SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

Boosted Symbolic Execution for Software Reliability and Security

Qualifying Exam by Haoxin TU

November 18, 2021

Outlines

- Background
  - What are software **reliability** and **security**?
  - What is **symbolic execution**? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Outlines

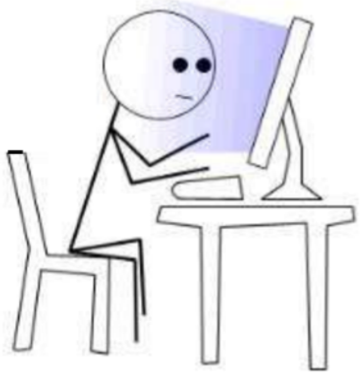
SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

- Background
  - What are software **reliability** and **security**?
  - What is **symbolic execution**? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

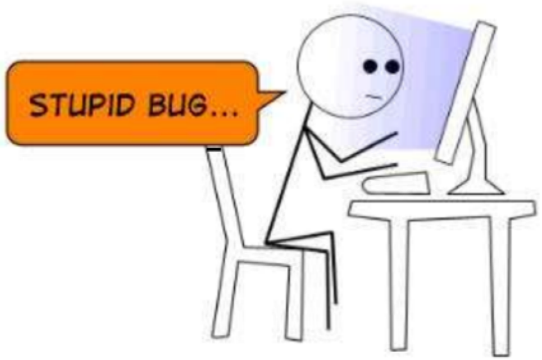
Background: software is everywhere

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

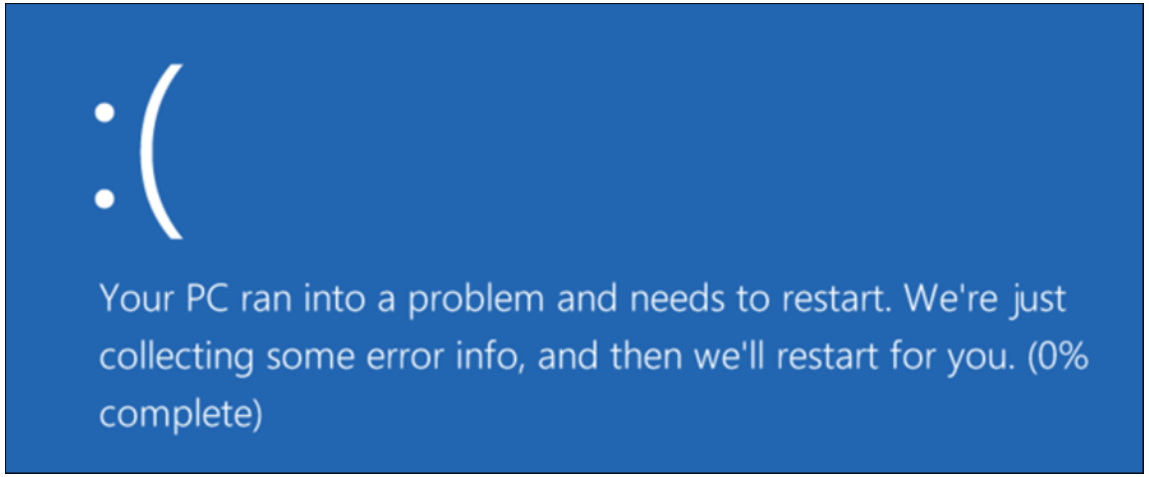
Programs are still written by humans, and will be written by humans

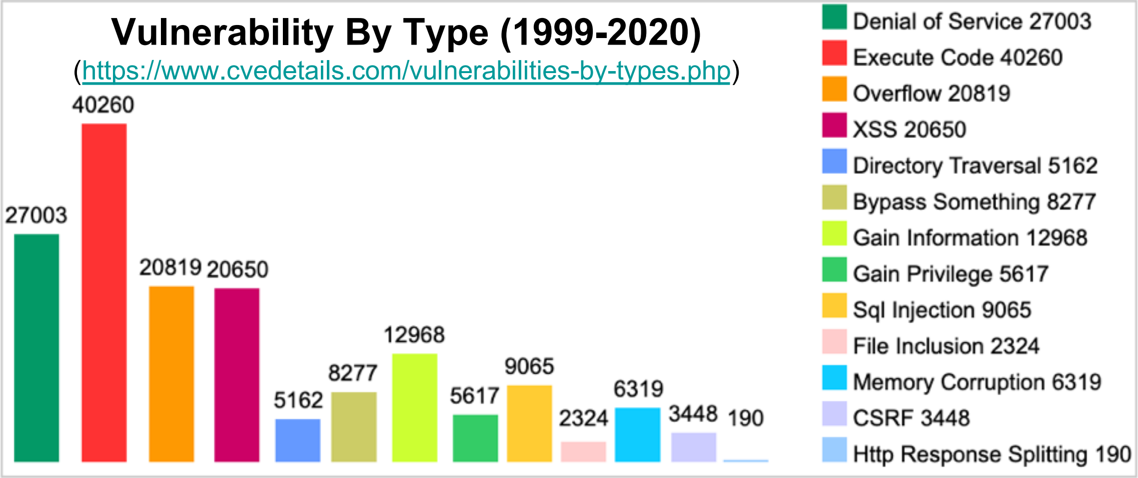


To Err is Human → Software Bugs



Background: bugs are always terrible





A crash

Even worse …

Security flaws

In short, bugs degrade reliability and security of software!

Outlines

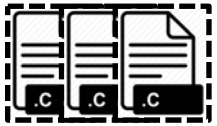
- Background
  - What are software **reliability** and **security**?
  - What is **symbolic execution**? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Background: reliability

- **What is software reliability?**
  - The extent to which software performs intended functions without a **failure (bug)**

A failure (bug) is always triggered by an input

35231+15200-2055
45*11112121
…



GCC

LLVM

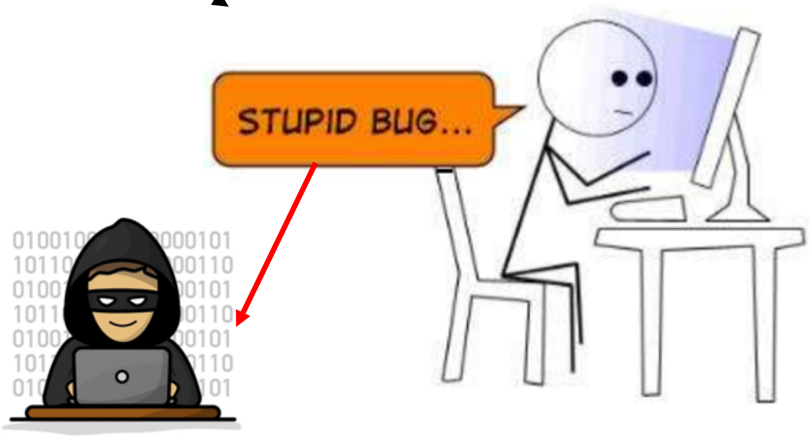
What kinds of inputs should we generate to trigger bugs?

(Depends on different types of software under test)

Background: security

- **What is software security?**
  - The extent to which software **continue** to function correctly under **malicious attacks**

From improving software reliability to security



- **<u>Find</u> bugs**

- **<u>Find</u> important bugs and <u>prove</u> them**
- **(An exploitable bug == A vulnerability)**

Outlines

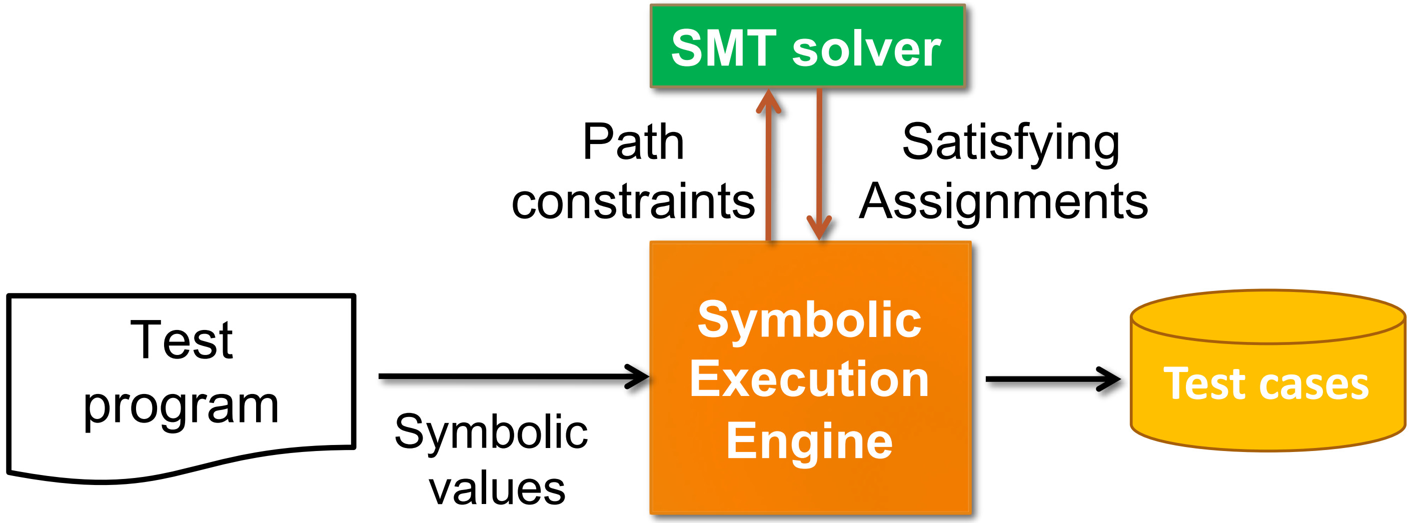
School of Information Systems

- Background
  - What are software **reliability** and **security**?
  - What is **symbolic execution**? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Background: symbolic execution (1/4)

- **What is symbolic execution?**
  - Proposed in 1976*, one of the most popular program analysis techniques, which scales for many **software testing** and **computer security** applications

