Utilization and Evolution of KLEE-based Technologies for Embedded Software Testing at Fujitsu

Indradeep Ghosh

Fujitsu Laboratories of America
Sunnyvale, CA, USA

April 20, 2018

Brief Introduction

- Director of Research
  - Fujitsu Labs of America, Sunnyvale, California
  - Lead the “Software Quality and Security Lab” – 10 researchers
  - Have been with Fujitsu 20 years
    - In Silicon Valley time that is close to 200
- PhD, Princeton University, EE
- Worked in
  - Hardware Test (PhD)
  - Hardware Verification/Validation (early 2000s)
  - Software Validation and QA (from 2006)

The Fujitsu Group

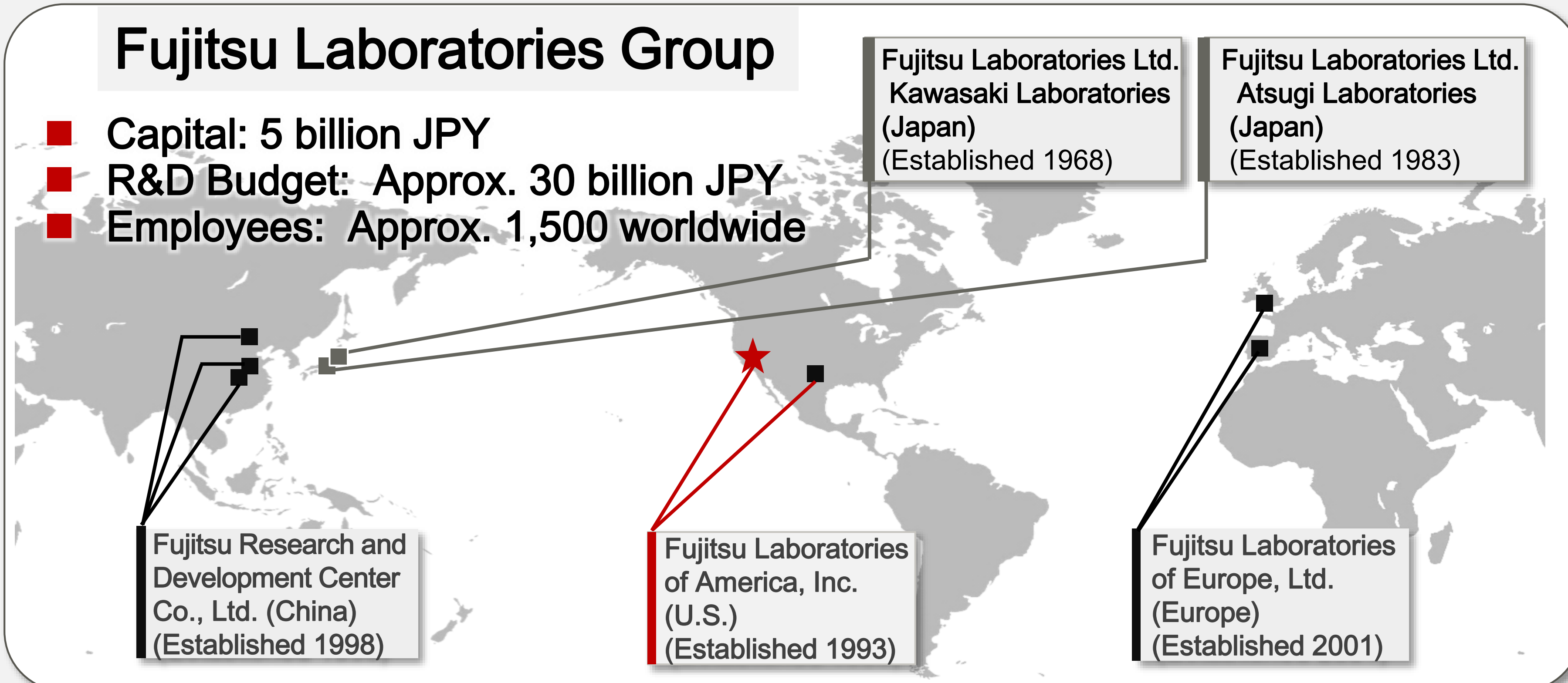
FUJITSU

(70% software related products & services)

FUJITSU (Est. June 1935) Revenue: $41 billion R&D Expenditure: $2.5 billion

Fujitsu Laboratories Group

- Capital: 5 billion JPY
- R&D Budget: Approx. 30 billion JPY
- Employees: Approx. 1,500 worldwide



A rich history
80 years
of Innovation

FORTUNE named Fujitsu as
"one of the World's Most Admired Companies"
for a third consecutive year.

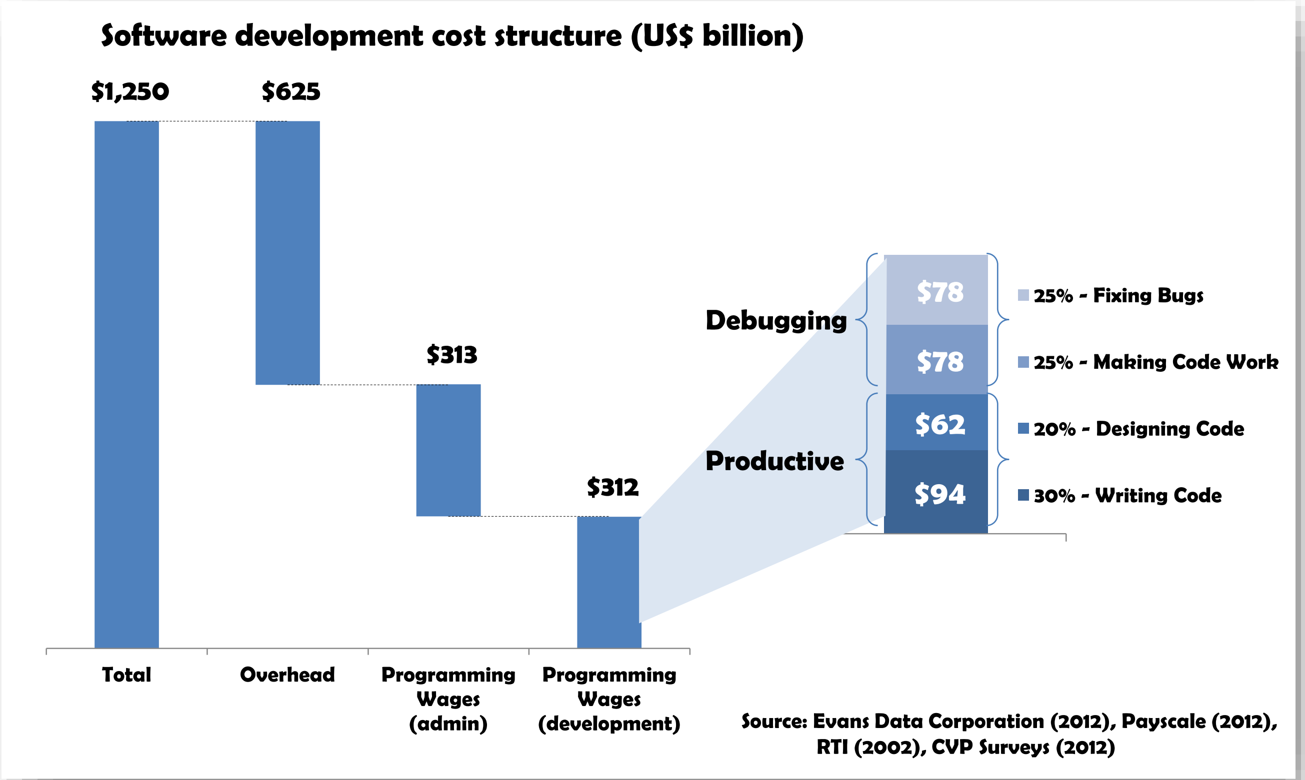
156 thousand employees

On site service
180+ countries

Motivation: Building Software is Expensive

FUJITSU

Source: *T. Britton et al., "Reversible Debugging Software," 2013.*



The global cost of software development (as of 2013) was estimated at US $1.25 Trillion !

- The annual global cost of debugging software is US $312 billion
- Software developers spend ***half their time*** finding and fixing bugs

Tremendous business potential for software developer productivity enhancement.

The Cost of (hardware & software) Bugs

FUJITSU

$$\frac{4{,}195{,}835}{3{,}145{,}727} = 1.333820449136241002$$

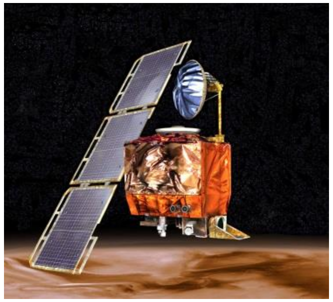
1.333739068902037589

Intel Pentium FDIV Bug – 1994

Total Cost: $475 million

Mars Climate Orbiter- 1999

Destroyed due to software on the ground generating commands in pound-force (lbf), while the orbiter expected newtons (N).



Mission Cost: $327.6 million

The global cost of debugging software (as of 2013) had risen to $312 billion annually. The research found that, on average, software developers spend 50% of their programming time finding and fixing bugs.

