CSmith++: Adding Core C++ Features to CSmith Fuzzer

Tamir Aviv Ilya Aizin Shahar Ben Hamo
Tel-Aviv University, School of Computer Science
{ iliyaaizin, benhamo1, tamiraviv }@cs.tau.ac.il

Abstract

The following document describes in detail results of an effort to find bugs in C++ compilers as part of "Workshop in automatic program generation for detecting vulnerabilities and errors in compilers and interpreters" lead by Professor Noam Rinetzky.

Csmith[1] is a fuzzer for the C programming language developed by a group of researchers at the University of Utah. In this workshop, we extended its functionality to include many C++ features, such as: classes, multiple inheritance, member and virtual functions and more.

Explanation of compiler choice

We have tested four compilers: G++, CLANG, Visual studio C++, Intel C++.
The first two are open source and free, Visual studio C++ is also free and Intel is fully commercial.
All these compilers promise to support most of C++11 features and C++ standard in general.
All these compilers are used in the industry, been available for years and are actively maintained.

Explanation of OS and architecture choice

We decided to test compilers on MS Windows OS, mostly because it is the most widely used operating system. That being said, our system, both CSmith++ and Validator, is fully portable and can be used on most OS's and architectures. We ran tests on both 32bit and 64bit Windows OS with corresponding compiler versions and target architecture.

Concentration on front end

In the original CSmith, design decision was made to focus on middle end of C compilers. Compiler middle ends perform complex optimizations, if safety checks for these optimizations are not strict enough, the optimization may change the semantics of the program leading to unexpected behavior of the program. For that reason, CSmith does not bother to produce code that will not increase coverage of IR, for example there is only one kind of loops, a for loop, other loop types were not added because they all are translated to the same IR construct. We, however, believe that C++ front ends are worth testing

There are few reasons why we expect to find bugs in the front end

- C++ has many more features than C, many of this features require complex semantic analysis (e.g. traversing multiple inheritance directed acyclic graph)
- There was not a lot of effort put into finding bugs in C++ compilers [2].
- CLANG and g++ are by far the buggiest components of LLVM and GCC frameworks respectively [2].
- Visual studio C++ has many currently open bugs in its front end, many of these bugs consider old constructs of C++ language that should have been implemented correctly by now.

Still avoiding undefined behavior

Not unlike CSmith the code that CSmith++ produces is syntactically correct, semantically sound and have one well defined meaning according to C++ standard (program's meaning is the collection of side effects it has on its global variables). Corollary, we don't produce programs with undefined behavior. If the program misbehaves at run time or doesn't compile – compiler is to blame.

Test case reduction techniques

Every program that CSmith generates prints out a check sum of global variables. When this checksum differs between executables compiled with different compilers or flags a bug is found in one of the compilers. CSmith produces quite large files, finding the exact part of the code that miscompiled and lead to difference in checksum is a real issue. Test case reduction techniques are needed. Bug report submitted to compiler developer will not be taken seriously unless they include a short and understandable peace of code. CSmith team developed a dedicated tool that automatically reduces test cases while not introducing undefined behavior to the code, but unfortunately it does not fully support C++ [4].

This was less of an issue for as. Since we decided to concentrate on front end, and on code that should compile, but doesn't, we can track the problematic peace of code just by looking on the compiler error message. These messages are usually quite informative and allow as to reduce test cases to small size in a matter of minutes. Bugs considering differences in runtime behavior (checksum) we reduced by hand, a much more time consuming task.

Bug types:

There are a few types of bugs in compilers that our system can detect:

- A perfectly legal C++ code that should compile, but doesn't
- Incorrect run time behavior of the compiled program
- A code that causes a compiler to crush or to hang indefinitely

  We do not try to find bugs considering code that should not compile but does.

Features added:

Classes:

We added a full support of classes, member fields, member functions and all types of inheritance.

C++ allows multiple inheritance and multiple virtual inheritance. Multiple virtual inheritance is type of inheritance where an object may have only a single version of base class sub object of a certain type, even though it indirectly inherits from this type several times. This introduces a whole bunch of complication and potential problems. As we discovered, **none** of the compilers we have tested implemented this logic perfectly.

Member lookup:

With multiple inheritance, a single class may inherit a member with the same name from several other classes. Sometimes these members become ambiguous, and cannot be accessed from the class's scope. The C++ standard has a detailed explanation of the complex logic that determines which members can be accessed in section 10.2 [class.member.lookup][3]. Since we decided to only produce correct C++ code, we had to implement this logic ourselves.