- **Key idea**



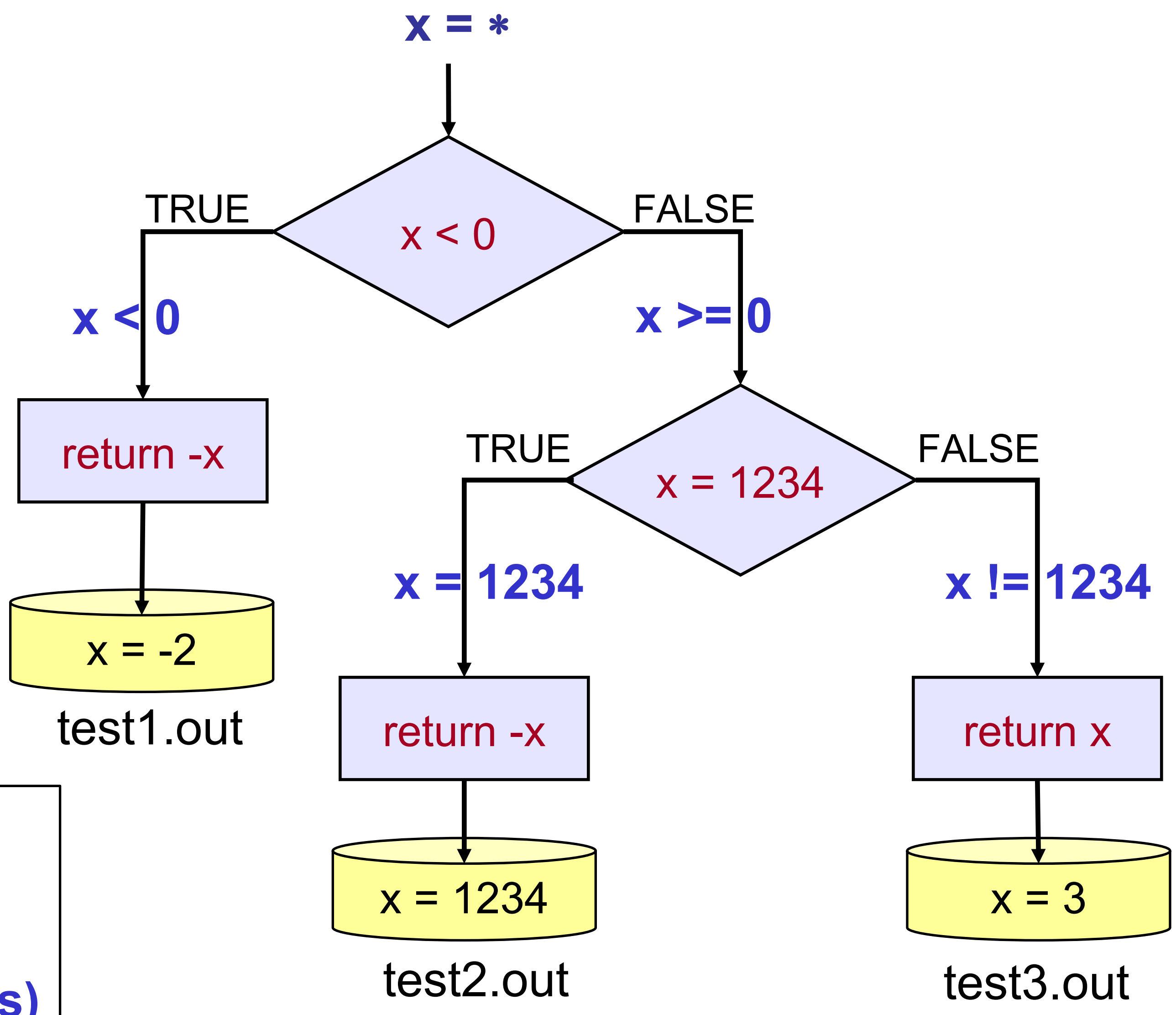
Symbolic **E**xecution (referred to as **SE**)

*James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394.

Background: symbolic execution (2/4)

- A toy example

```
int bad_abs(int x)
{
    if (x < 0)
        return –x;
    if (x == 1234)
        return –x;
    return x;
}
```

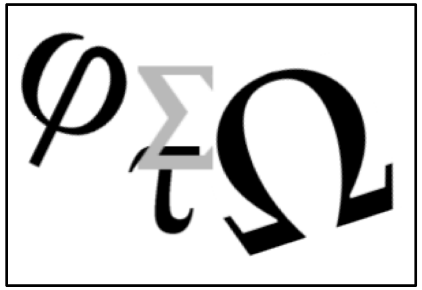


x < 0; **x >= 0 && x = 1234;** **x >= 0 && x != 1234;** **(path constraints)**	⇨	**test1.out** **test2.out** **test3.out** **(test cases)**

Background: symbolic execution (3/4)

❑ How could that work?

- Execute the program with symbolic inputs
- Represent *equivalent* ***execution paths*** with ***path constraints***
- **Solve path constraints** to obtain one representative input that exercises the program to go down that specific path



Path constraints

Z3 STP
CVC4

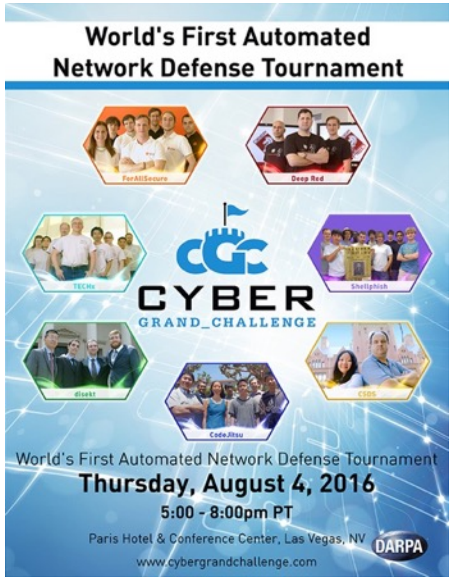
Constraint Solver

Worked!

Background: symbolic execution (4/4)

- **Why we need it?**

- **Reason 1 : Software is unreliable and unsecure**
  - Advanced software testing and verification approaches should be used
- **Reason 2 : Symbolic execution is a promising approach**
  - Has been used in many domains
    - high-coverage test generation, automated debugging, automated program repair, exploit generation, wireless sensor networks, online gaming, …
  - Has been used in many program languages
    - C/C++, C#, Java, Python, JavaScript, .Net, Ruby, …



https://www.darpa.mil/news-events/cyber-grandchallenge

- **Milestone: DARPA Cyber Grand Challenge (CGC)**
- **Ability** of each team:
  - Automatic **vulnerability finding, patching, and exploit generation** at run-time

Symbolic execution is an integral part in the approaches of **TOP 3** winning teams!

Outlines

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

- Background
  - What are software **reliability** and **security**?
  - What is **symbolic execution**? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Background: types of SE (1/3)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

❑ Static SE and dynamic SE

- **Static (classic SE)**
  - Fully symbolic execution
- **Practical issue**:
  - Constraint solver limitations
    - dealing with complex path constraints

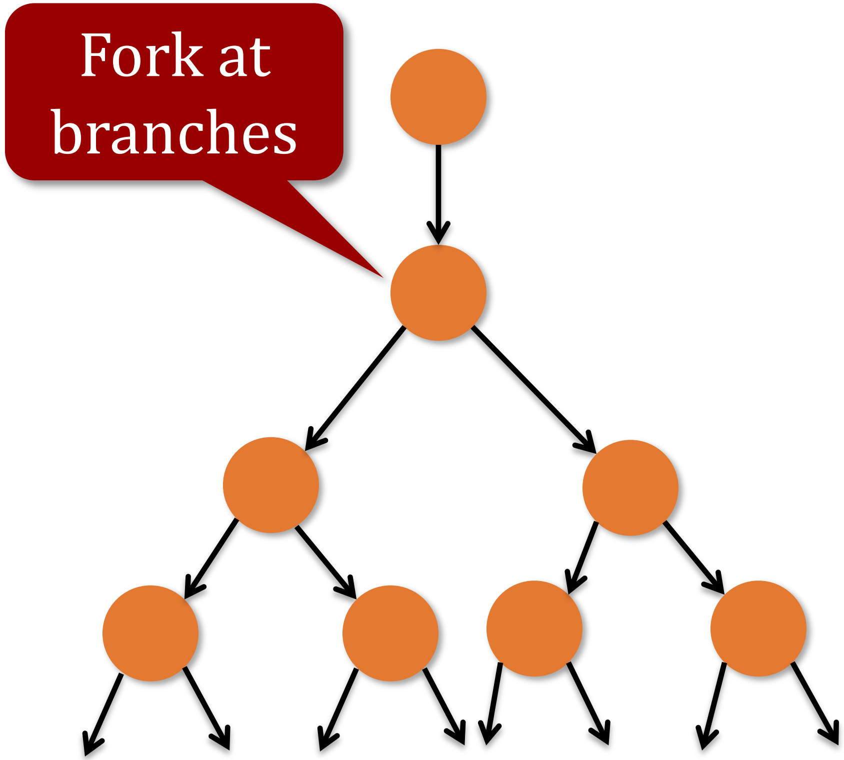
- **Dynamic (modern SE)**
  - Mix concrete and symbolic execution
  - Also called concolic execution
- **Benefits**:
  - More effective
  - More practical

The ability of constraint solver improved greatly!