- Cambridge University Research Study[1]

[1] http://www.prweb.com/releases/2013/1/prweb10298185.htm

Heartbleed OpenSSL Vulnerability

April 2014

"The Heartbleed bug has likely cost businesses tens of millions of dollars in lost productivity" (Reuters)

Software Testing ⇒ Software Quality

FUJITSU

- Testing is a dominant element in the processes to establish confidence in the correctness of software
- High-quality test suites are notoriously laborious to develop
  - **Product code of SQLite** version 3.7.17 consists of **81.3 KSLOC**, while its **test suite** is **91,421.1 KSLOC**, *i.e.*, **1,124x larger** than the product code itself
- Automatic test generation holds the promise of helping reduce the cost of software testing

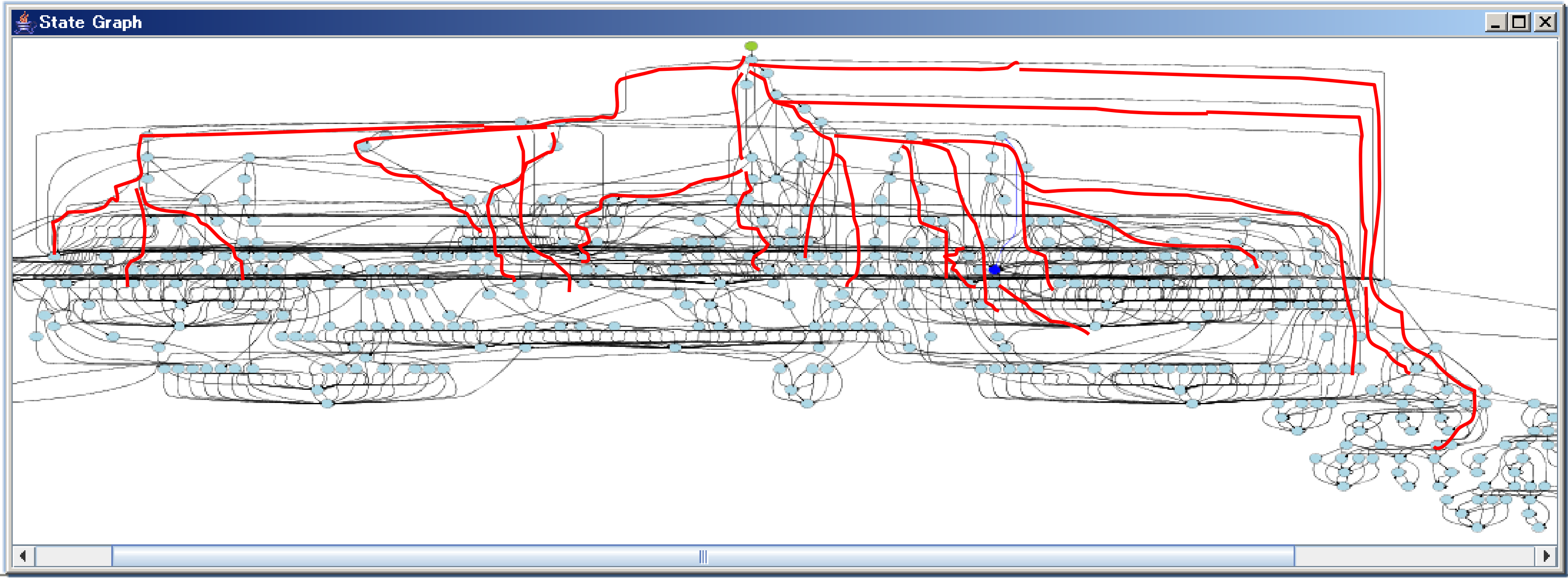
What kind of Software is relevant?

- Did an extensive survey within Fujitsu companies and subsidiaries
  - This type of data is hard to come by not well documented
- Three major categories emerged
  - Enterprise software mainly client-server type
    - Server side code written in Java
    - We have research and a tool on this based on Java Pathfinder
  - Embedded software in networking switches, scanners, vending machines etc.
    - Written in C/C++
    - This will be a target of this talk
  - Client/user side code in an UI/Browser
    - Written in JavaScript, Python, Php etc.
    - We have done some work on this too.

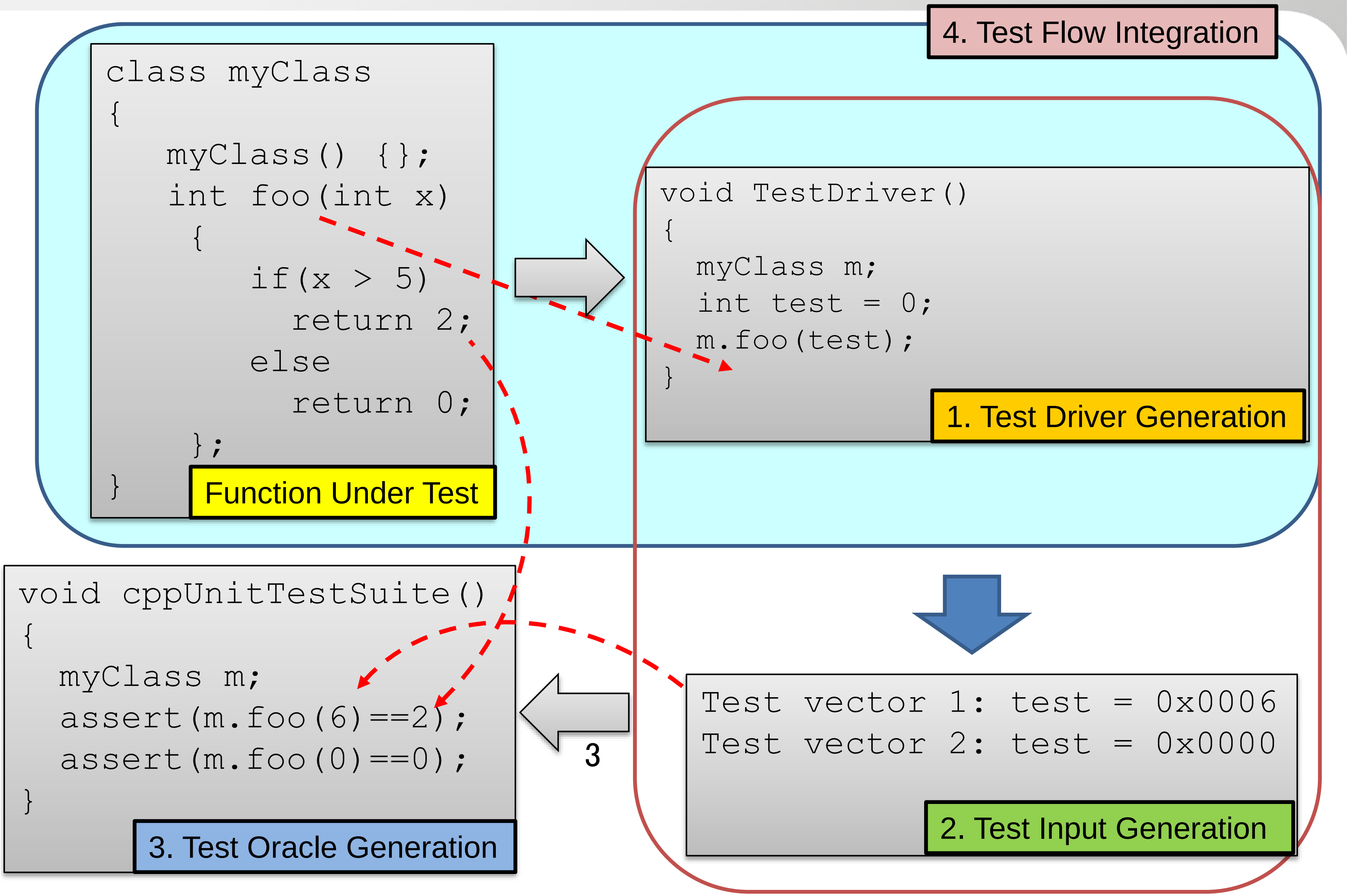
Main problems in the QA process?

FUJITSU

- Testing is time consuming and boring
  - Writing tests was a completely manual process
- Testing was often outsourced to third parties
  - Third parties do not have complete understanding of spec to test
- Test coverage is often poor



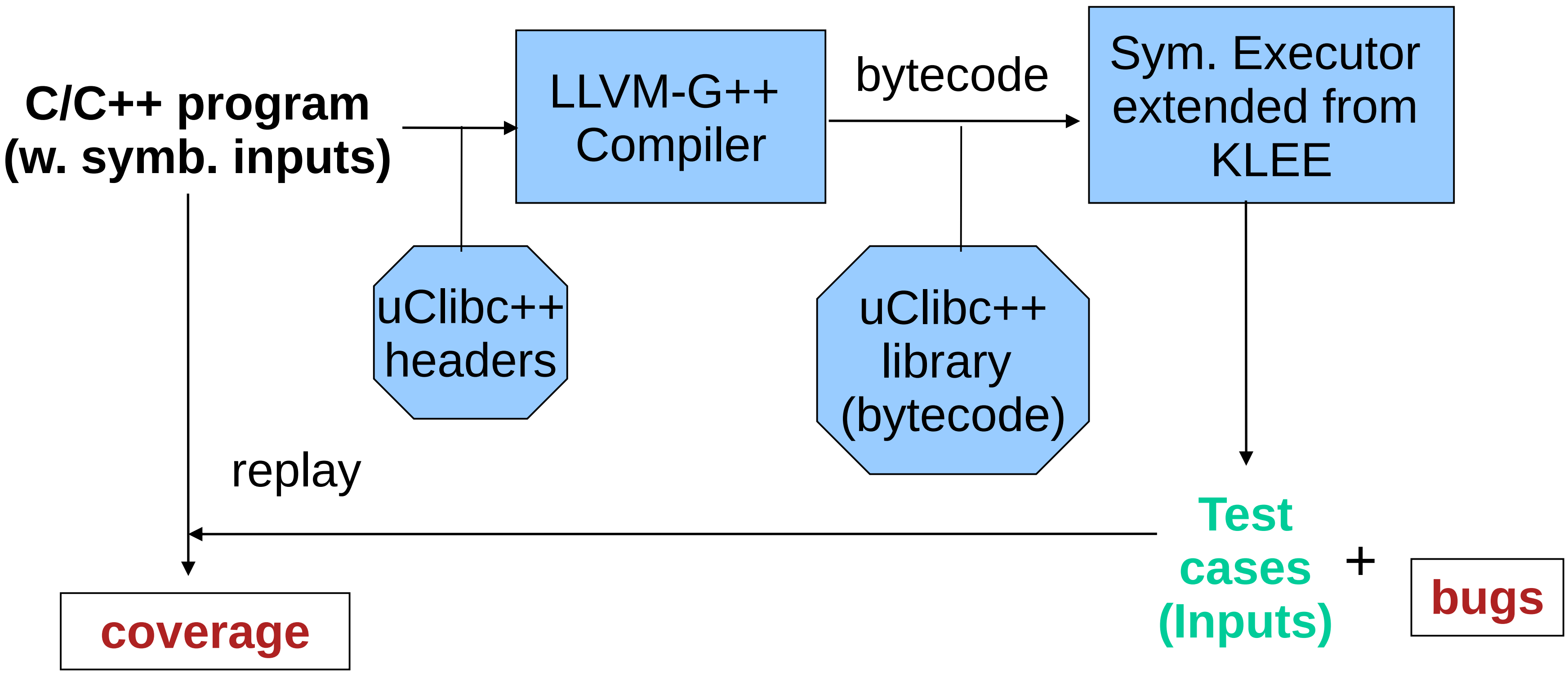
Automated Test Generation



Our Approach

- Input generation
  - Symbolic execution based test generation (KLEE based)
    - Implementing a String based SMT solver
  - Unit level automatic test generation
- Test-driver generation
  - Iterative diagnosis-driven refinement
- Oracle problem
  - User defined, mutation, general errors or golden model
- Test flow integration
  - Incremental test generation from existing test-suites