Access control:

Like most languages, C++ has three types of access control: public, protected and private. In addition, C++ has public, protected and private inheritance, meaning that the access of the members received from base classes is changed according to inheritance access. This allows a class to inherit, while making the base class and its members inaccessible from outside of the class. Protected and private inheritance do not maintain an is-a relationship between classes and do not allow polymorphism. C++ also allows a class to declare another class as 'friend' and thus granting it full access to its private members. These features are defined in section 11 [class.access][2]. We implemented all these features in CSmith++.

Virtual functions and polymorphism:

Like most OOP languages C++ allows to declare a member function as virtual. This means that the actual binding between the calling object and a member function will happen in run time based on the dynamic type of the object. Here also, multiple inheritance introduces several complications. According to C++ standard, section 10.3 [class.virtual][3]: a virtual function inherited via two different paths in the inheritance graph cannot be overwritten twice or more, by different function on these paths, unless these functions themselves are then being overwritten by a single function. We had to implement the logic required for CSmith++ to produce code that follows this convention, while using as much virtual functions as possible.

Compiler Tester (The Validator)

It operates in the following way:

An HTML server is established on the working machine and the user can access the validator using a browser of her choice.

From a web page a user can: initiate automatic continuous testing, run a single test and view bugs found (including the test code, statistics about compilation and execution and compiler error messages). All the information about the testing is stored in SQL DB (using SQLITE Python module) and is accessible at any time. A user can query the DB at any time to get information the GUI does not present.

The validator is build according to following design principles:

- Portability: The validator itself is written entirely in Python (HTML+CSS for GUI, SQL for storing results) and is fully portable.
- Extensibility: The use of SQL allows the Compiler Tester to run for a long period of time while preserving and efficiently accessing tests data.
- Concurrency: The Compiler Tester can spawn as many threads as user sees fit. Each such thread will continuously run end to end test case: generation, compilation and result comparison.
- Generality: The Compiler Tester can be used to test other languages, fuzzers and compilers by changing its configuration.

Perhaps one of the most important aspects of Compiler Tester it is its ability to provide a kind of “self-fuzzing” services for CSmith++.

If all compilers succeed in compiling the test case file and if all the executables they’ve produced output the same checksum – we consider this whole test as a success. If one of the compilers has a result which conflicts with others, either in compilation success or runtime behavior of the executable, we found a bug in the compiler. But what happens if all compilers fail to compile the test case file, or all the executables crash in runtime? CSmith++ should only produce code that, according to C++ standard, compiles, and it should not produce code that crashes in runtime, because this implies undefined behavior in the code. But C++ is complex, and if professional compiler developers made mistakes how can we guarantee that our logic is correct? When compilers unanimously think that our code is incorrect for the same reason, we can safely assume that there is in fact a bug in CSmith++. In such a way, we can recognize bugs in our, CSmith++, code and fix them. We call these kind of bugs – generation bugs, and the whole approach of letting compiler “voting” participate in testing of our code we call self-fuzzing.

We can model this approach as a voting system in which every compiler and CSmith++ have a vote (CSmith++ always votes that the code is correct), votes are cast by running the test, whoever is in the minority must be wrong.

About 10% percent of programs that CSmith++ generates never terminate. When running executables, the Validator has no way of knowing whether a program will terminate or not. This means that the Validator needs to set a timeout for execution of the programs. When executing a single program that was compiled by different compilers it may occur that some of the executables will terminate before the timeout, while others not. This may suggest difference in run time behavior between compilers, which is a bug, or simply a difference in produced code efficiency, which is not a bug. We treat such test cases as “Warnings”, the Validator marks and stores them for further investigation by a human.

Bugs

We found bugs in all compilers we have tested!

All the bugs presented are minimized from the original CSmith++ output and trying to simplify the code further will not reproduce the issue.

MS Visual C++ (2015 update 3)

This is by far the most common bug we encountered:

```
struct S0 {};
struct S1 : S0 {};
struct S2 : virtual S0, S1 {};  //doesn't work
struct S3 : S1, virtual S0 { }; // works

int main() {}
//Error C2584   'S2': direct base 'S0' is
inaccessible; already a base of 'S1'
```

In C++ a class can indirectly inherit from other class several times. Furthermore, the order of inheritance should not matter here according to C++ standard, section 10.1.2 [class.mi][3].

Because of the error message - we suspect that a conscious decision was made by the development team not to follow the standard here.

Table 1: Incidence of bug

Number of test cases	Bug occurrence in test cases	Commonness
38423	14935	38.87%

```
1. struct S0 {};
2. struct S1 : S0 {};
3. struct S2 : virtual S0 {};
4. struct S3 : S2, S1 {};

5. int main()
6. {
7.      S3 a, b;
8.      b = a;          // works
9.      S3 c = a;       // doesnt work
10.}
//Error C2594 'argument': ambiguous conversions
from 'const S3' to 'const S0 &'
```

In this code struct S3 inherits from S0 twice. Like in previous example this is perfectly legal, and copy constructor in line 9 should work. In any case, the line 8 allows the same functionality and should therefore behave the same.