Background: types of SE (2/3)

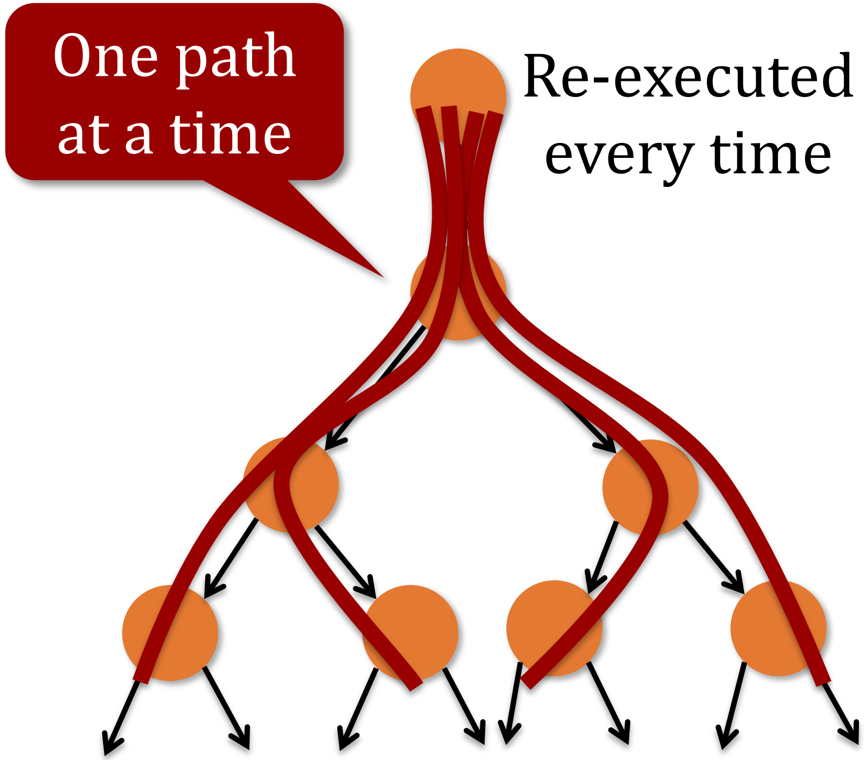
- Online SE and Offline SE

Online



- Main issue: **Hit Resource Cap**

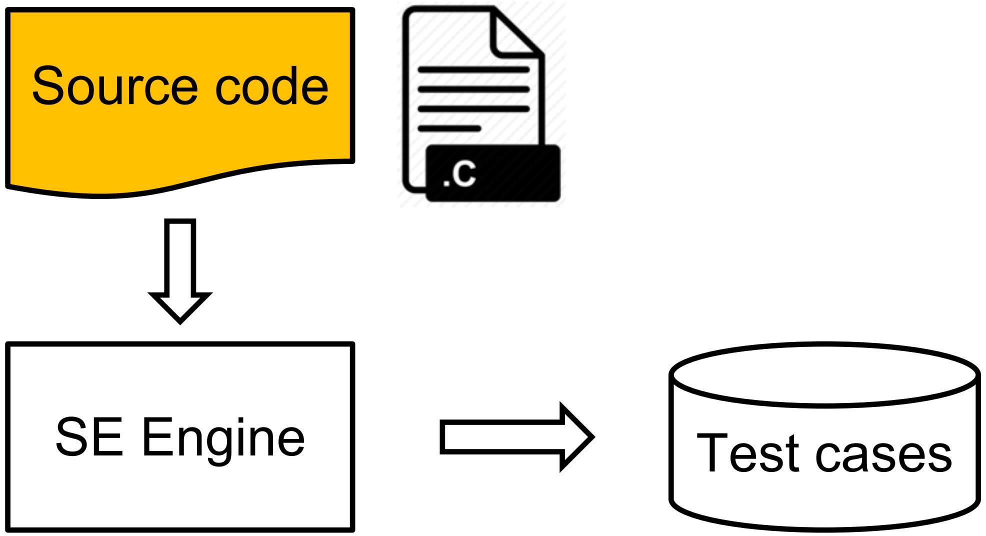
Offline

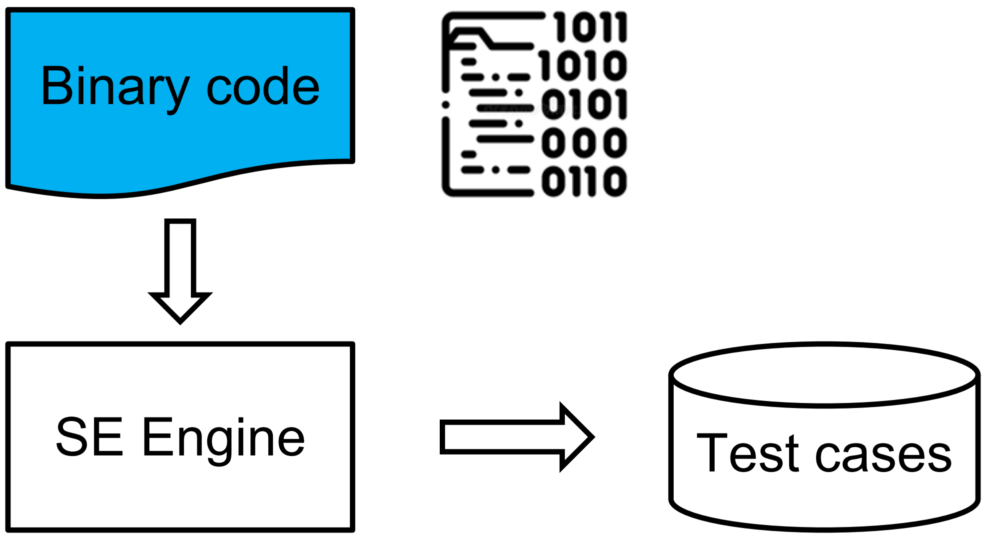


- Main issue: **Inefficient**

Background: types of SE (3/3)

- Source code-based SE and binary-based SE





Outlines

- Background
  - What are software **reliability** and **security**?
  - What is **symbolic execution**? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

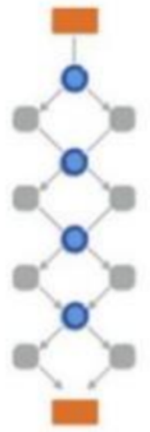
Main challenges: path explosion (1/5)

SMU Singapore Management University | School of Information Systems

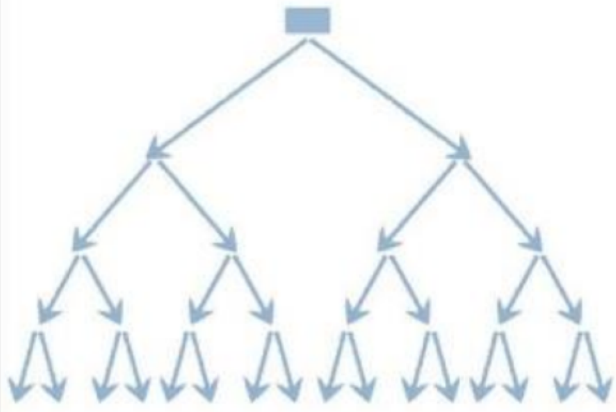
❑ **How does symbolic execution deal with path explosion?**

```
void process(char input[3]) {
    int counter = 0;
    if (input[0] == 'a') counter++;
    if (input[1] == 'b') counter++;
    if (input[2] == 'c') counter++;
    if (counter >= 3) success();
    error();
}
```

- Exponentially many execution paths



4 conditional nodes



16 (2^4) execution paths

- **Possible solutions**
  1. Random search (DFS and BFS)
  2. Coverage-guided search
     - consider new coverage

Seo, Hyunmin, and Sunghun Kim. "How we get there: a context-guided search strategy in concolic testing." Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014.

Main challenges: memory modeling (2/5)

SMU Singapore Management University | School of Information Systems

❑ How does the engine handle symbolic loads or symbolic writes?

```
1. int array [N] = { 0 };
2. array [i] = 10; // i symbolic
3. assert(array[j] != 0); // j symbolic
```

- **Possible solutions**
  1. Fully symbolic
     - consider **any possible** outcome ⟹ **N** states, accurate but not scale
  2. Fully concrete
     - consider **one possible** outcome ⟹ **1** state, scale but not accurate
  3. Partial symbolic and concrete
     - concretize writes,
     - concretize loads when hard ⟹ **K** states, scale but (in) accurate

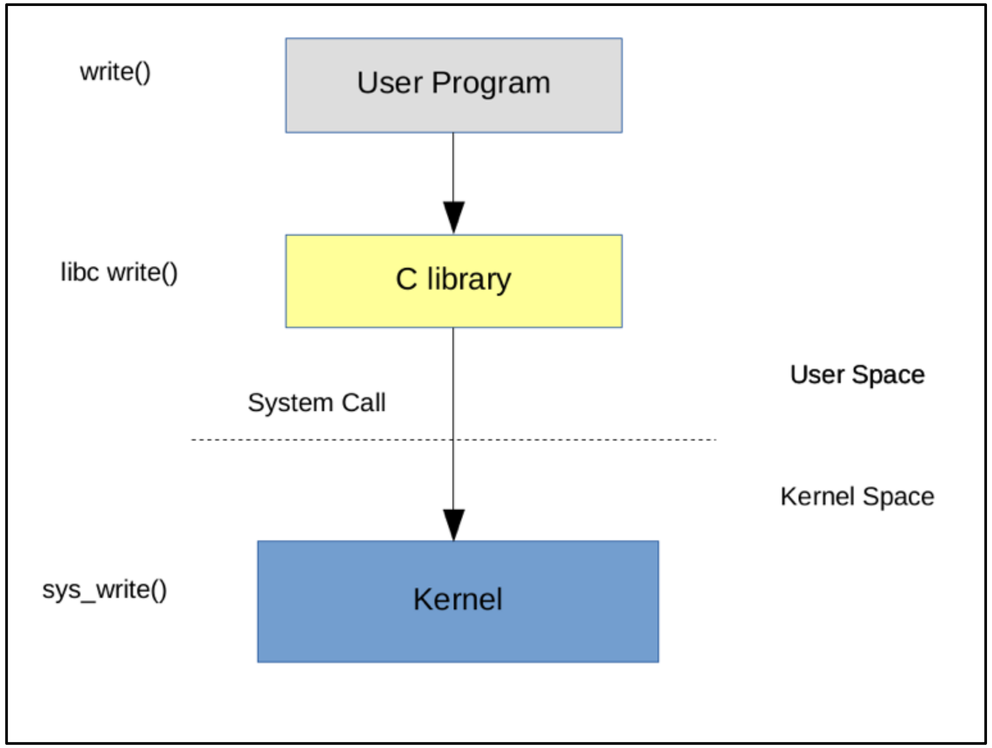
Main challenges: environment modeling (3/5)

SMU Singapore Management University | School of Information Systems

❑ How does the engine handle interactions across the software stack?

```
int main(argc, argv) {
  if (argc == 0) {
      ...
  }
  p = malloc(...);
  if (p == NULL) {
      ...
  }
  ...
}
```