KLOVER: KLEE for C++ programs



KLOVER Steps

FUJITSU

- Extension of KLEE
- Add support for extra LLVM instructions, mainly intrinsic functions
  - >30 such instructions *e.g. llvm.stacksave, llvm.memcpy*, ...
  - C++ specific instructions, e.g. instructions (llvm.atomic.*) pertaining to the C++ memory model
- Dynamically link the bytecode of a C++ library (implementation) in runtime
  - we extend the publicly available library uClibc++
  - we optimize this library for symbolic execution

KLEE limitations that we faced

FUJITSU

- Sometimes slow, and path blow up
  - It executes the low level bytecode for each API in the library
  - It relies on a constraint solver to calculate symbolic values (computations in AES are too complicated to the solver)
  - The implementation of an API may contain too many branches in the library implementation
    - Take the string library as example. Klee may generate thousands of paths for a simple string program, most of which are useless paths

Our Optimizations

- To speed up KLOVER and scale it to large programs
  - Re-implement some core API in the uClibc++ library
    - Remove the branches within an API implementation so as to avoid useless paths
    - Intercept the calls to some APIs and replace the calls with customized handling
  - Develop our own solvers for constraint solving
    - The PASS string solver in FLA
    - Application-oriented solvers (e.g. for database applications)

Eg: Original Implementation of string.compare (uClibc++)

```
_UCXXEXPORT int compare(const basic_string& str) const{
        ...
        int retval = strcmp(vector<Ch, A>::data, str.vector<Ch, A>::data, len);
        ...
}
```

```
int strncmp(register const char *s1,
                    register const char *s2, size_t n) {
while (n && (*s1 == *s2)) {
            if (!*s1++)
                    return 0;
        ++s2;
        --n;
}

return (n == 0) ? 0 : (*s1 < *s2) ? -1 : 1);
}
```

KLOVER will create > n branches within this API call

Two calls to this API will lead to > n^2 paths

Example: Re-Implementation of string.compare

FUJITSU

```
_UCXXEXPORT int compare(const basic_string& str) const {
…          // some initialization work
int v = 0;       // 1, 0 and -1 stand for gt, eq and lt respectively

for (size_type i = 0; i < rlen; i++) {
            v += (!v) * ((operator[](i) > str[i]) - (operator[](i) < str[i]));
}

v += (!v) * ((vector<Ch, A>::elements > str.elements) -
      (vector<Ch, A>::elements < str.elements));
return v;
}
```

KLOVER now explores only one path within this API call; no useless branches will be created (note that no “if” or “while” is involved).

More Issues

- Executing the API code is slow even using the improved implementation
- It cannot handle symbolic lengths

```
for (size_type i = 0; i < rlen; i++) {
  …
}
```

KLOVER will fail when rlen is symbolic

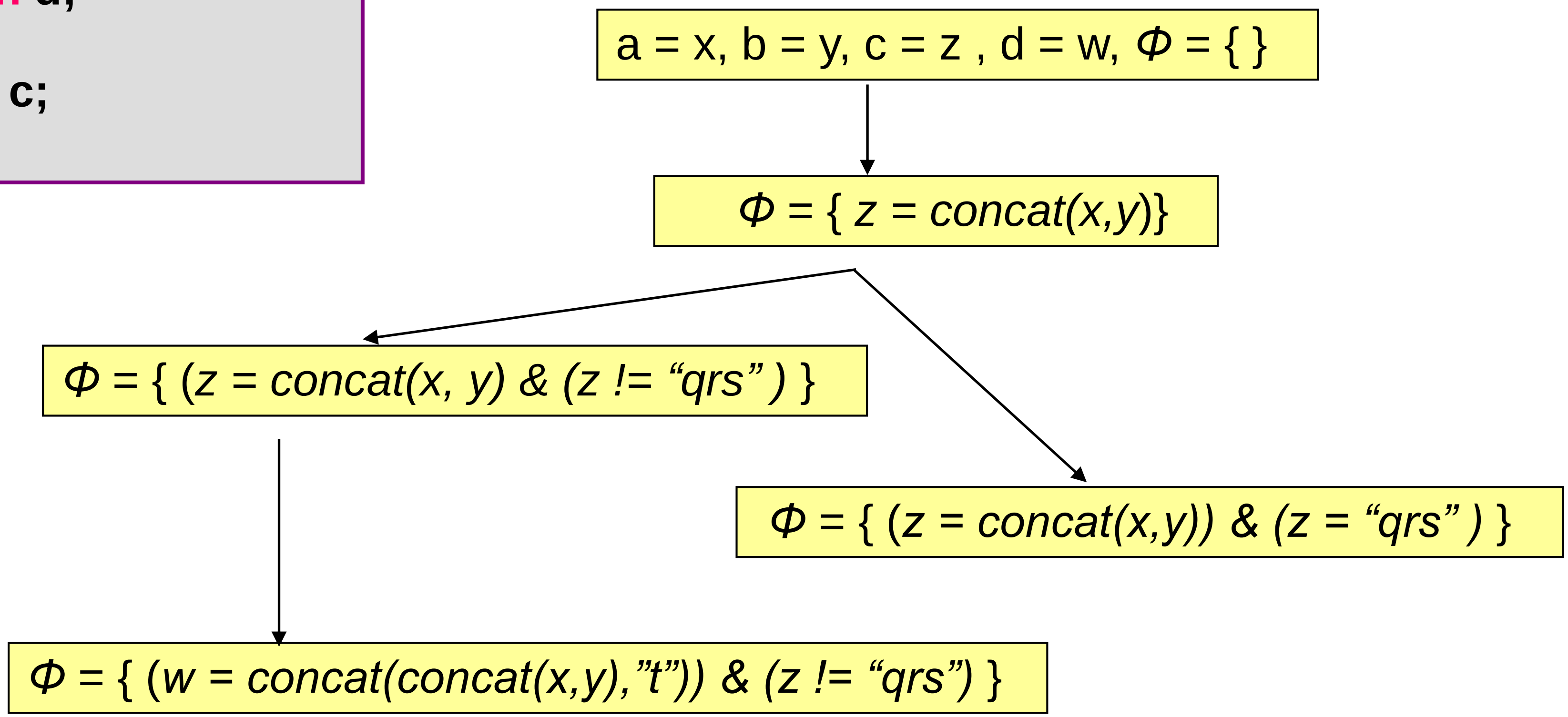
- We prefer a more capable and efficient solver
  – We extend the KLEE IR to support string operations
  – We intercept all string API calls and build the IR
  – We check the satisfiability of a string IR expression with customized solver which support symbolic lengths

Handling Symbolic Stings

```
foo(String a, String b) {
  String c;
  String d;
  c = a.concat(b);
  if !(c.equals("qrs")) {
    d = c.concat("t");
    return d;
  }
  return c;
}
```

We need to create a solver that solves for such string constraints

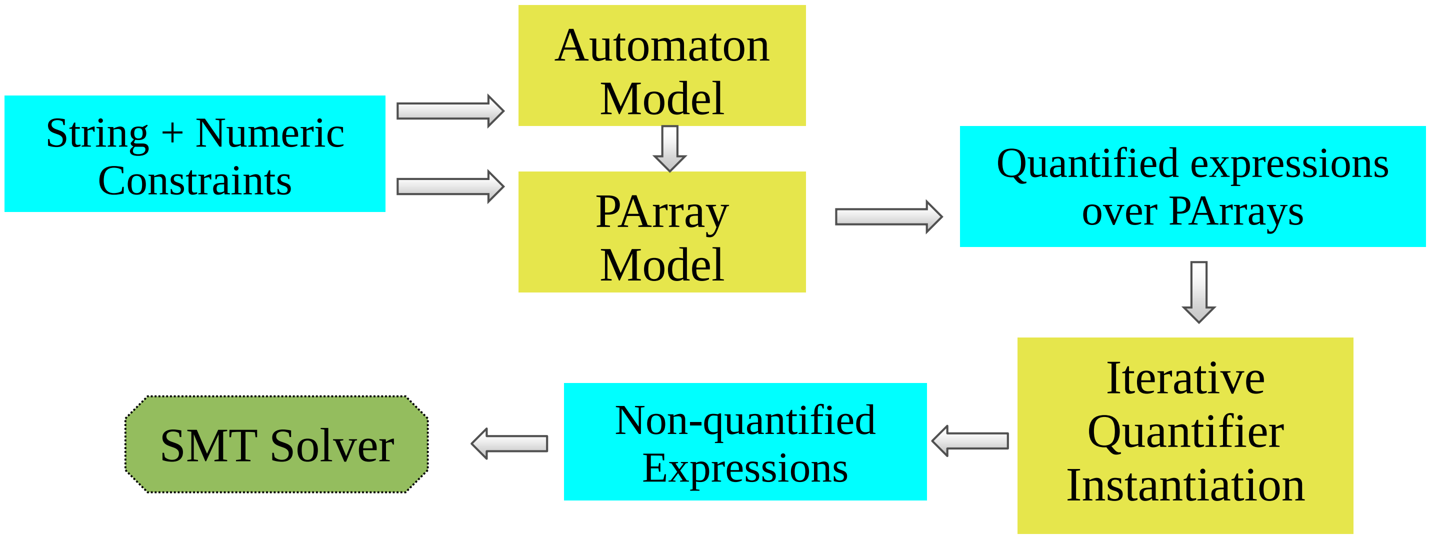
Φ is symbolic expression
x,y,z,w are symbolic strings



PASS (Parameterized Array based String Solver)

FUJITSU

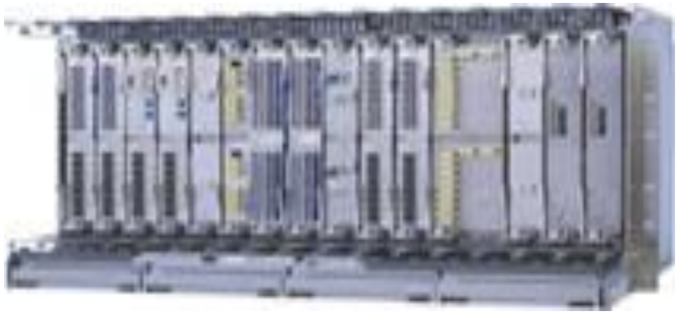
- Model strings as parameterized arrays (PArrays)
  - s1 : [0][1]…[len-1]
  - both the index and the length are parameterized (i.e. symbolic)
  - string constraints are quantified expressions with symbolic indices
    - e.g. $\forall i \in [k, k+10] : s1[i] \neq c$, for symbolic k and c
- Use quantifier elimination to find solutions or prove unsat
  - use symbolic length based heuristics to remove quantifiers
  - always terminate for bounded lengths



