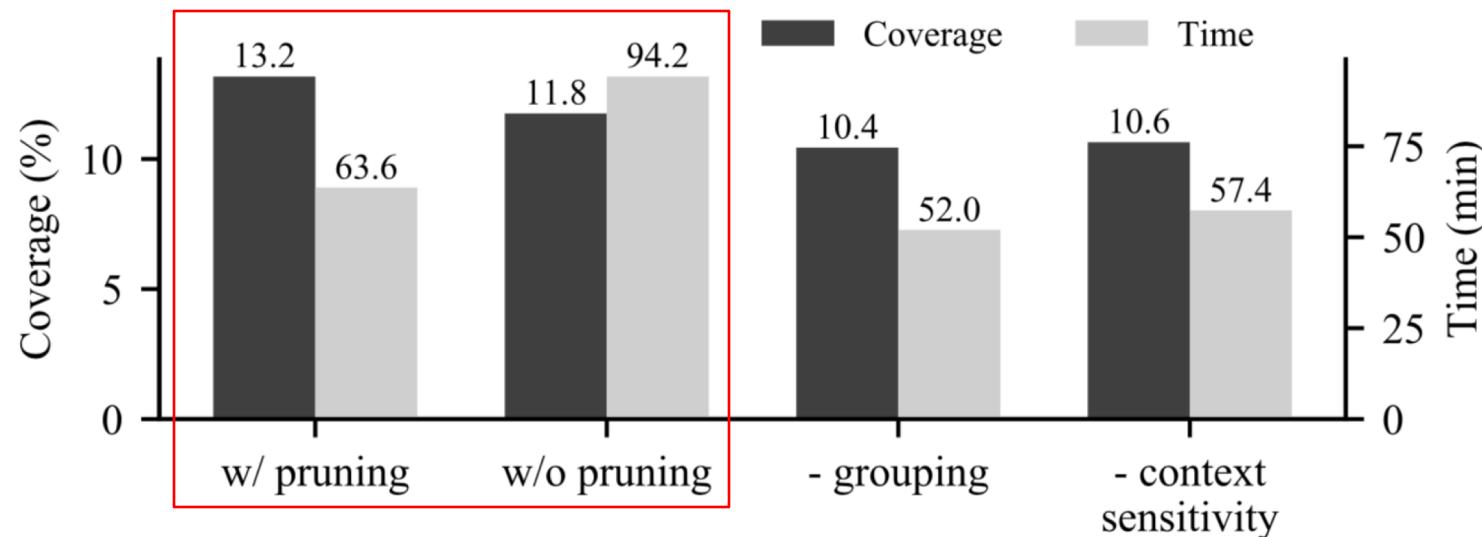
- 結果: 総合してVUzzerの約14倍のバグを発見

[9] VUzzer: Application-aware evolutionary fuzzing [NDSS'17]

[10] LAVA: Large-scale automated vulnerability addition [IEEE SSP'16]

# EXPERIMENT (BBL PRUNING)

- QSYM “with” vs “without” BBL pruning
- 検体: libjpeg, libpng, libtiff, file (利用者が多いオープンソース)
  - 入力: 各検体の最大カバレッジテストケース\*5
    - libjpegだけ4つ、計19個
- 各テストケース5分で切り上げ、カバレッジと時間の合計を比較



- 結果: BBL pruning有りは、無しより高カバレッジかつスケール
  - grouping, context-sensitivityがどちらか欠けても性能は出ない

# ANALYSIS OF NEW BUGS (FFMPEG)

- バグ種: Out-of-bounds read (境界外読み込み)
  - CVE-2017-17081\*
- 原因: 幅(stride)と高さ(h)の境界チェック不足

```
1 // @libavcodec/x86/mpegvideodsp.c:58 (ffmpeg 3.4)
2 if ((ox ^ (ox + dxw))
3     | (ox ^ (ox + dxh))
4     | (ox ^ (ox + dxw + dxh))
5     | (oy ^ (oy + dyw))
6     | (oy ^ (oy + dyh))
7     | (oy ^ (oy + dyw + dyh))) >> (16 + shift)
8 || (dxx | dxy | dyx | dyy) & 15
9 || (need_emu && (h > MAX_H || stride > MAX_STRIDE)))
10 { ... return; }
11 // the bug is here
```

- 発見への特筆事項:
  - QSYM: fuzzerだけでは到達できない複雑な制約を突破

# ANALYSIS OF NEW BUGS (FILE)

- バグ種: Stack overflow
  - CVE-2017-1000249\*
- 原因: descszのチェックがトートロジーになる

```
1 // @src/readelf.c:513 (file 5.31)
2 if (namesz == 4
3     && strcmp((char *)&nbuf[noff], "GNU") == 0
4     && type == NT_GNU_BUILD_ID
5     && (descsz >= 4 || descsz <= 20)) {...}
```

- 発見への特筆事項:
  - 初期入力にELFファイルは含まれていない
    - →QSYM: 自力でテスト生成に成功

# DISCUSSION

---

- Fuzzerについて
  - 今回はQSYM+AFLだったが、他のFuzzerも使用可
    - VUzzer[9]とかAFLFast[34]とか
  
- 制限事項
  - x86\_64特化のため、AMDなどには未対応
  - 浮動小数点命令など、未対応の命令もまだある
  - 自己書き換え、リフレクションには対応できない（はず）

[9] VUzzer: Application-aware evolutionary fuzzing [NDSS'17]

[34] Coverage-based Greybox Fuzzing as Markov Chain [CCS'16]

# CONCLUSION

---

- **QSYM: Hybrid Fuzzing**用に調整された**Concolic Executor**
  - 現実の複雑なアプリケーションでも解析がスケール
- 実験:
  - QSYM vs Driller → wins!
    - in DARPA CGC dataset
  - QSYM vs VUzzer → wins!
    - in LAVA-M dataset
- 8つのアプリケーションから13個の未知のバグを発見
  - AFL, OSS-Fuzzで検査済みの検体