Run

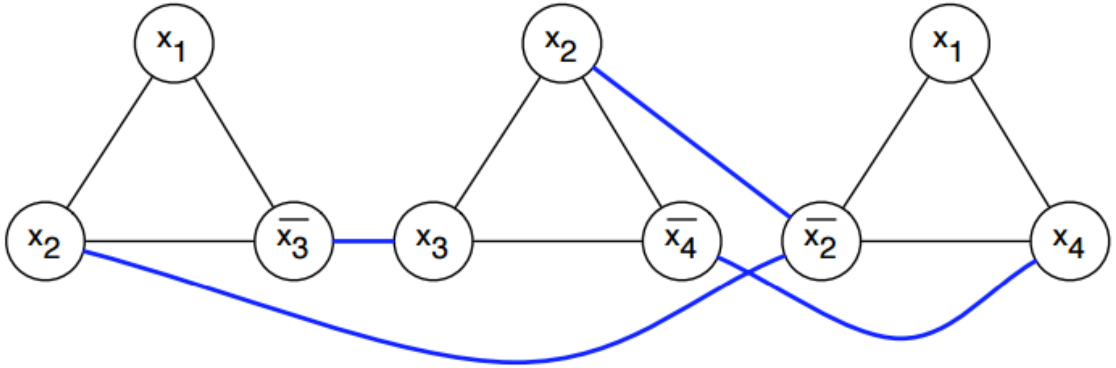


- **Possible solutions**
  1. Fully modeling the environment
  2. Partially modeling the environment
  3. Native execution

Main challenges: constraint solving (4/5)

❑ How does a constraint solver handle complex constraints?

$$(x_1 \lor x_2 \lor \overline{x_3}) \land (x_2 \lor x_3 \lor \overline{x_4}) \land (x_1 \lor \overline{x_2} \lor x_4)$$



SAT or UNSAT?

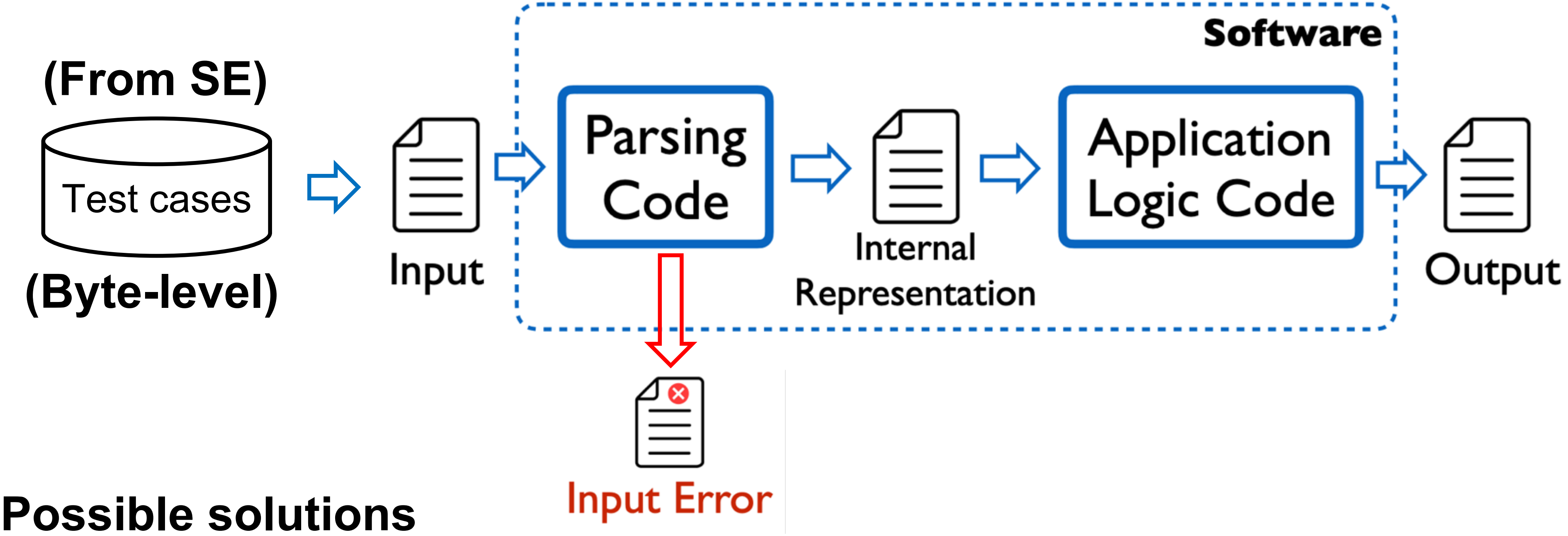
- **Bottleneck**
  1. NP Complete problem
     - (although practical in practice)
  2. Dominates the runtime

- **Possible solutions**
  1. Irrelevant constraint elimination
  2. Incremental solving
  3. Caching

Main challenges: test-case generation (5/5)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

❑ **How does symbolic execution generate structured test cases?**



- **Possible solutions**
  1. Grammar-based generation
     - Use grammar specification to guide generation
  2. Program mutation
     - Modifying existing programs

Outlines

- Background
  - What are software **reliability** and **security**?
  - What is **symbolic execution**? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Related work (overall picture)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

Boosting symbolic execution

Source code-based SE engine

- KLEE (OSDI’ 08)
- PSPA (FSE’ 20)
- SymCC (USENIX Security’20)

Binary-based SE engine

- S2E (ASPLOS’ 11)
- Angr (S&P’ 16)
- SymQEMU (NDSS’ 21)

SE for structured test-case generation

- Test generation using symbolic grammars (FSE’ 07)
- Grammar-based white-box fuzzing (PLDI’ 08)
- Grammar-agnostic symbolic execution (FSE’ 21)

SE for vulnerability detection

- Driller (NDSS’ 16)
- QSYM (USENIX Security’ 18)
- SAVIOR (S&P’ 20)

SE for Automatic Exploit Generation (AEG)

Stack-overflow based

- AEG (NDSS’ 11)
- Mayhem (S&P’ 12)

Heap-overflow based

- Revery (CCS’ 18)

Outlines

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

- Background
  - What are software **reliability** and **security**?
  - What is **symbolic execution**? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Boosting SE : KLEE (OSDI'08)

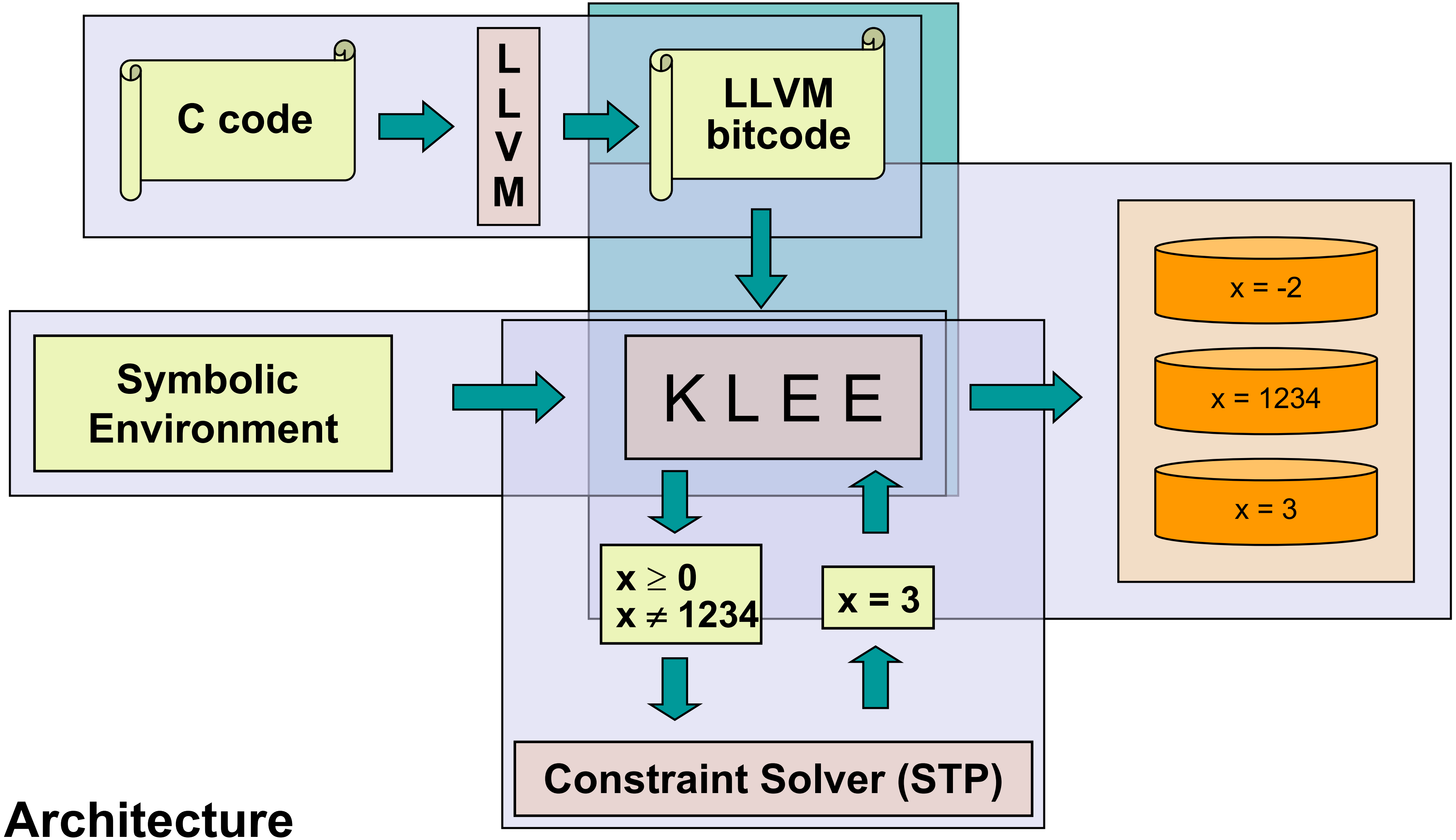
❑ Problem: Testing System Code Is Hard

- **Solution**
  – Based on ***symbolic execution*** and ***constraint solving*** techniques

- KLEE aims to resolve **three scalability challenges**
  1. Exponential number of paths
     - Random path search
     - Coverage-optimized search
  2. Expensive constraint solving
     - Eliminating irrelevant constraints
     - Caching solution
  3. Interaction with environment
     - Support for symbolic command line arguments, files, links, pipes, etc.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008." **KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs**". In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, USA, 209–224.