PASS: String Solving with Parameterized Array and Interval Automaton
Haifa Verification Conference, 2013

KLOVER Evaluation

FUJITSU



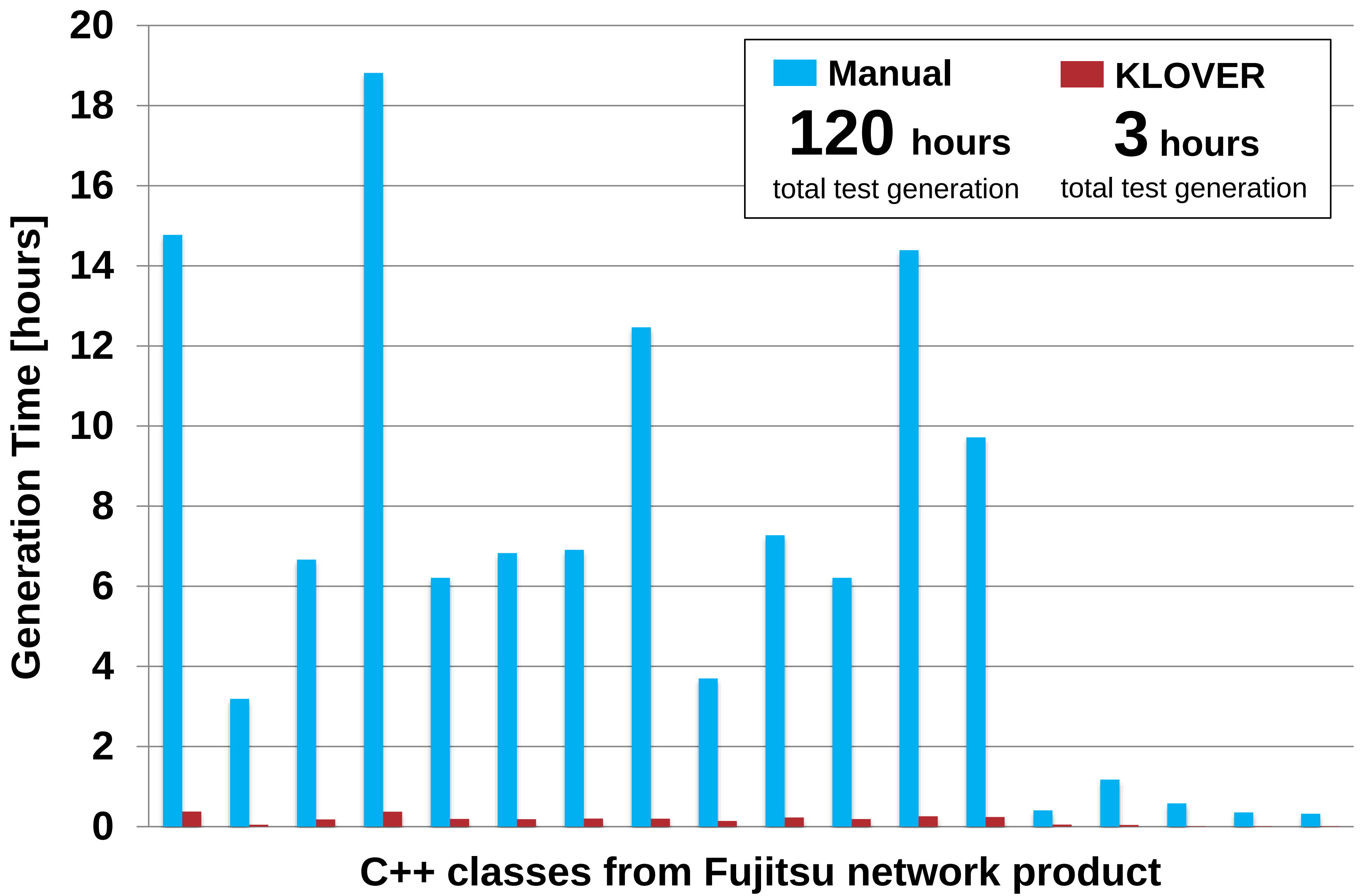
FUJITSU LAYER 2 PHOTONIC SWITCH

4 million lines of C++ code

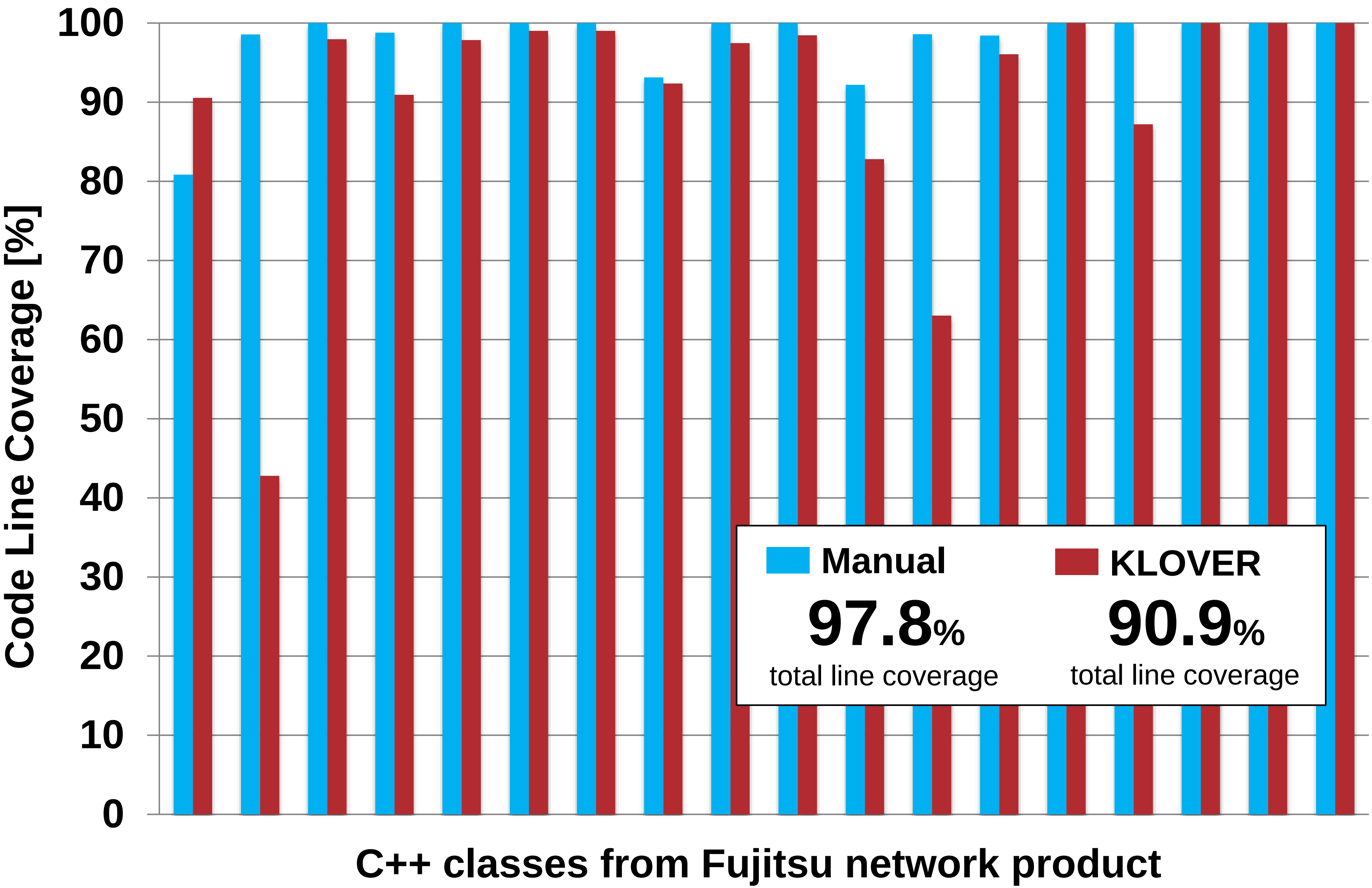
- Manual testing: 4hrs per class
- Unit test quality was poor

- ❑ Applied KLOVER to 5 of 50 modules in switch code
    - ❑ Total about 500K lines of code
- ❑ KLOVER achieved 80% line coverage and found several bugs in code
- ❑ Technology was subsequently transferred to Fujitsu Network Computing, Texas

