Runtime Verification of Memory Safety via Source Transformation

Zhe Chen, Junqi Yan, Wenming Li, Ju Qian, and Zhiqiu Huang
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
Nanjing, Jiangsu, China
zhechen@nuaa.edu.cn

ABSTRACT

The unsafe features of C often lead to memory errors that can result in vulnerabilities. Many runtime verification tools are widely used to detect memory errors. However, existing tools lack DO-178C compliance, show limited performance, and demonstrate poor accessibility, e.g., lacking platform-independence. In this paper, we propose to implement dynamic analysis tools using source-to-source transformation, which operates on the original source code to insert code fragments written in ANSI C, and generates source files similar to the original files in structure. We show that source transformation can effectively avoid the mentioned drawbacks of existing tools, but it also faces many new challenges in implementation.

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Software and its engineering → Software reliability**; **Software safety**; **Software testing and debugging**;

KEYWORDS

runtime verification, dynamic analysis, memory safety, C programs, source-to-source transformation

ACM Reference Format:
Zhe Chen, Junqi Yan, Wenming Li, Ju Qian, and Zhiqiu Huang. 2018. Runtime Verification of Memory Safety via Source Transformation. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3194962

1 INTRODUCTION

C is widely used for implementing systems software and embedded software which are usually security or safety critical systems, e.g., avionics systems. Unfortunately, the unsafe language features, such as the low-level control of memory, often lead to memory errors that can result in vulnerabilities. Along with the development of runtime verification [6, 11], dynamic analysis tools are widely used to detect memory errors. These tools are implemented via instrumentation at different levels of software systems.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3194962

(1) IR-level (Intermediate Representation level) including the Single Static Assignment (SSA) form [8, 12] and the C Intermediate Language (CIL) form [9, 10]. These tools are usually integrated with a host C compiler to instrument during compilation. They only need to handle simple statements, as source code has been significantly simplified by the time they run. But such tool is inherently tied to a specific host C compiler and its supported architectures.
(2) Binary-level [13]. These tools directly instrument the binaries after compilation. But this approach usually causes high runtime overhead, because it is difficult to perform high-level optimizations. Besides, binary rewriting is also inherently tied to a specific instruction-set architecture.
(3) Hardware-level [7]. But this needs special hardware support which may be not pervasively available.

However, existing tools lack DO-178C compliance, show limited performance, and demonstrate poor accessibility, e.g., lacking platform-independence. In this paper, we propose to implement dynamic analysis tools using source-to-source transformation. As we will show, source transformation can effectively avoid the mentioned drawbacks of existing tools. But it also faces many new challenges in implementation.

2 SOURCE TRANSFORMATION & BENEFITS

The general idea of source-to-source transformation is to implement existing checking algorithms via source-level instrumentation. That is, it operates on the original source code to insert code fragments written in ANSI C, and generates source files similar to the original files in structure. Compared with existing approaches, we can reap the following benefits by using source transformation.

DO-178C Compliance. Existing tools compile instrumented code into executable binaries, thus the source code after instrumentation is invisible. This is highly undesirable in developing safety-critical systems. For example, according to DO-178C, the de facto standard for developing avionics software systems [3], all tools used for DO-178C development must be part of the certification process, unless their output can be verified. According to its tool qualification requirement, if the output of a tool is part of the airborne software and thus could insert an error, then the tool should be determined as a TQL-1 level tool (the most rigorous Tool Qualification Level). This means, the tool must have been designed and developed following the strict process specified by DO-330 [4]. Unfortunately, all existing dynamic analysis tools fall into this category, as they insert embedded code fragments. However, none of them follow the DO-330 process, or are likely to do so.

Note that there is an exception that, if the output of any used tool can be verified in the DO-178C process, then the tool does not need to be certified. The DO-178C process requires strict reviews and analyses of source code during the software verification process, either by humans or using formal verification techniques [5]. However, most of verification techniques are performed on source code [1, 2], thus the generated binaries cannot be well verified. Furthermore, even if they generate source code, but if significantly differing from the original code in structure, then it will be hard to check its compliance with the low-level requirements, traceability and verifiability.

Thanks to source transformation, the source code after instrumentation is visible to developers, and can be further verified by following DO-178C. Thus it can be deployed in the system. When a memory error is detected at runtime, its cause can be reported or recorded. Note that the error is risky but not necessarily fatal, thus the error can be fixed in next upgrades.