Boosting SE : KLEE (OSDI'08)



- **KLEE Architecture**

Boosting SE : KLEE (OSDI'08)

❑ Results are promising

- Automatically generate high coverage test suites
  - Over 90% on average on ~160 user-level apps
- Find deep bugs in complex systems programs
  - Including higher-level correctness ones

- **Pros**
  - ✓ High coverage grantee
  - ✓ Good bug-finding capability
- **Cons**
  - Path exploration strategy is simple
  - Lack of support symbolic write/read, float point, etc.

Boosting SE

Approaches	Main ideas	Pros and cons
KLEE(OSDI' 08)	1. Random and Coverage-optimized search 2. Eliminating irrelevant constraints and caching 3. Support for environment modeling	✓ High code coverage and good bug-finding capability ○ Search strategies are simple ○ Lack of support (e.g., float point)
KLEE was improved …		
S2E (ASPLOS' 11)	1. Selective symbolic execution 2. Relaxed execution consistent model	✓ Scale to testing large real systems ○ High overhead
Angr (S&P' 16)	1. Reproduce many existing approaches in offensive binary analysis in a coherent framework 2. Present the different analyses and the challenges	✓ A unified framework for effective binary analysis ○ High overhead (interpreting)
SymCC (USENIX Security' 20) **SymQEMU**(NDSS' 21)	1. Compilation-based (rather than interpreting) symbolic execution for source code/binary 2. Perform the instrumentation on the IR level (Programming language independent)	✓ Fast symbolic execution ✓ Architecture independent and low implementation complexity ○ Offline (Inefficiency issue)

Outlines

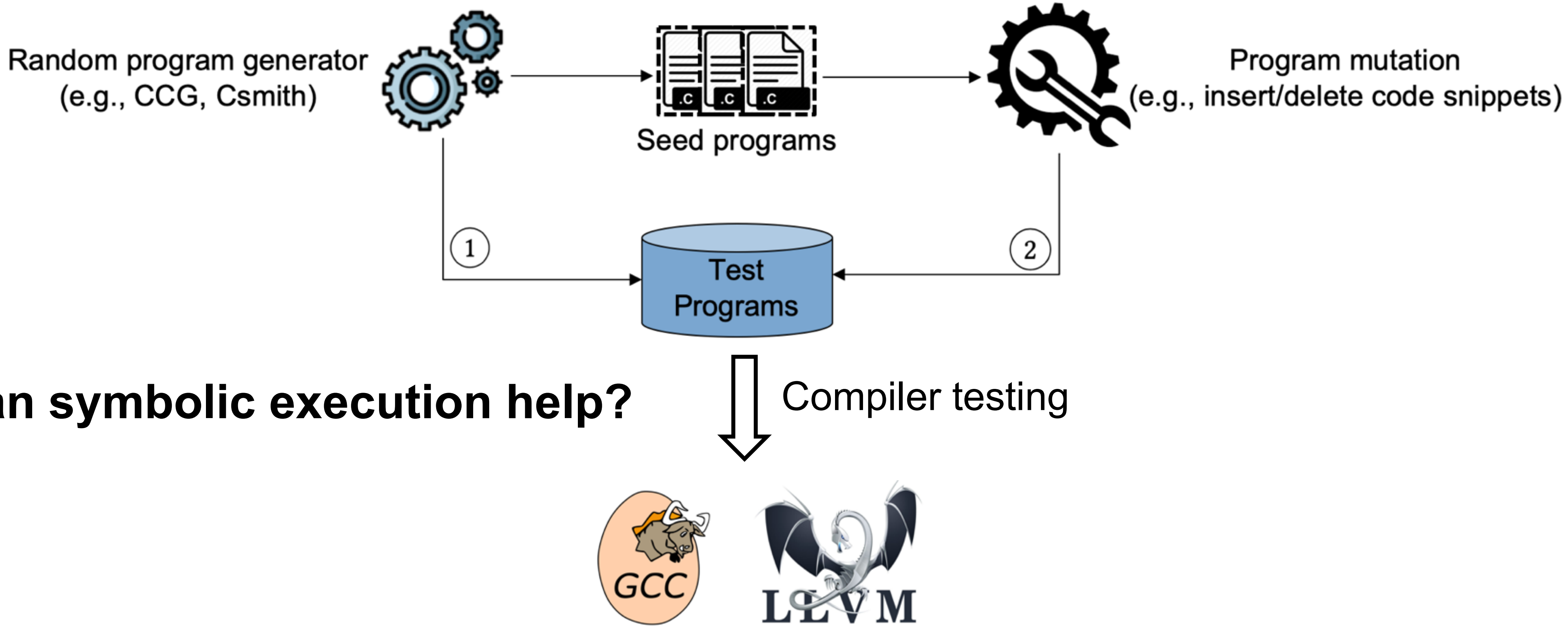
SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

- Background
  - What are software reliability and security?
  - What is symbolic execution? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Improving reliability of software

- Target software which needs structured test inputs (e.g., compilers)

- How to generate valid test programs for compiler testing?



Can symbolic execution help?

Improving reliability of software

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

CESE (FSE' 07)

- **Main idea**
  - Uses symbolic grammars that balance the random enumeration test generation and directed symbolic test generation

1. **Grammar for *SimpCalc* inputs**

$$
\begin{array}{lcl}
\text{Expressions } e & ::= & (e) \mid e * e \mid e/e \mid e \,\%\, e \mid e + e \mid e - e \\
 & & \mid e \vee e \mid e \wedge e \mid -e \mid l \mid n \\
\text{Letters } l & ::= & [a - zA - Z] \\
\text{Numbers } n & ::= & [0 - 9]
\end{array}
$$

"11+11" ✓

"11+1x" ✗

2. **Symbolic grammar**

$$
\begin{array}{lcl}
\text{Letters } l & ::= & \alpha \\
\text{Numbers } n & ::= & \beta
\end{array}
$$

R. Majumdar and R.-G. Xu, "**Directed test generation using symbolic grammars**," in The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers, New York, NY, USA, Sep. 2007, pp. 553–556.

Improving reliability of software

❑ CESE (FSE' 07)

- **Pros**
  - ✓ Generate structured test cases
  - ✓ Improve the code coverage compared to existing single random testing or symbolic execution
- **Cons**
  - Limited scope
  - Need grammar specification

Improving reliability of software

Approaches	Main ideas	Pros and cons
CESE (FSE'07)	1. Combine the advantage of selective enumerative generation with symbolic execution 2. The use of symbolic grammars that balance the two competing requirements	✓ Achieves better coverage on structured test cases ○ Limited scope ○ Need grammar specification
CESE was improved …		
Grammar-based fuzzing (PLDI'08)	1. Generation of higher-level symbolic constraints 2. A custom constraint solver that solves constraints on symbolic grammar tokens.	✓ Applicable to large software (e.g., JavaScript interpreter) ○ Need grammar specification
Grammar-agnostic SE (ISSTA'21)	1. Symbolize tokens instead of input bytes 2. Collecting the byte-level constraints of token values 3. Token symbolization and constraints solving	✓ No need grammar ✓ Achieves better coverage and speedups ○ Limited scope (simple Java)

Outlines

- Background
  - What are software reliability and security?
  - What is symbolic execution? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Improving security of software

- Question: Given a program, how to find vulnerabilities and generate exploits for them automatically?

Vulnerability Detection + Automatic Exploit Generation

- Random testing (Fuzzing)
  - Inefficiency
- Symbolic execution
  - Path explosion
- Hybrid testing
  - Combine fuzzing and symbolic execution

- Stack overflow based
  - Restore stack layout
- Heap overflow based
  - Restore heap layout

Outlines

- Background
  - What are software reliability and security?
  - What is symbolic execution? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Improving security of software

- Driller (NDSS' 16)

Fuzzing vs Symbolic execution

```
x = input()

def recurse(x, depth):
  if depth == 2000
    return 0
  else {
    r = 0;
    if x[depth] == "B":
      r = 1
    return r + recurse(x
[depth], depth)

if recurse(x, 0) == 1:
  print "You win!"
```