Benchmark Results: Test Generation Time

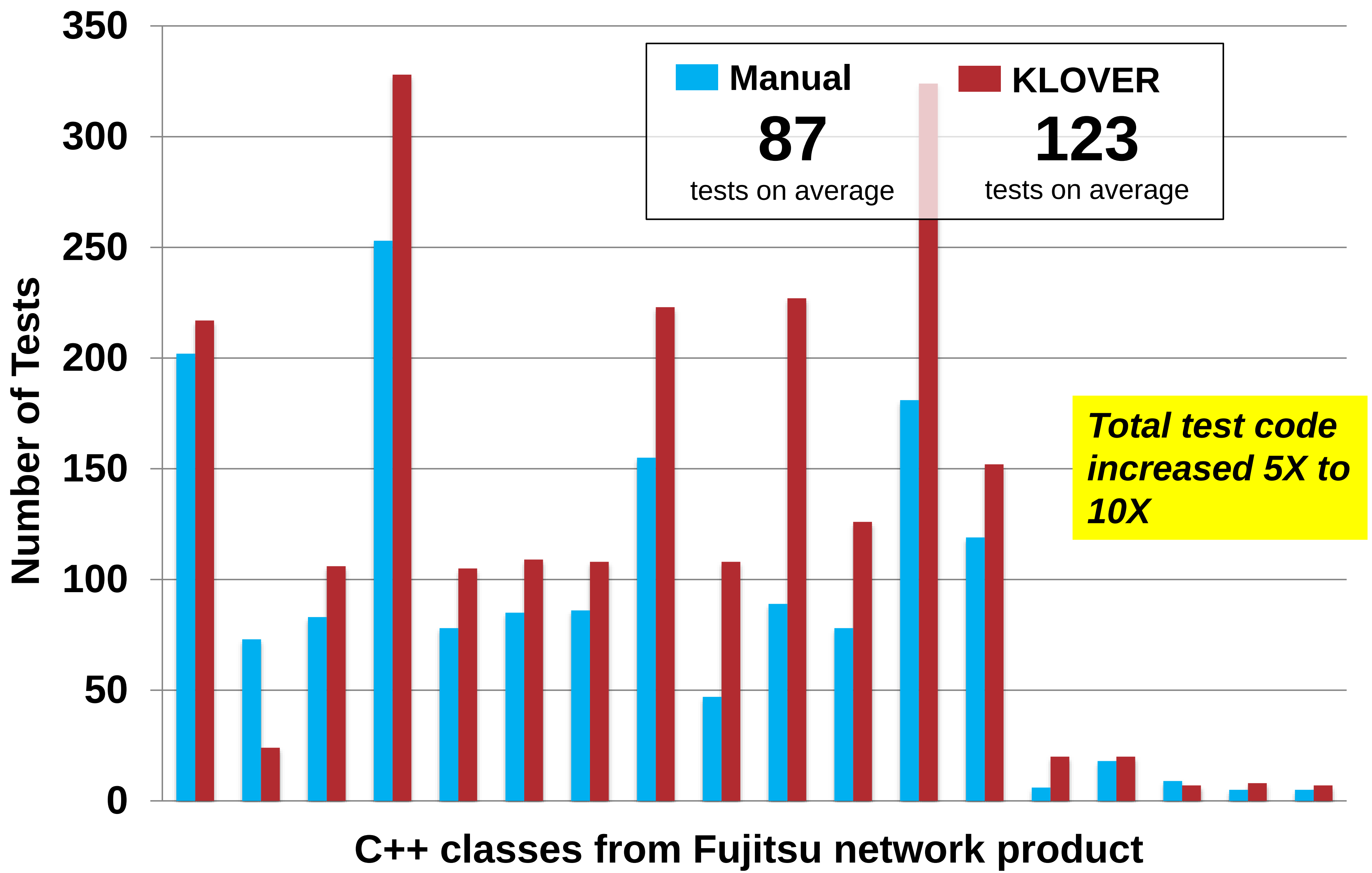


Benchmark Result: Code Coverage



Benchmark Result: Number of Tests

FUJITSU

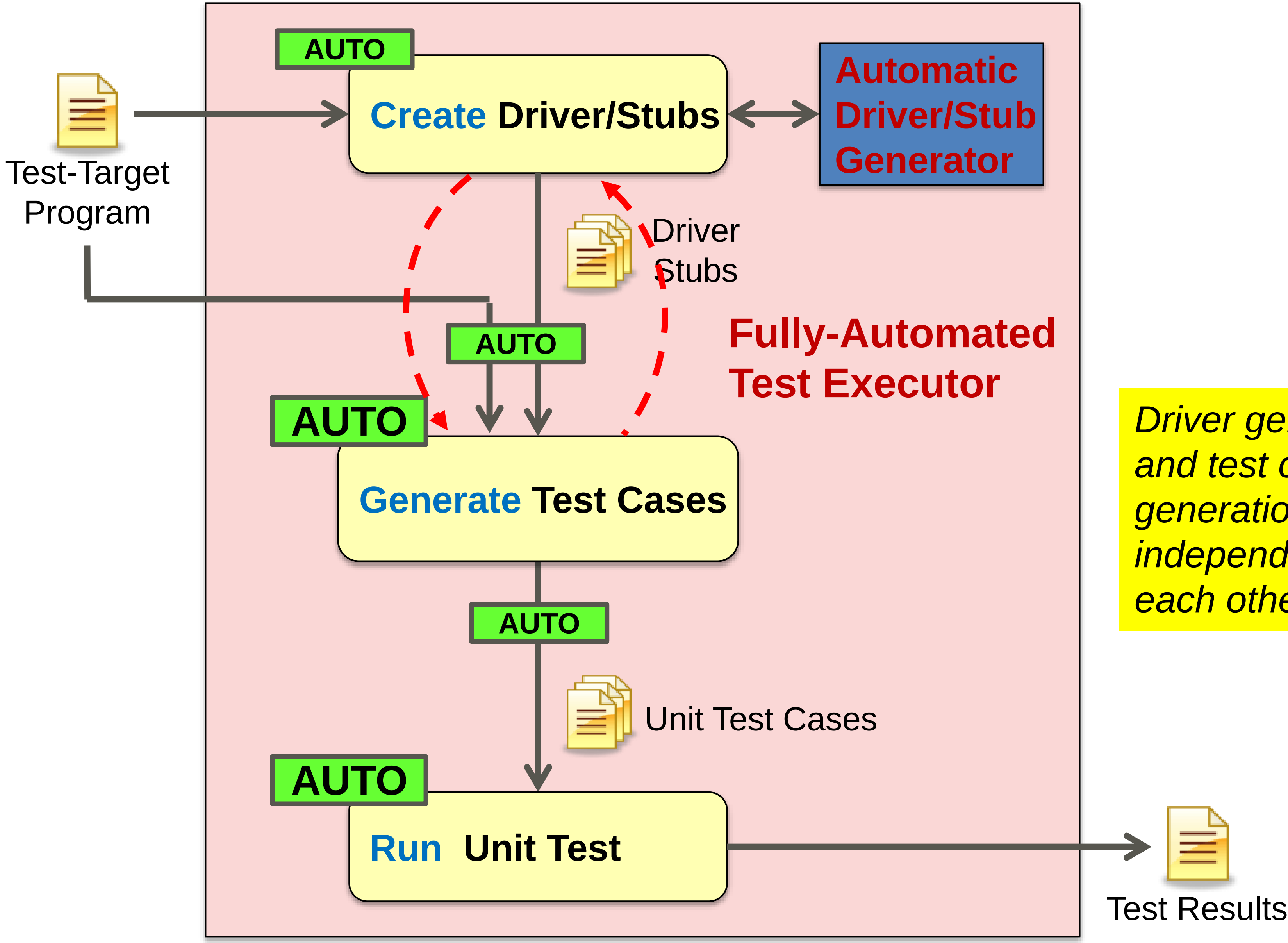


Understanding/Maintaining Auto Generated Test Code

FUJITSU

- Huge number of new test cases generated
- Difficult to understand intent of new tests
- Exceptions caused by unconstrained/invalid inputs
- Difficult to maintain such huge test suite
- The importance of writing compact test cases is well recognized
  - Quotes from the GNU bug reporting instructions (LLVM, Mozilla & Webkit have similar guidelines):
    - *“smaller test cases make debugging easier”*
    - *“GCC developers prefer bug reports with small, portable test cases”*
    - *“minimized test cases can be added to the GCC test suites”*

Test Case quality depends on Test Driver quality



Driver generation and test case generation not independent of each other

Motivational Example: Naïve Approach

FUJITSU

```
void naive_driver() {
  void* x = malloc(1024);
  register_symbolic(x);
  int exp;
  register_symbolic(exp);
  assert(packet_receive(x) == exp);
}
```

Naïve Driver

Not obvious what the function expects. Pass a symbolic memory object of some size

```
int packet_receive(void* x) {
  packet* y = (packet*) x;
  memcpy(y->payload, y->buf, y->size);
  data* z = (data*) y->payload;
  if (z->flag == FLAG_OK) {
    return 0;
  }
  return 1;
}
```

packet_receive() implicitly assumes

- **x** points to **packet** object
- **y->size** is the size of **data** object
- **y->payload** points to **data** object

Symbolic execution would not be able to cover this branch

Intelligent driver generation is the key!

Example: Diagnosis-Driven Driver Refinement

FUJITSU

```
void driver_1() {
  void* x = NULL;
  int exp;
  register_symbolic(exp);
  assert(packet_receive(x) == exp);
}
```

Minimal Driver

Symbolic Execution

```
int packet_receive(void* x) {
  packet* y = (packet*) x;
  memcpy(y->payload, y->buf, y->size);
  data* z = (data*) y->payload;
  if (z->flag == FLAG_OK) {
    return 0;
  }
  return 1;
}
```

Diagnosis Information

(1) **x** is accessed as **packet** type
(2) **sizeof(y->payload)** ≥ **y->size**
(3) **y->payload** is accessed as **data** type
(4) **z->flag** field is used in path condition

Driver Enhancement

```
void driver_2() {
  data* x_buf = new data();
  register_symbolic(x_buf->flag);
  packet* x = new packet();
  x->size = sizeof(data);
  x->buf = x_buf;
  x->payload = new data();
  int exp;
  register_symbolic(exp);
  assert(packet_receive(x) == exp);
}
```

Enhanced Driver

Example: Iterative Test Generation

FUJITSU

- Re-uses existing test-cases from the previous iteration, cloning and modifying them minimally at fine-grained level to create new test-cases

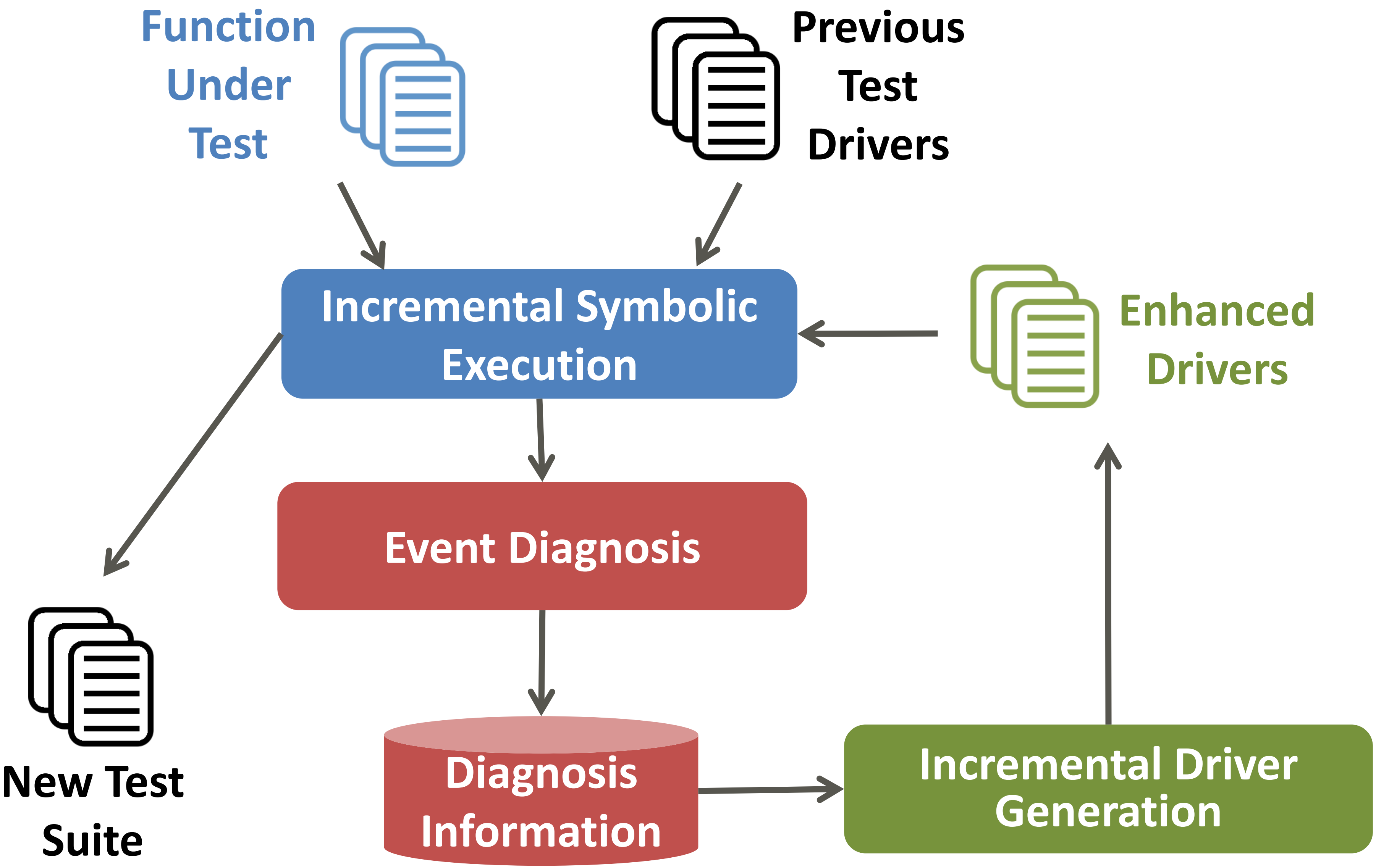
```
void test_1() {
  data* x_buf = new data();
  x_buf->flag = FLAG_OK;
  packet* x = new packet();
  x->size = sizeof(data);
  x->buf = x_buf;
  x->payload = new data();
  assert(packet_receive(x) == 0);
}

void test_2() {
  data* x_buf = new data();
  x_buf->flag = FLAG_INVALID;     // !test_1
  packet* x = new packet();
  x->size = sizeof(data);
  x->buf = x_buf;
  x->payload = new data();
  assert(packet_receive(x) == 1); // !test_1
}
```