Performance. The inserted code for tracking memory information and performing checks can cause runtime overhead. The performance of tools is partly limited by the representations they operate on. For example, IR-level instrumentation usually outperforms binary-level, as the original and inserted code can be optimized by compilers. However, we noticed that the inserted code at IR-level also cannot be fully optimized after instrumentation, as some optimization phases have passed by the time they run. Thanks to source transformation, we could further reduce runtime overhead by exploiting aggressive optimizations provided by compilers, both in front-end and back-end.

Accessibility. Existing tools are usually platform-dependent. Note that C is a typical cross-platform language, thus many developers need to deploy the instrumented software system either on various platforms, or on a special embedded system which is not general or even confidential, especially in the avionics or defense industry. Thanks to source transformation, the user can use the original deploy platforms, as the generated code is compiler-independent and platform-independent.

3 CHALLENGES IN IMPLEMENTATION

Unlike existing tools working on simplified IR, there are many new challenges in implementing dynamic analysis via source transformation. Here we list, not exhaustively, some of the difficulties that we will face when raise to source-level instrumentation.

- Side effects. Statements having side effects are quite common in C programs, e.g., pointer assignment "`p = q++`", pointer dereference "`*(p++)`", even together "`p = *(q++)`". The existing algorithms cannot handle these statements, as we cannot insert statements immediately before or after an inner expression. In contrast, such statements are split into several statements without side effects in the SSA form.
- Variants of unary dereferences. Programmers may dereference pointers via array subscript expressions or member expressions, e.g., "`(arr+i)[j]`" or "`(p+i)->m`". Their instrumentation obviously differs from unary dereferences. In contrast, such statements are replaced by pointer arithmetic and unary dereferences in the SSA form.
- Nested function calls, e.g., `f1(f2(p))`. The difficulty results from the expiring pointer value returned by the inner call `f2(p)`, whose address cannot be used to index and retrieve its metadata. In contrast, such statements are split into two separate function calls in the SSA form, and the return value of `f2` is explicitly stored in a new temporary variable.

The reader will find more unmentioned difficulties in practice.

4 TOOL AND FUTURE WORK

We have initiated a project to implement a source-to-source transformation tool, namely RVMS. Its novelty lies in that, to the best of our knowledge, it will be the first tool at real source-level, rather than pseudo source transformation at IR-level. Currently, it can handle only simple C statements. We are still seeking for a systematic and elegant solution for the mentioned challenges. We will also conduct experiments to validate the benefits of source transformation, especially performance as the other two benefits are obvious. We will also try to integrate RVMS into our runtime verification tool MOVEC [2].

ACKNOWLEDGMENTS

This work is supported by the Joint Research Funds of National Natural Science Foundation of China and Civil Aviation Administration of China (No. U1533130).

REFERENCES

[1] Zhe Chen, Yi Gu, Zhiqiu Huang, Jun Zheng, Chang Liu, and Ziyi Liu. 2015. Model Checking Aircraft Controller Software: A Case Study. *Software-Practice & Experience* 45, 7 (2015), 989–1017.
[2] Zhe Chen, Zhemin Wang, Yunlong Zhu, Hongwei Xi, and Zhibin Yang. 2016. Parametric Runtime Verification of C Programs. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016) (Lecture Notes in Computer Science)*, Vol. 9636. Springer, 299–315.
[3] RTCA DO-178C. December 2011. *Software Considerations in Airborne Systems and Equipment Certification.* RTCA, Inc.
[4] RTCA DO-330. December 2011. *Software Tool Qualification Considerations.* RTCA, Inc.
[5] RTCA DO-333. December 2011. *Formal Methods Supplement to DO-178C and DO-278A.* RTCA, Inc.
[6] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303.
[7] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *39th International Symposium on Computer Architecture (ISCA 2012).* IEEE Computer Society, 189–200.
[8] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010.* ACM, 31–40.
[9] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (2005), 477–526.
[10] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, John Launchbury and John C. Mitchell (Eds.). ACM, 128–139.
[11] Grigore Rosu, Wolfram Schulte, and Traian-Florin Serbanuta. 2009. Runtime Verification of C Memory Safety. In *Proceedings of the 9th International Workshop on Runtime Verification, RV 2009 (Lecture Notes in Computer Science)*, Vol. 5779. Springer, 132–151.
[12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA.* USENIX Association, 309–318.
[13] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the 2005 USENIX Annual Technical Conference.* USENIX, 17–30.