Fuzzing Wins

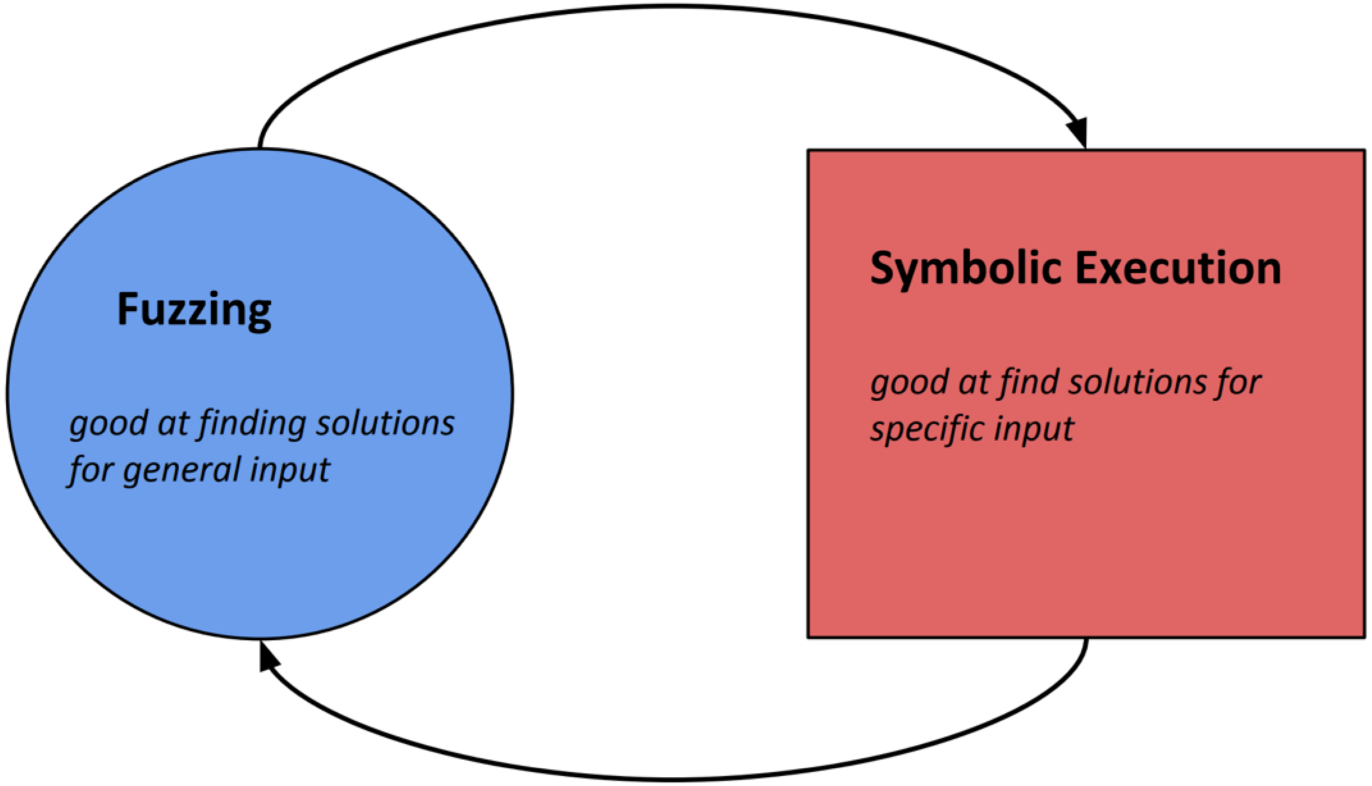
```
x = int(input())
if x >= 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Symbolic execution Wins

Improving security of software

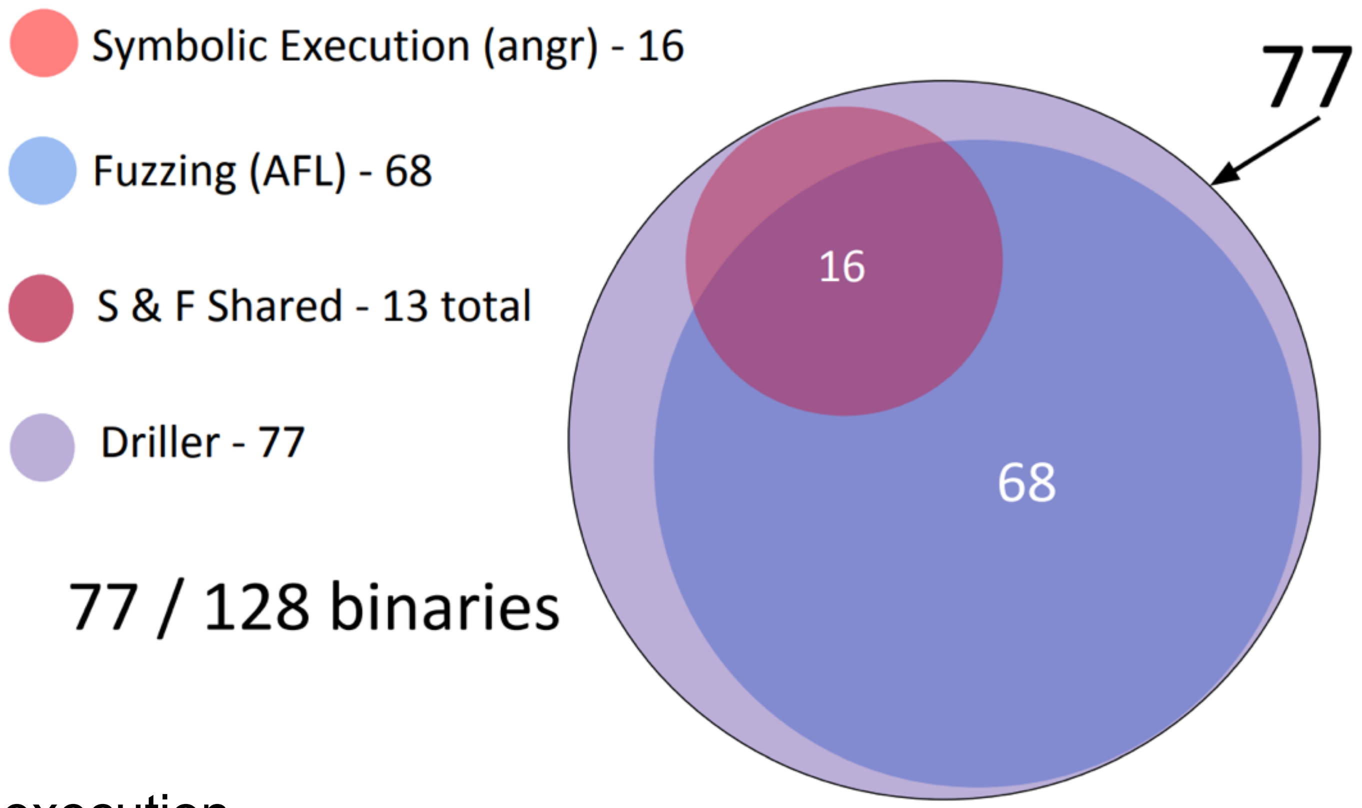
SMU Singapore Management University | School of Information Systems

- **Driller** (NDSS' 16)
- **Main idea**
  - Combine **fuzzing** and **symbolic execution** to leverage their strengths while mitigating their weakness



Improving security of software

❑ Results



Binary Crashes per Technique

- **Pros**
  - ✓ Complement fuzzing and symbolic execution
  - ✓ Explore deep code region
- **Cons**
  - Performance issue inherited from symbolic execution

Improving security of software

Approaches	Main ideas	Pros and cons
Driller (NDSS'16)	1. Combine fuzzing and symbolic execution 2. Fuzzing finds solutions for general conditions 3. SE finds solutions for specific conditions	✓ Complement fuzzing and symbolic execution ✓ Could identify deep bugs ○ Performance issue
Driller was improved …		
QSYM (USENIX Security' 18)	1. Tightly integrate the symbolic emulation with the native execution into hybrid fuzzing 2. Optimistically solve constraints and prune uninteresting basic blocks	✓ Fast symbolic execution through efficient emulation. ○ High implementation effort ○ Coverage-guided search
SAVIOR (S&P '20)	1. Replace the coverage-centric design 2. Enhance hybrid testing with bug-driven prioritization and bug-guided verification	✓ Improve vulnerability detection capability ○ Incomplete bug labeling

Outlines

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

- Background
  - What are software reliability and security?
  - What is symbolic execution? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

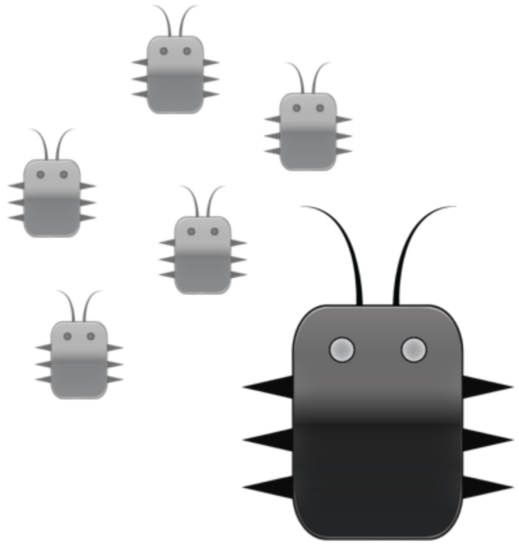
Improving security of software

❑ What is Automatic Exploit Generation (AEG)?