1st iteration generates **test_1()**

2nd iteration modifies two lines with an annotation "`// !test_1`".

Overview: Fine-Grained Incremental Generation



Incremental Symbolic Execution

FUJITSU

	Basic Symbolic Execution	C++ Support	Incremental Execution	Event Diagnosis
KLEE	✓			
KLOVER	✓	✓		
FSX	✓	✓	✓ *(Similar to DiSE & Memoise)*	✓

Note: FSX tool is built from scratch.
It does not share code with any of these tools.

Event Diagnosis

Outputs a diagnosis for a given indicative event

```
int packet_receive(void* x) {
  packet* y = (packet*) x;
  memcpy(y->payload, y->buf, y->size);
  data* z = (data*) y->payload;
  if (z->flag == FLAG_OK) {
    return 0;
  }
  return 1;
}
```

Indicative Events

(1) Null pointer access at **y->payload**
(2) Out-of-bound access at **memcpy()**
(3) Branch-not-taken at
if (z->flag == FLAG_OK)

Type Information

y is accessed as **packet** type
z is accessed as **data** type

Range Information

sizeof(y->payload) ≥ y->size

Relevant Input Set

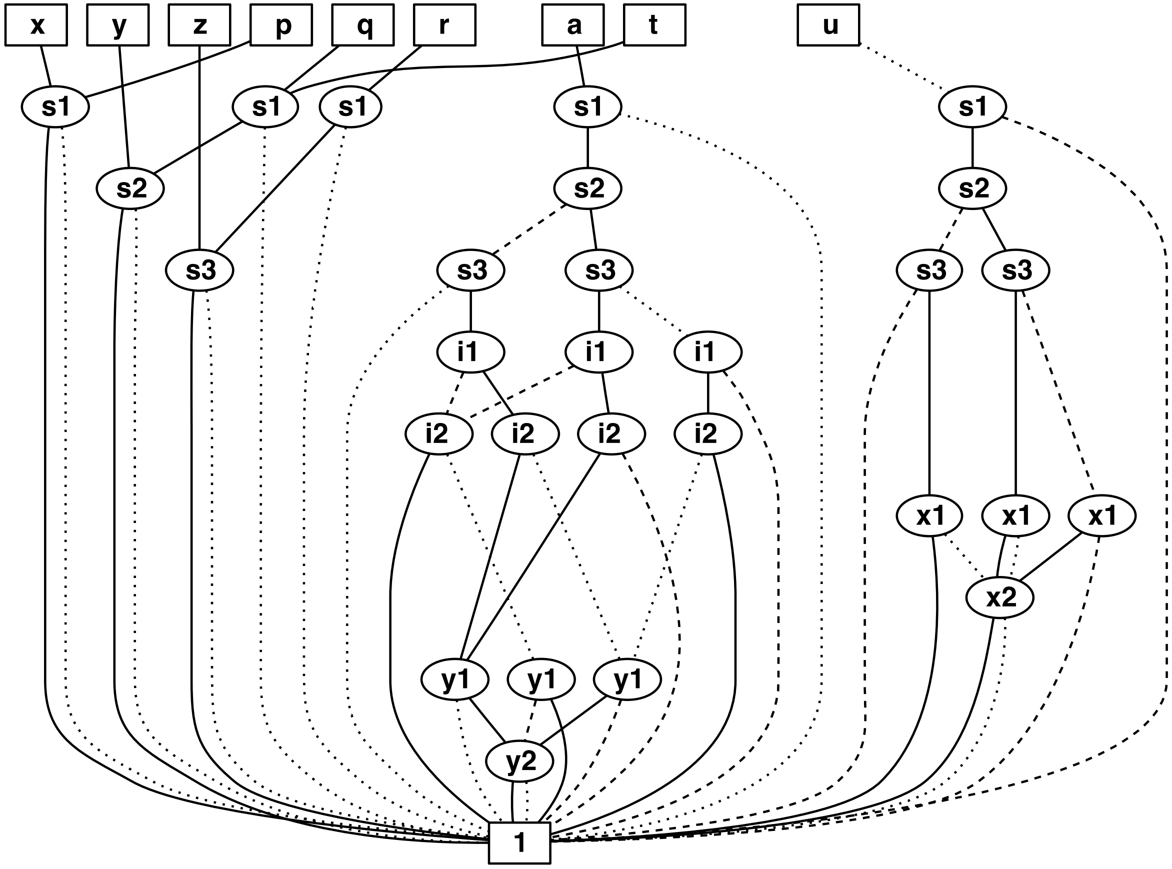
y → x, y->size → x->size
y->buf[] → x->payload[]
z->flag → x->payload->flag

Diagnosis Information

Precise Computation of Relevant Input Sets

FUJITSU

Encoded as Boolean characteristic functions, and compactly represented through Reduced-Ordered Binary Decision Diagrams (ROBDDs)



Incremental Driver Generation

Enhances previous drivers to eliminate events by modifying values of inputs relevant to such events

Null pointer access

```
void* x = NULL;
      . . .
memcpy(y->payload,y->buf,y->size);
```

→

```
void* x = (void*) new packet();
      . . .
memcpy(y->payload,y->buf,y->size);
```

Out-of-bound access

```
x->size = 0;
      . . .
memcpy(y->payload,y->buf,y->size);
```

→

```
x->size = sizeof(data);
      . . .
memcpy(y->payload,y->buf,y->size);
```

Branch-not-taken

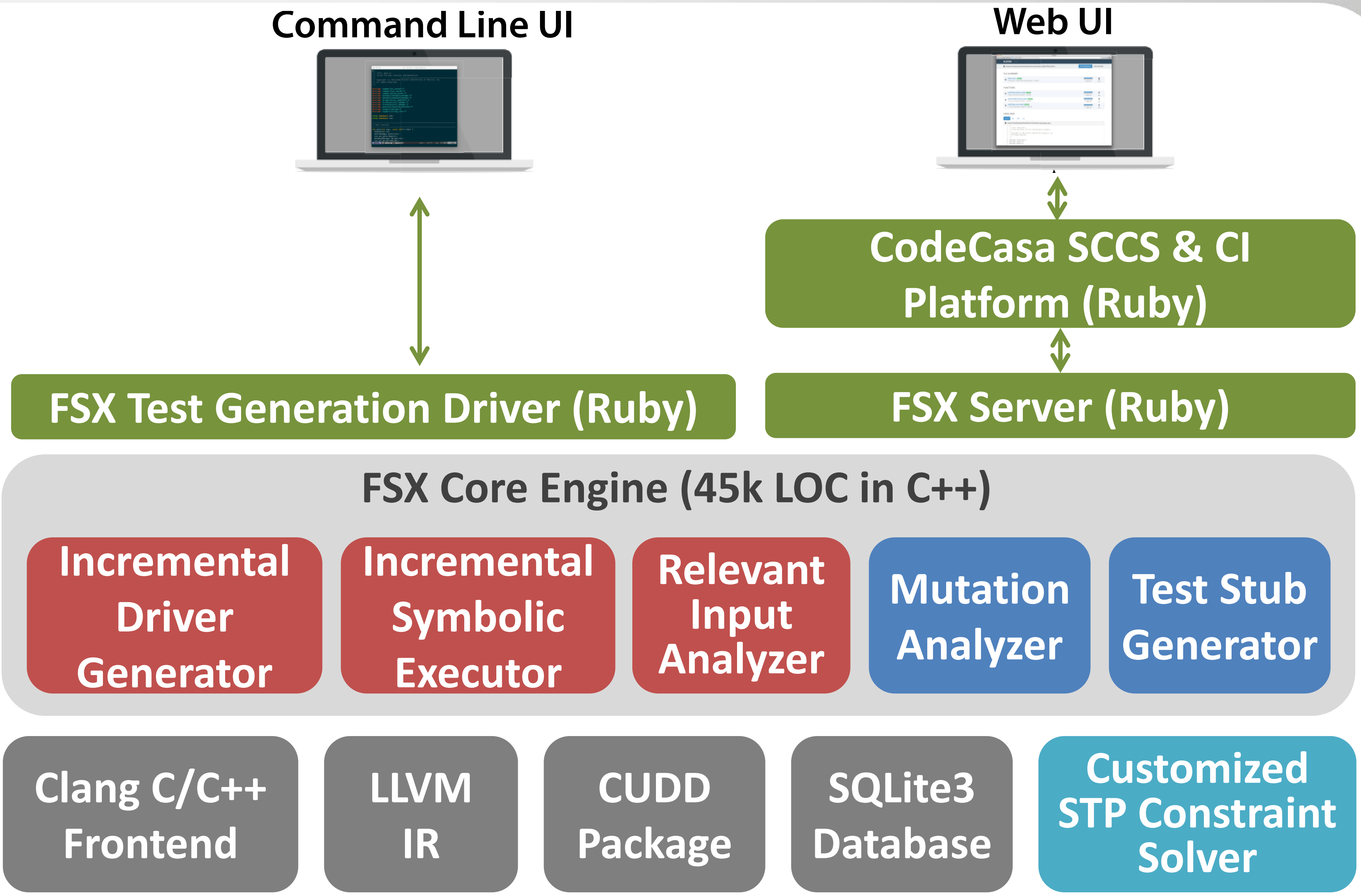
```
x->buf->flag = FLAG_DEFAULT;
      . . .
if (z->flag == FLAG_OK) {
```