Table 2: Incidence of bug

Number of test cases	Bug occurrence in test cases	Commonness
38423	1560	4.06%

```
struct S0 {
        int  a = 1;
};
struct S1 : virtual S0 {};
struct S2 : S1, protected S0 {};
struct S3 : virtual S2, virtual S0 {};

int main(){
        S3 a, b;
        b = a;
}
```

```
//Error C2594   'static_cast': ambiguous
conversions from 'const S3' to 'const S0 &'
```

The assign operator should work here, even though S3 does inherit twice from S0, there is no ambiguity here.

Table 3: Incidence of bug

Number of test cases	Bug occurrence in test cases	Commonness
38423	432	1.12%

```
struct S0 {};
struct S1 : private virtual S0 {};
struct S2 :  S1 {};

int main(){
        S2 a;
}
//Error C2280   'S2::S2(void)':  attempting  to
reference a deleted function
```

The constructor of S2 is deleted for some reason.

Table 4: Incidence of bug

Number of test cases	Bug occurrence in test cases	Commonness
38423	9737	25.34%

First code segment:

```
int f1() {
        return 1;
}
void f2(unsigned int ui1) {}

int main() {
        short a = 0;
        f2(a & (f1() | -1));
}
```

Second code segment:

```
void f(unsigned long b){}

int main()
{
        short av;
        short* a = &av;
        f(av | ((*a) = 0));
}
```

Third code segment:

```
void func(unsigned int n1)
{

}

int main(int argc, char* argv[])
{
        short m;
        int arr[1];
        func(m & (arr[0] = -1L));
}

// fatal error C1001: An internal error has
occurred in the compiler.
```

These three code segments cause the compiler to crash unexplainably. It's a legal C++ code, but even if it wasn't the compiler should not crash in any case, furthermore, these segments compile perfectly with visual studio C compiler.

Table 5: Incidence of bug

Number of test cases	Bug occurrence in test cases	Commonness
38423	12	0.03%

Clang (3.8.1)

```
struct S0{
   int n = 1;
};
struct S1: virtual S0 {};
struct S2: virtual S1 {};
struct S3: protected S2, virtual S1 {};

int main ()
{
        S3 a;
        int num = a.n;
}
//error : cannot cast 'S3' to its protected base
class 'S0'
//error : 'n' is a protected member of 'S0'
```

In this code struct S3 inherits member 'n' from S0 via two different paths one with protected access and the other with public access. According to C++ standard, section 11.6 [class.paths][3]: the access is that of the path that gives most access.

Table 6: Incidence of bug

Number of test cases	Bug occurrence in test cases	Commonness
27507	155	0.56%

```
int main()
{
        short  sh = 9L;
        unsigned int ui1 = (sh <=
0xD48FBCB67A80957DLL);
        printf("%X\n", ui1);
}
```

This program prints 0, even though it should print 1. The problem here is that Clang tries to store this big constant in a "long long" (signed 64bit) variable where it cannot fit, but it should store it in an "unsigned long long" (unsigned 64bit), where it can fit. The standard clearly states that that is the correct behavior at section 2.14.2 [lex.icon].

G++ (5.3.0)

```
struct S0 { int n; };
struct S1 { int n; };
struct S2 : S0, S1 { };
struct S3 : virtual S2 { int n; };
struct S4 : virtual S2 {};
struct S5 : S4, S3 { };
int main() {
        S5 s;
        int num = s.n;
}
//error: request for member 'n' is ambiguous
```

In this code struct S5 inherits member 'n' via two different paths: from S3 that defined 'n' himself, and from S4 that inherits 'n' from S2.

In S2, member 'n' is indeed ambiguous, but because S5 inherits from S2 only once, and only in one of the paths from S2 to S5 ,member 'n' is hidden, S5 should inherit only the member 'n' of the struct that hides the member.

Changing the order of inheritance of S5 bases (inherit from S3 first) should not change the semantics of member lookup, but here it makes the code compile, section 10.2 [class.member.lookup][3], in C++ standard, implies that the order is not important.

Table 7: Incidence of bug

Number of test cases	Bug occurrence in test cases	Commonness
38423	853	2.22%

Intel C++ (16.0.3.207)

```
struct S0 { int n; };
struct S1 : S0 {};
struct S2 : virtual S0, virtual S1 {};
struct S3 : S1, S2 { int n; };
struct S4 : S3, S2, virtual S0 {};

int main (){
        S4 s;
        s.n;
}
//error: "S4::n" is ambiguous
```

According to algorithm of member lookup in section 10.2 [class.member.lookup][3], access from S4 to 'n' is unambiguous because member 'n' in S3 hides all other members named 'n'.