```
I = input();
if (I < 42)
  vuln();
else
  safe();
```

Program

AEG



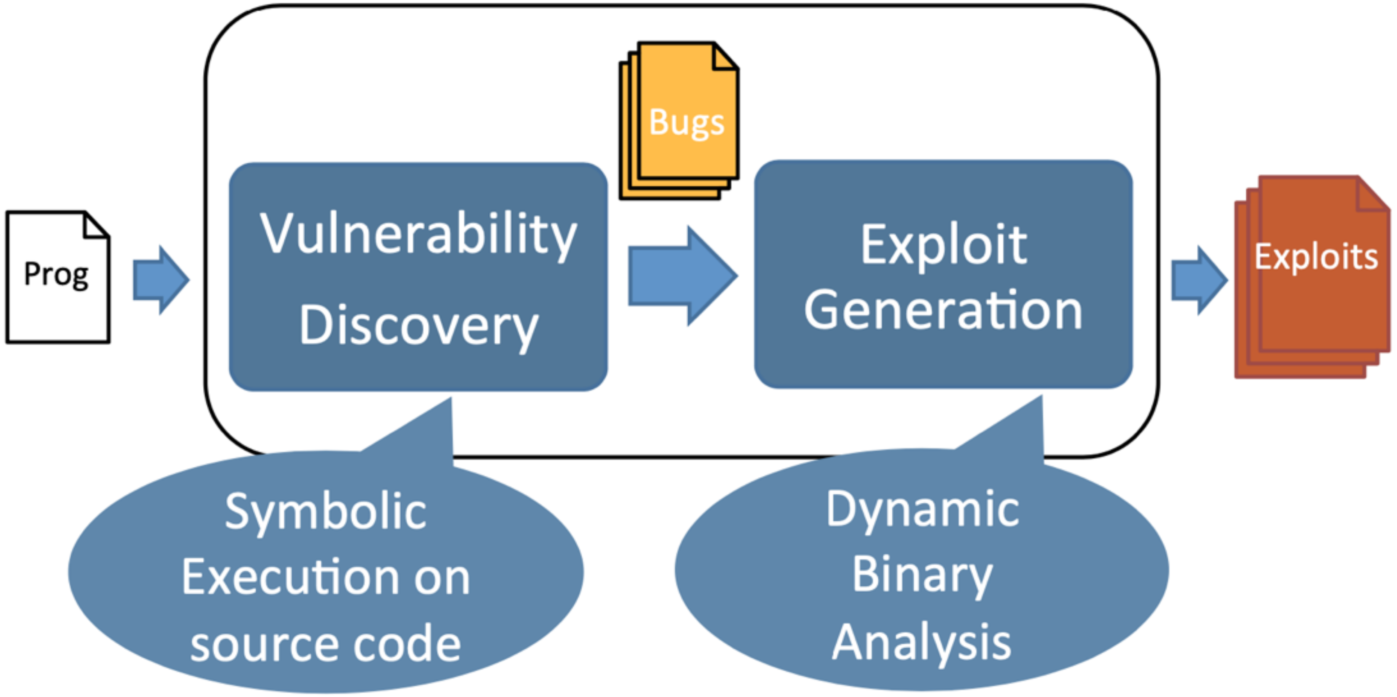
Exploits

Automatically Analyze vulnerabilities & Generate Exploits

Improving security of software

AEG (NDSS'11)

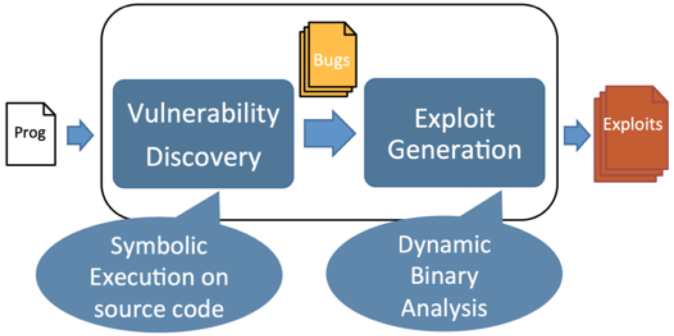
- **Problem**
  - How to make AEG more practical?
- **Solution**



T. Avgerinos, S. K. Cha, B. L. Tze Hao, and D. Brumley. "**AEG: Automatic Exploit Generation".** In Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11), 2011.

Improving security of software

❑ AEG (NDSS'11)



- **Symbolic execution (Preconditioned)**
  - Goal: Discover "buggy" predicates
  - Key insights:
    - Exploring: only explore buggy paths (Fast)
    - Searching: buggy (most likely to exploit)-path-first (Fast still)
      - Search for exploitable path in paths along buggy paths

- **Dynamic binary analysis**
  - Goal: Test exploitability of buggy path
  - Key insight:
    - Generate runtime information and exploit constraints

AEG (NDSS'11) - Results

Name	Advisory ID	Time	Exploit Type	Exploit Class
Iwconfig	CVE-2003-0947	1.5s	Local	Buffer Overflow
Htget	CVE-2004-0852	< 1min	Local	Buffer Overflow
Htget	-	1.2s	Local	Buffer Overflow
Ncompress	CVE-2001-1413	12. 3s	Local	Buffer Overflow
Aeon	CVE-2005-1019	3.8s	Local	Buffer Overflow
Tipxd	OSVDB-ID#12346	1.5s	Local	Format String
Glftpd	OSVDB-ID#16373	2.3s	Local	Buffer Overflow
Xserver	CVE-2007-3957	31.9s	Remote	Buffer Overflow
Aspell	CVE-2004-0548	15.2s	Local	Buffer Overflow
Corehttp	CVE-2007-4060	< 1min	Remote	Buffer Overflow
Exim	EDB-ID#796	< 1min	Local	Buffer Overflow
Socat	CVE-2004-1484	3.2s	Local	Format String
Xmail	CVE-2005-2943	< 20min	Local	Buffer Overflow
Expect	OSVDB-ID#60979	< 4min	Local	Buffer Overflow
Expect	-	19.7s	Local	Buffer Overflow
Rsync	CVE-2004-2093	< 5min	Local	Buffer Overflow

Analyzed **14** applications for 3 hours and generated **16** working exploits

- **Pros**
  - ✓ An end-to-end system for automatic exploit generation
  - ✓ Fast vulnerability discovery and effective exploit generation
- **Cons**
  - Need source code
  - Only stack overflow based
  - Performance issue

Improving security of software

Approaches	Main ideas	Pros and cons
AEG (NDSS'11)	1. Model exploit generation for control flow hijack attacks as a formal verification problem 2. Combine source code and binary level analysis 3. Precondition symbolic execution	✓ An end-to-end system for automatic exploit generation ○ Need source code ○ Only stack overflow based
AEG (NDSS'11) was improved …		
Mayhem (S&P' 11)	1. Hybrid symbolic execution: actively managing execution paths without exhausting memory 2. Index-based memory modeling (Work on binary)	✓ Balance between speed and memory requirements ○ Only stack overflow based
Revery (CCS '18)	1. Search for exploitable states in paths diverging from crashing paths (not in the same path) 2. Generate control-flow hijacking exploits for heap-based vulnerabilities	✓ Target on heap overflows ✓ Improve exploit derivability ○ Limitations of diverging path exploration

Outlines

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

- Background
  - What are software reliability and security?
  - What is symbolic execution? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
    - Symbolic execution for bug detection
  - Towards improving **security** of software
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Research gaps

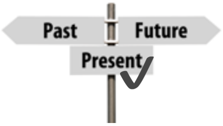
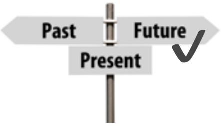
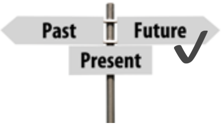
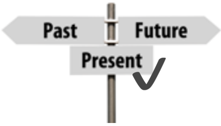
Domains	Existing solutions	Limitations
Boosting symbolic execution	1. Path exploration and memory modeling (KLEE) 2. Scalability and environment model (S2E) 3. Performance (SymCC, SymQEMU)	○ Path exploration: coverage guided or random ○ Lack of security foundations
Structured test case generation	1. Symbolic grammar (CESE) 2. Grammar constraints and costumed solver(PLDI'08) 3. Token-level symbolization (ISSTA' 21)	○ Limited scale of software ○ Well-defined inputs (e.g., C) can not be generated
Vulnerability detection	1. Hybrid fuzzing (fuzzing + SE) (Driller) 2. Symbolic emulation for better performance (QSYM) 3. Bug-driven selection and verification (SAVIOR)	○ Bug-labeling strategy is not complete (only UBSan) ○ Limited scale of software
Automatic exploit generation	1. Exploit generation as formal verification (AEG) 2. Hybrid symbolic execution for efficiency (Mayhem) 3. Target heap overflow and diverging path (Revery)	○ Diverging path exploration strategy is random ○ Limited exploitable types

Outlines

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

- Background
  - What are software reliability and security?
  - What is symbolic execution? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gaps
- Research plans and onging work
- Conclusion

Research plans

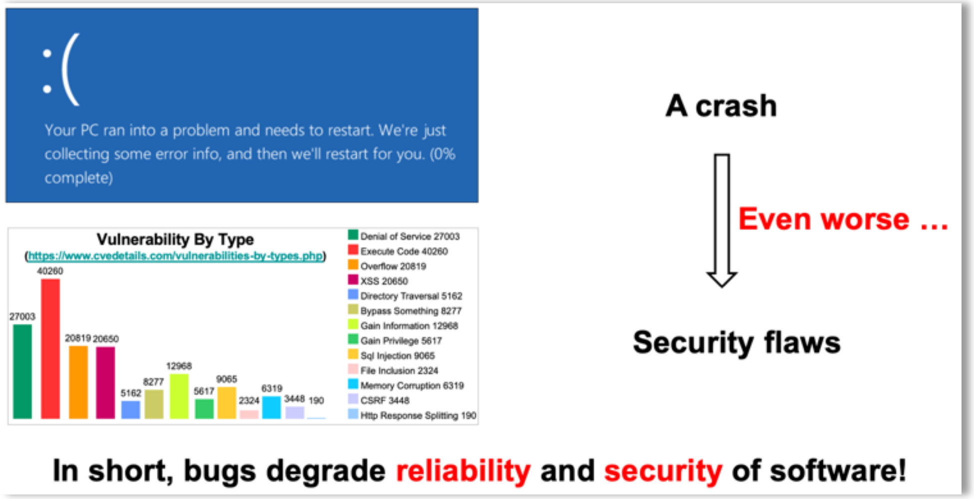
Plans	Highlights	Status
Symbolic dynamic memory allocation for SE	1. Most SE engine models concrete address for dynamic allocated memory 2. Tricky bugs may be triggered by different allocated address; symbolic address can alleviate this problem	➢ Ongoing work 
Grammar-guided test generation for compilers	1. Grammar specifications for large software are usual available (ANTLR supports 100+ grammars) 2. Scalable grammar-guided SE for test case generation	➢ Future work  (More investigation)
Bug-oriented path exploration for SE	1. Path explosion is still a open and unaddressed challenge 2. Exploring buggy execution paths first under limited resource can be useful for effective vulnerability detection	➢ Future work  (More investigation)
Automatic exploit generation	1. Effective and efficient diverging path exploration (using SE rather than fuzzing) 2. Attack targets setting for generating working exploits	➢ Ongoing work 

Outlines

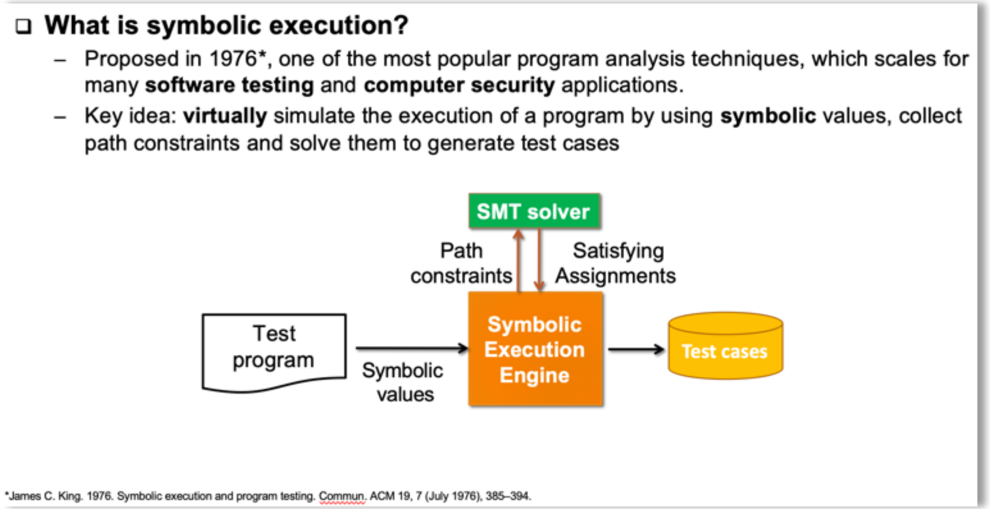
SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

- Background
  - What are software reliability and security?
  - What is symbolic execution? Why we need it?
  - Different **types** of symbolic execution
- Main **challenges** in symbolic execution
- Related work
  - Towards **boosting** symbolic execution
  - Towards improving **reliability** of software
    - Symbolic execution for structured test-case generation
  - Towards improving **security** of software
    - Symbolic execution for vulnerability detection