→

```
register_symbolic(x->buf->flag);
      . . .
if (z->flag == FLAG_OK) {
```

Tool Implementation

FUJITSU



Experimental Setup

FUJITSU

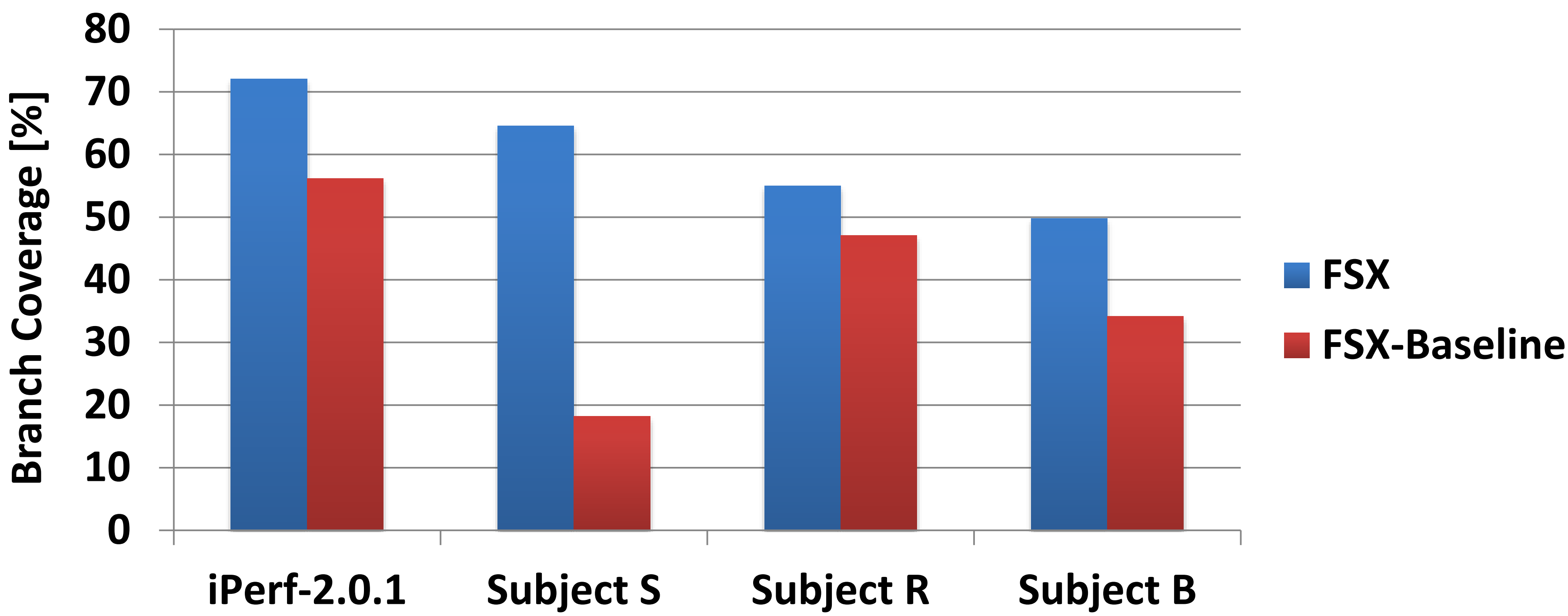
- Baseline tool: FSX-Baseline
  - Generates naïve driver where all assignable variables including function arguments, member variables and global variables are assigned symbolic values and any pointers are set to new objects
  - Symbolic executor implements same search strategy as KLEE
- Benchmark subjects
  - iPerf: network bandwidth measurement tool (5k LOC in C++)
    - Representative set of challenging issues for automatic test generation
    - 5 versions from version 2.0.1 to 2.0.5 used as a software evolution example
  - Three embedded software subjects from Fujitsu commercial network products
    - Subject S (39k LOC in C), Subject R (12k LOC in C), Subject B (15k LOC in C)

RQ1: Full Test Suite Generation

FUJITSU

Can FSX generate high-quality unit tests for large system software?

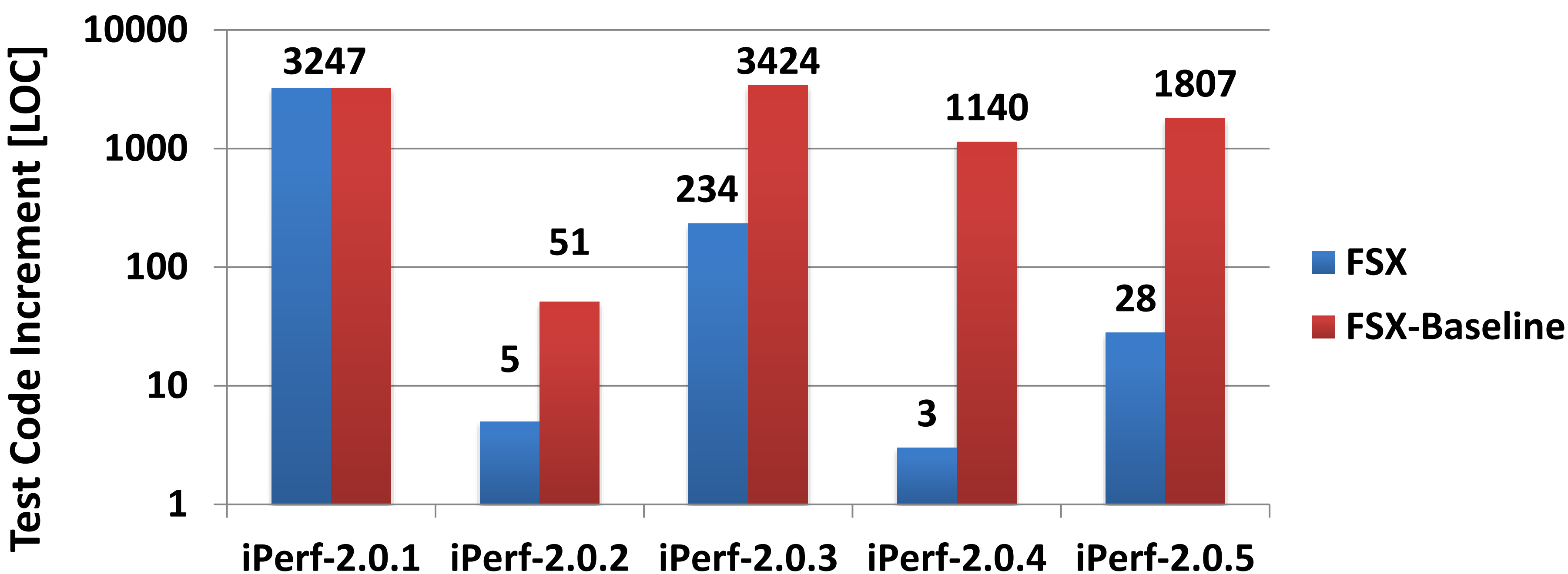
- FSX is able to generate higher quality tests than FSX-Baseline for all benchmarks



RQ2: Test Suite Augmentation

Can FSX perform test-suite augmentation, minimizing maintenance cost of new test code?

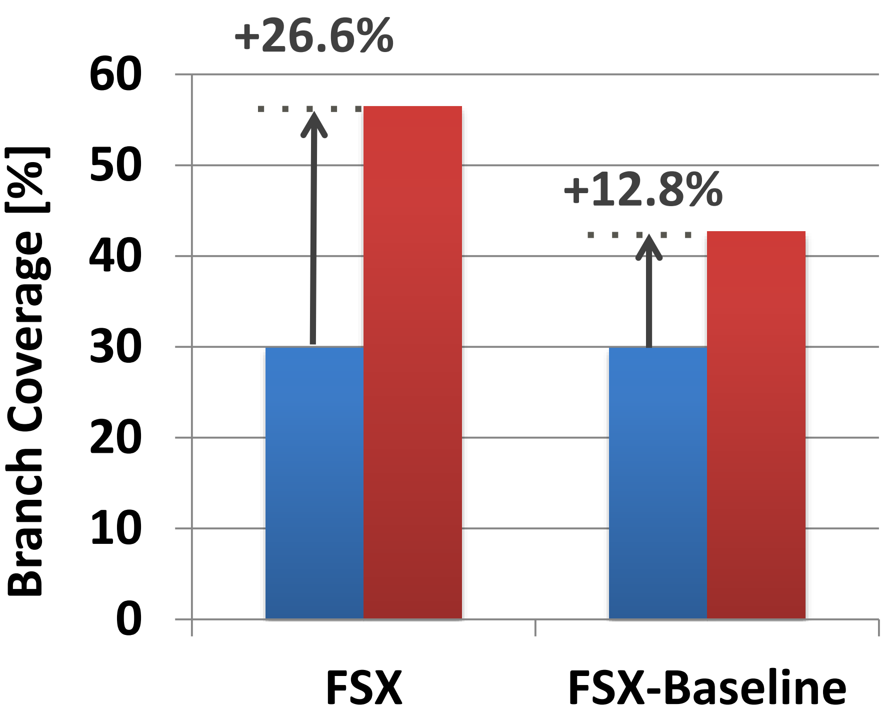
- FSX re-used existing test-cases from the previous version, cloning and modifying them minimally while FSX-Baseline generated new test-cases for uncovered functionality

