1. Introduction

This literature review is produced for the 'Debugging 42' project for the ENGR 489 course. Compiler testing can a difficult and time consuming process and Leroy [1] states that compiler-introduced bugs are widely considered to be extremely difficult to find and track down.

This literature review will explore the current approaches to testing and debugging of compilers and will then explore how ideas and tools in the current literature can be applied to the 'Debugging 42' project.

2. Verified compiler

Harrison [2] states that testing by itself is usually inadequate to guarantee correctness due to the large number of possible inputs and program states. This is especially true in case of a compiler which has a practically infinite number of possible inputs and states. Therefore we can conclude that compilers are highly likely to contain bugs as most large scale programs will contain bugs [2].

A logical idea then could be to use formal specifications and verification to produce our goal of a bug-free compiler. Written code itself could be bug free but bugs in the compiler could change the way the code works after being compiled and invalidate all testing and verification on the source code itself. Formal verification has a history of use in many areas including CPU design [2] [3], smart contracts [4] (INSERT MORE HERE) but verification of a compiler is less explored than some of the previously mentioned areas. Although this does not mean that research has not been published in this area. The first proof was published in 1967 [5]

2.1. Leroy's semantic preservation

Leroy [1] presents a compiler called CompCert for a large subset [6] of the C language.

This paper [1] presents a formally verified compiler that provides some semantic preservation. This compiler produces a compiled program C from source code S. Then introduced is B, an observable behaviour. Leroy then introduces some notions of semantic preservation using these definitions. Firstly,

$$\forall B,\ S \Downarrow B \iff C \Downarrow B \tag{1}$$

This notion states that the observable behaviours of the source program and the compiled program are exactly the same. Leroy states that is the strongest notion of semantic preservation but explains that it is too strong to be a usable notion, as if the source language is not deterministic compilers can select differing behaviours for the source program. An example of evaluation of order of expressions in C compilers is provided for the notion being too strong. Leroy then introduces some weakened or relaxed definitions of (1) that are more useful. Leroy states two of these as the most useful, (2) and (3).

$$S \textbf{ safe} \implies (\forall B,\ C \Downarrow B \implies S \Downarrow B) \tag{2}$$

Leroy defines S **safe** as meaning that "none of the possible behaviours of S is a 'going wrong' behaviour"[1]. Therefore we can define this "weakened" notion as if S, the source program, does not go wrong, then neither does C, the compiled program. Leroy then states that the CompCert compiler focuses on source and target languages that are deterministic. Then with this considered Leroy introduces the idea that in the conditions that both the source and target languages are deterministic that we can prove (2) is equivalent to:

$$\forall B \notin \textbf{Wrong},\ S \Downarrow B \implies C \Downarrow B \tag{3}$$

Where **Wrong** is the "set of 'going wrong' behaviors" [1]. Leroy states that this property is generally easier to prove with a proof by induction on the execution of S, the source program.

2.2. Verified compiler

Leroy then introduces some approaches to establish that a given compiler preserves the semantics of the source code as discussed in section 2.1. Leroy models the compiler as a total function *Comp* from a source program S to either code that compiles,

$(Comp(S) = \mathbf{OK}(C))$

or code that throws a compile-time error,

$(Comp(S) = \mathbf{Error})$

Leroy states that a compile-time errors are when the compiler cannot produce code, such as syntax or type errors, but also introduces an important caveat, a compile-time error is also thrown if the code exceeds the scope or capabilities of the given compiler.

This caveat becomes important when discussing Leroy's CompCert, as mentioned previously, the compiler supports "a large subset of the C language" [1]. This subset excludes operators related to structs and unions, unstructured statements such as **goto**, **switch**, and **longjmp**, block-scoped local variables, and **static** variables. Further information on this subset can be obtained from Blazy et al. [6].

Leroy states that a compiler *Comp* can be to be verified if has a formal proof of the property:

$$\forall S, C, \quad Comp(S) = \mathbf{OK}(C) \implies S \approx C \tag{4}$$

Or that a fully verified compiler either throws an error or produces a compiled program that satisfies the desired correctness property. Leroy states that a compiler that always fails:

$$\forall S,\ Comp(S) = \mathbf{Error} \tag{5}$$

Does verify, even if it is useless. Leroy states that no matter if the compiler succeeds in compiling the source programs is not actually an issue of correctness, but an issue of the quality of implementation. This is important to the testing of other compilers as Leroy states [1] that a quality of implementation issue must be addressed by "non-formal methods such as testing" [1]. Leroy then states that the most important feature of a verified compiler is that it does not silently produce incorrect code which can only be *guaranteed* with formal verification.

2.3. Further useful definitions

We can define some useful properties of compiled code from definitions in section 2.1. Firstly using the same definitions from section 2.1,

$$\forall B,\ \neg(S \Downarrow B \iff C \Downarrow B) \tag{6}$$

We can introduce this definition. This describes the case of a compiler bug, or when the compiler produces compiled code that exhibits different observable behaviours than the source code that was used to create the compiled program. This definition is a clear extension of (1) where the equivalence is negated to account for the observable behaviours differing.