    - Symbolic exeuction for automatic exploit genenration
  - Research gap
- Research plan and onging work
- Conclusion

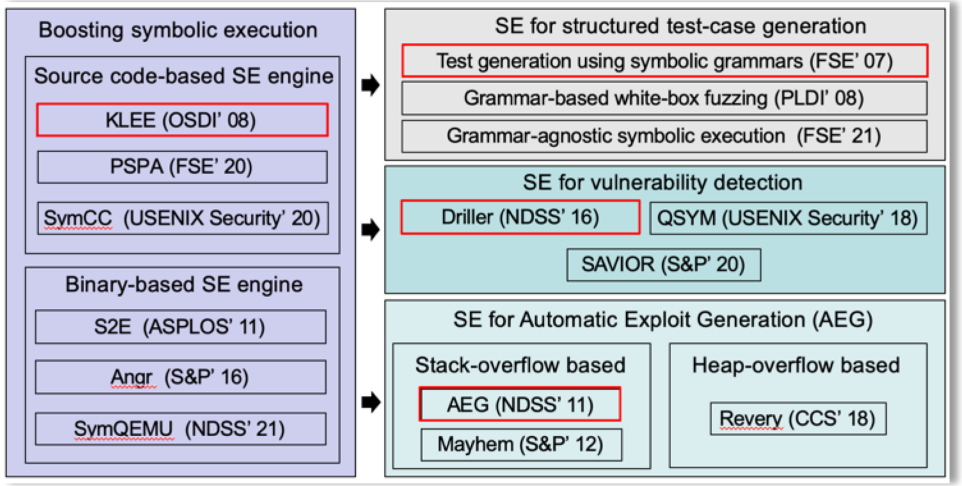
Conclusion



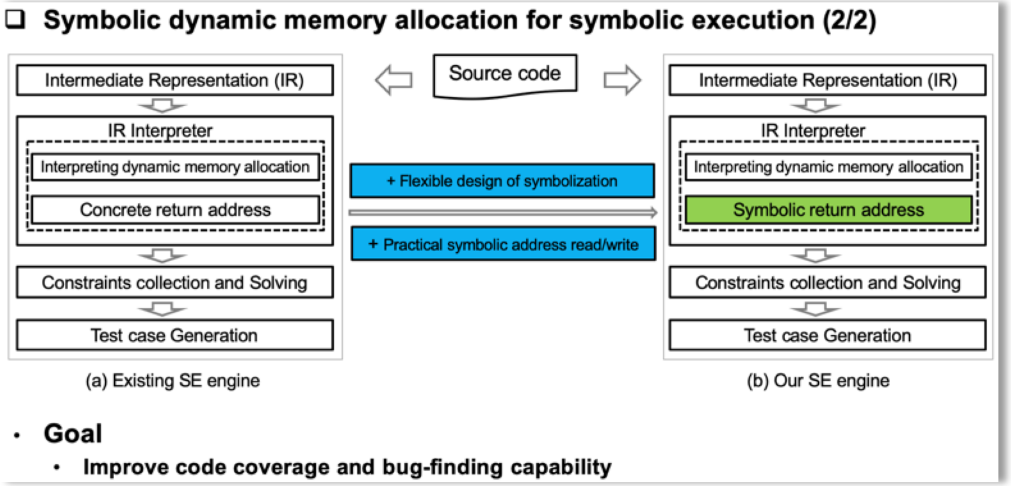
:(
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)
Vulnerability By Type
(https://www.cvedetails.com/vulnerabilities-by-types.php)
Denial of Service 27003
Execute Code 40260
Overflow 20819
XSS 20650
Directory Traversal 5162
Bypass Something 8277
Gain Information 12968
Gain Privilege 5617
Sql Injection 9065
File Inclusion 2324
Memory Corruption 6319
CSRF 3448
Http Response Splitting 190
A crash
Even worse …
Security flaws
In short, bugs degrade reliability and security of software!



What is symbolic execution?
– Proposed in 1976*, one of the most popular program analysis techniques, which scales for many software testing and computer security applications.
– Key idea: virtually simulate the execution of a program by using symbolic values, collect path constraints and solve them to generate test cases
SMT solver
Path constraints
Satisfying Assignments
Test program
Symbolic values
Symbolic Execution Engine
Test cases
*James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394.



Boosting symbolic execution
Source code-based SE engine
KLEE (OSDI' 08)
PSPA (FSE' 20)
SymCC (USENIX Security' 20)
Binary-based SE engine
S2E (ASPLOS' 11)
Angr (S&P' 16)
SymQEMU (NDSS' 21)
SE for structured test-case generation
Test generation using symbolic grammars (FSE' 07)
Grammar-based white-box fuzzing (PLDI' 08)
Grammar-agnostic symbolic execution (FSE' 21)
SE for vulnerability detection
Driller (NDSS' 16)
QSYM (USENIX Security' 18)
SAVIOR (S&P' 20)
SE for Automatic Exploit Generation (AEG)
Stack-overflow based
AEG (NDSS' 11)
Mayhem (S&P' 12)
Heap-overflow based
Revery (CCS' 18)



Symbolic dynamic memory allocation for symbolic execution (2/2)
Source code
Intermediate Representation (IR)
IR Interpreter
Interpreting dynamic memory allocation
Concrete return address
Constraints collection and Solving
Test case Generation
(a) Existing SE engine
+ Flexible design of symbolization
+ Practical symbolic address read/write
Intermediate Representation (IR)
IR Interpreter
Interpreting dynamic memory allocation
Symbolic return address
Constraints collection and Solving
Test case Generation
(b) Our SE engine
Goal
Improve code coverage and bug-finding capability

References

[1] Automated Test Generation: **“A Journey from Symbolic Execution to Smart Fuzzing and Beyond”** (Keynote by Koushik Sen)

[2] Zhide Zhou, Zhilei Ren, Guojun Gao, He Jiang. “**An empirical study of optimization bugs in GCC and LLVM”**. JSS, 2021.

[3] James C. King. 1976. **Symbolic execution and program testing**. Commun. ACM 19, 7 (July 1976), 385–394.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. 2018. **“A Survey of Symbolic Execution Techniques”.** ACM Computer Survey. 51, 3, Article 50 (July 2018), 39 pages.

[5] Seo, Hyunmin, and Sunghun Kim. "**How we get there: a context-guided search strategy in concolic testing**." Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008.” **KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs”.** In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, USA, 209–224.

[7] C. Cadar and K. Sen, “**Symbolic execution for software testing: three decades later**,” Commun. ACM, vol. 56, no. 2, pp. 82–90, 2013.

[8] V. Chipounov, V. Kuznetsov, and G. Candea, “**S2E: a platform for in-vivo multi-path analysis of software systems**,” in Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, New York, NY, USA, Mar. 2011, pp. 265–278.

[9] S. Poeplau and A. Francillon, “**SymQEMU: Compilation-based symbolic execution for binaries**,” presented at the in Proceedings of the 2021 Network and Distributed System Security Symposium, 2021.

[10] Y. Shoshitaishvili et al., “**SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis**,” in 2016 IEEE Symposium on Security and Privacy (SP), May 2016, pp. 138–157.

[11] S. Poeplau and A. Francillon, “**Symbolic execution with SymCC: Don’t interpret, compile!,”** in 29th USENIX Security Symposium, 2020, pp. 181–198.

[12] David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. “**Past-sensitive pointer analysis for symbolic execution**”. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020). New York, NY, USA, 197–208.

References

[13] R. Majumdar and R.-G. Xu, "**Directed test generation using symbolic grammars**," in The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers, New York, NY, USA, Sep. 2007, pp. 553–556.

[14] P. Godefroid, A. Kiezun, and M. Y. Levin, "**Grammar-based whitebox fuzzing**," in Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, Jun. 2008, pp. 206–215.

[15] W. Pan, Z. Chen, G. Zhang, Y. Luo, Y. Zhang, and J. Wang, "**Grammar-agnostic symbolic execution by token symbolization**," in Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Denmark, Jul. 2021, pp. 374–387.

[16] N. Stephens et al., "**Driller: Augmenting Fuzzing Through Selective Symbolic Execution**," presented at the Network and Distributed System Security Symposium, San Diego, CA, 2016.

[17] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. "**QSYM: a practical concolic execution engine tailored for hybrid fuzzing".** In Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18). USENIX Association, USA, 745–761.

[18] Y. Chen et al., "**SAVIOR: Towards Bug-Driven Hybrid Testing**," in 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, May 2020, pp. 1580–1596.

[19] T. Avgerinos, S. K. Cha, B. L. Tze Hao, and D. Brumley. "**AEG: Automatic Exploit Generation".** In Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11), 2011.

[20] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. "**Unleashing Mayhem on Binary Code"**. In Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12). IEEE Computer Society, USA, 380–394.

[21] Y. Wang et al., "**Revery: From Proof-of-Concept to Exploitable**," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto Canada, Oct. 2018, pp. 1914–1927.

- **Acknowledgement**

  Some pictures are adapted from the presentation slides of above references.

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Information Systems

Thank you && Questions?

Boosted Symbolic Execution for Software Reliability and Security

Qualifying Exam by Haoxin TU

November 18, 2021