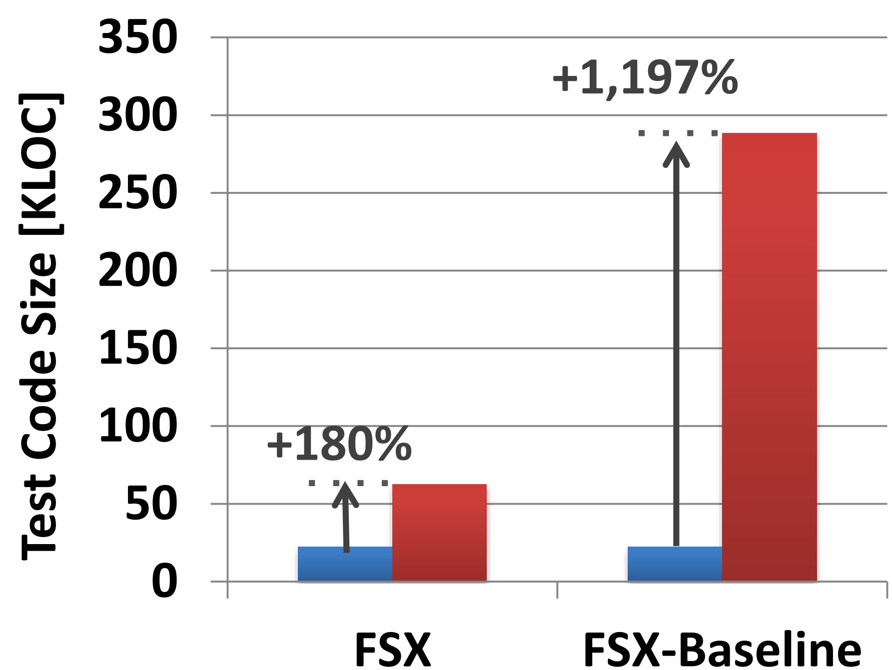


RQ3: Test Suite Enhancement

Can FSX enhance an existing test-suite, minimizing maintenance cost of the new test code?

- FSX's enhancement boosts the coverage much more than FSX-Baseline's while adding much less lines



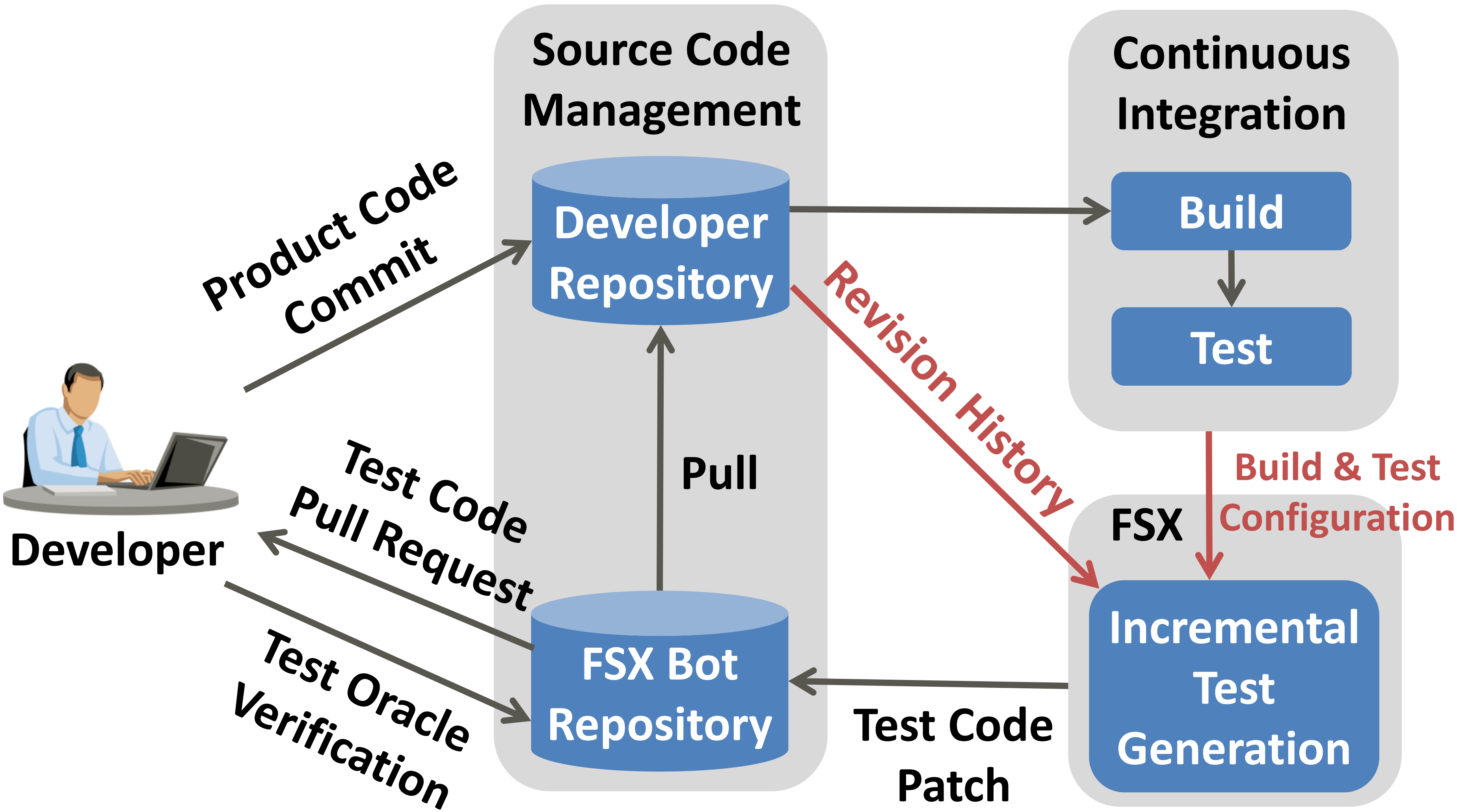


Test Oracle Problem

FUJITSU

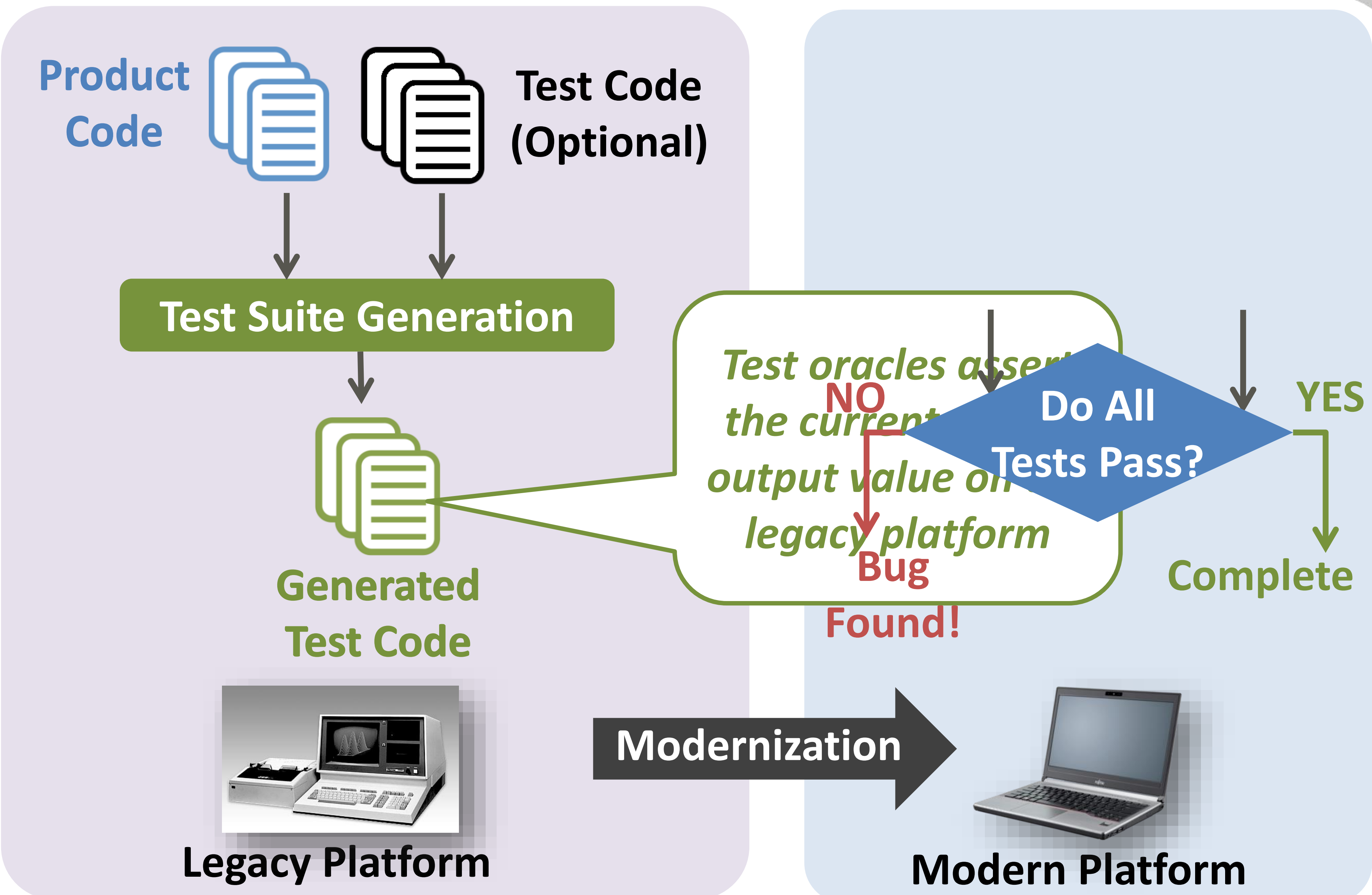
- No easy answer
- Regression test suites from previous model
- User defined assertions in code
- Normal error conditions like exceptions, crashes etc.
- Automatic translation from a specification language like UML
- Using UI feature based invariants

Use-case 1: SCM/CI System Integration



Use-case 2: Application Modernization

FUJITSU



Deployment: Modernizing Vending System

FUJITSU

Product

Buggy Product Code Example

```
        . . .
if (memcmp(x, y, n) == -1) {
        .   .
}
        . . .
```

RHEL 5: memcmp() returns either -1, 0, or 1

RHEL 7: memcmp() returns the value difference between two memory blocks

Legacy Vending System

Modern Vending System

Conclusions

- Fujitsu research concentrated on making automated software testing useable in an industrial development environment
- Created techniques for test driver generation, test input generation, and seamless integration into the test and development cycle at unit testing level
- Tools are being used internally by software teams to enhance test and debug productivity
- Future Challenges:
  - scaling to larger modules for system or integration test
  - better ways of generating test oracles
  - better human – tool coordination
  - some effort in automatic debugging and repair

Acknowledgements

Mukul Prasad

Hiroaki Yoshida

Guodong Li

FUJITSU

shaping tomorrow with you