Next we work with (2) and modify this definition to work for unsafe behaviour.

$$S\ \mathbf{unsafe} \implies \neg(\forall B,\ C \Downarrow B \implies S \Downarrow B) \tag{7}$$

We can define S **unsafe** as the opposite of S **safe** from (2) or that some of the behaviours of S are 'going wrong' behaviours. We can

further modify this definition as to make it easier for us to use:

$$S \textbf{ unsafe} \implies (\exists B,\ C \Downarrow B \implies\!\!\!\!\!\!/\ \ S \Downarrow B) \quad (8)$$

Or that S **unsafe** as if there exists an observable behaviour where the behaviour of the source code does not match the behaviour of the compiled code.

We can define a compiler introduced bug as (8), as a compiler introduced bug can be described as when the observable behaviours of the source code, in this case S, do not match the observable behaviours of the compiled executable, C. Note the use of "observable behaviour", in the case of nonmatching unobservable behaviour UB:

$$\forall UB,\ C \Downarrow UB \implies\!\!\!\!\!\!/\ \ S \Downarrow UB \qquad (9)$$

Where for all the unobserved behaviours belonging to compiled code do not imply all the unobserved behaviours in the source code. We can also slightly modify our definition by saying that for all the unobserved behaviours belonging to compiled code do not *necessarily* imply all the unobserved behaviours in the source code. Whereas as (1) states that observed behaviours in the compiled program *must* imply observed behaviours in the source program.

2.4 TEMP TITLE

Overall the CompCert compiler is an interesting project and has information that could be used in the testing and debugging of a compiler it uses a formal verification approach to the problem and there are some issues with this approach if it was to be applied to debugging the 42 compiler. If Leroy's approach [1] was to be applied to a different language rather than the subset of C that CompCert compiler uses some issues may arise. Firstly the compiler produces code only for PowerPC, a RISC instruction set architecture (ISA). Targeting other ISAs may prove to be more difficult especially if dealing with a non-RISC [7] ISA. This would be the case with most 42 programs which would presumably primarily target the x86 ISA.

Secondly since the CompCert compiler only works for a subset of C, it may be more difficult to produce a verified compiler for other languages especially considering only a subset of C is supported and C is considered to a fairly simple language [8]. An additional considering while considering language complexity and its relationship to the difficulty of verifying a compiler is that 42 is purely object-orientated language [9]. While C can be used in an object-orientated way [10], it does not support some more advanced object-orientated features such as inheritance. These more advanced object orientated features that are present in 42 and 42's support for metaprogramming would mean that it would be extremely difficult to create a verified compiler for 42.

Bibliography

[1] X. Leroy, "Formal Verification of a Realistic Compiler," *Commun. ACM*, vol. 52, pp. 107–115, July 2009.

[2] J. Harrison, "Formal verification at intel," in *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pp. 45–54, June 2003.

[3] J. O'Leary, "Formal verification in intel cpu design," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pp. 152–, June 2004.

[4] T. Abdellatif and K. L. Brousmiche, "Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, Feb 2018.

[5] J. Mccarthy and J. Painter, "Correctness of a compiler for arithmetic expressions," pp. 33–41, American Mathematical Society, 1967.

[6] S. Blazy, Z. Dargaye, and X. Leroy, "Formal Verification of a C Compiler Front-end," in *Proceedings of the 14th International Conference on Formal Methods*, FM'06, (Berlin, Heidelberg), pp. 460–475, Springer-Verlag, 2006.

[7] D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," *SIGARCH Comput. Archit. News*, vol. 8, pp. 25–33, Oct. 1980.

[8] B. W. Kernighan, "A programming language called C," *IEEE Potentials*, vol. 2, pp. 26–30, Dec 1983.

[9] M. Servetto, "42- metaprogramming as default."

[10] A. Schreiner, *Object-Oriented Programming With ANSI-C*. Hanser, 1994.

[11] F. Arnaboldi, "Exposing Hidden Exploitable Behaviors in Programming Languages Using Differential Fuzzing," p. 19.

[12] M. A. Dave, "Compiler Verification: A Bibliography," *SIGSOFT Softw. Eng. Notes*, vol. 28, pp. 2–2, Nov. 2003.

[13] X. Leroy, "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, (New York, NY, USA), pp. 42–54, ACM, 2006.

[14] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to Prioritize Test Programs for Compiler Testing," in *2017 IEEE/ACM 39th International Conference on Software En-*

gineering (ICSE), pp. 700–711, May 2017.

[15] T. Schlipf, T. Buechner, R. Fritz, M. Helms, and J. Koehl, "Formal verification made easy," *IBM Journal of Research and Development*, vol. 41, pp. 567–576, July 1997.

[16] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An Empirical Comparison of Compiler Testing Techniques," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 180–190, May 2016.

[17] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 266–277, April 2016.

[18] M. Servetto and E. Zucca, "A metacircular language for active libraries," *Science of Computer Programming*, vol. 95, pp. 219 – 253, 2014. Selected and extended papers from Partial Evaluation and Program Manipulation 2013.

[19] R. Böhme and S. Köpsell, "Trained to accept?: A field experiment on consent dialogs," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, (New York, NY, USA), pp. 2403–2406, ACM, 2010.