Table 8: Incidence of bug

Number of test cases	Bug occurrence in test cases	Commonness
32953	42	0.1275%

```
struct S0 {
};

struct S2 : virtual S0 {
};

S2 g_1[3];

int main(int argc, char* argv[])
{

}
```

This code, compiled with optimizations disabled and targeting 32bit architecture causes the resulting executable to crash at run time. Interesting observation: increasing the size of an array, even by one, will not reproduce the crash.

Table 9: Incidence of bug

Number of test cases	Bug occurrence in test cases	Commonness
32953	47	0.1426%

```
#include <stdio.h>

struct S0 {
        int t = 11;
};

struct S2 : virtual S0 {
};

S2 g_1[1];

int main(int argc, char* argv[])
{
        printf("%d", g_1[0].t);
}
```

This program prints 0, when it, clearly, should print 11.

There are a few additional bugs we have found that are not reducible to small representative code segments.

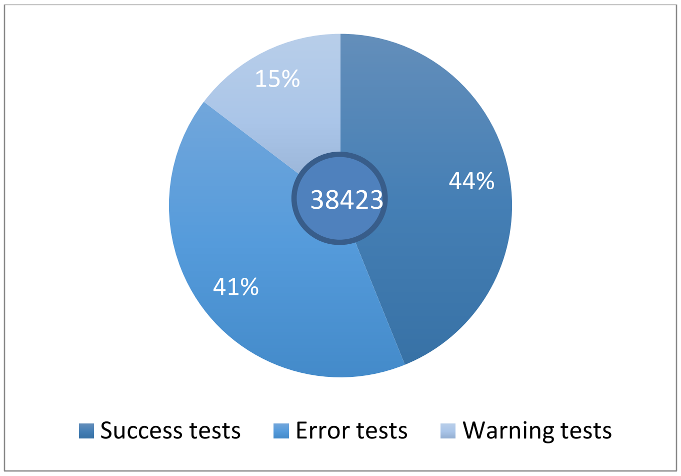
- The Clang compiler crashes when compiling code with very large and complicated multiple inheritance DAG. The smallest example we found contained more than 20 classes, many of which inherited from up to 6 other classes
- Code compiled with Intel compiler sometimes crashes unexpectedly at runtime. Debugging the code raises an exception that states that there is a problem with storing one of the call stack pointers before a function call.

Statistics

We ran the compiler tester for the total CPU time of: **41.18 days** (14% test case generation, 30% compilation, 57% execution)

The number of test cases produced is: **38423**

Classification of tests results:



This is the classification of all the test cases.

- Success tests: tests cases that did not reveal any bugs.
- Error tests: test cases that found bugs (not unique)
- Warning tests: test cases that produced a warning (some executables terminated before the timeout and some did not)

Interesting observation: More than half of the errors are Visual studio bugs (not unique).

Classification	Number of test cases
Success	16857
Error	15929
Warning	5637

Number of bugs (not unique) per compiler:

	Number of bugs
gcc	853
visual	26676
clang	155
intel	89

Number of bugs: amount of test cases that revealed bugs (not unique)

Compilation and execution time:

	Compilation time	Execution time
gcc	2.242834254	0.054309319
visual	1.360926594	0.195643505
clang	1.866462022	0.293105933
intel	25.36779129	0.251681917

- Compilation time: the average time it took each compiler to compile test cases that did not result in compilation error.
- Execution time: the average time it took executables produced by each compiler to terminate (only executables that terminated before the timeout are included).

We can see that Intel's much slower compilation did not result in faster execution of the kind of code CSmith++ produces.

Timeout:

	Compilation timeout
gcc	1
visual	0
clang	0
intel	1045

Compilation timeout: number of test cases for each compiler in which the compilation was stopped by the compiler tested due to timeout.

In fact, compilation timeout is the main reason why Intel's compilation times in the previous table are not even worse.

Checksum:

Error test cases	Checksum error	%
15929	276	1.73%

Error test cases: the number of test cases that found bugs (not unique)

Checksum error: number of tests cases where bugs were recognized by comparing checksum of global variables.

References

[1] Xuejun Yang, Yang Chen, Eric Eide and John Regehr. Finding and Understanding Bugs in C Compilers. 2011

[2] Chengnian Sun, Vu Le, Qirun Zhang and Zhendong Su. Toward Understanding Compiler Bugs in GCC and LLVM. ISSTA '16, July 18-22, 2016, SaarbrÃijcken, Germany

[3] Working Draft, Standard for Programming Language C++, 2012-01-16

[4] John Regehr. Slides from talk at PLDI 2012