Project Status Report: Secure Software Validation Tools – Automatic Detection of Exploitable Buffer and Integer Overflow using Information Flow Tracking

Performers

Muhammad Monir Hossain, Mark Tehranipoor (PI)

1. Introduction

Buffer and integer overflows are some of the most available software vulnerabilities that have critical security impacts due to numerous triggering conditions, tampered memory boundaries, and a lack of scalable verification methods. An adversary can trigger an exploitable overflow via user-defined inputs. Although several existing tools can detect buffer and integer overflow vulnerabilities to some extent, the major limitations remain in the domain of an excessive number of false positives for static analyzers and poor detection coverage by dynamic methods due to limited vulnerability triggering input vectors. Hence, tracking the user-accessible input and performing comprehensive bound checking of buffers and arithmetic operations can fortify a program suffering from buffer and integer overflow, respectively. Under this task, we proposed an automatic framework leveraging information flow tracking (IFT) and the insertion of boundary checking assertions to detect these main attack vectors exploiting buffer and integer overflow.

2. Proposed Workflow

The proposed framework consists of three major operational stages- (1) information flow tracking (IFT) analysis, (2) assertion generation, insertion, and instrumentation, and (3) formal verification and vulnerability detection, as shown in Figure 1. At first, the user C program is compiled using Clang/LLVM compiler infrastructure [1], and the LLVM IR code is generated. The compiled LLVM IR code is analyzed to find taint propagation for all instructions. For efficient IFT analysis, we developed taint propagation logic for LLVM IR instruction set architecture (ISA). In the first stage, IFT analysis tool detects all vulnerable/tainted instructions and extracts array, pointer, vulnerable API, or arithmetic operation-related information required to generate an assertion in the next phase. In the next phase, we insert assertions for tainted instructions where we find the possibility of buffer overflows or arithmetic overflows that may cause potential buffer and integer overflow, respectively.

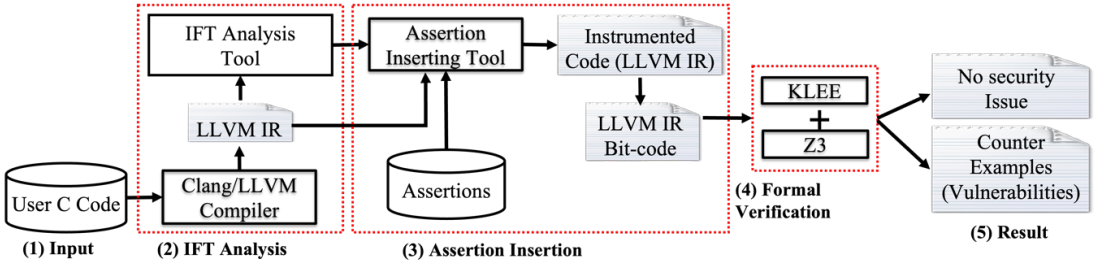

Figure 1: Implementation workflow of proposed framework for buffer and integer overflow detection.

Finally, we provide the instrumented LLVM IR with necessary assertions to a symbolic execution engine. The violations of boundary assertions prove the absence of boundary checking in arrays, pointers, vulnerable APIs, and arithmetic operations, where instructions are already tainted or vulnerable. As

assertions are inserted for tainted instructions, violations of assertions indicate potentially exploitable buffer overflow vulnerability in case of array-related overflows and integer overflows where an arithmetic operation is involved. Below, we describe the three operational stages indetails.

2.1 Automatic IFT Analysis Tool

In this framework stage, we observe each instruction and analyze the possibility of taint propagation from operands to the resultant variable through our developed IFT analysis tool in C# in static time. For accurate taint propagation, we define taint propagation logics for LLVM IR instruction set architecture [2]. We consider both types of information flow, i.e., explicit and implicit. In the previous report, we describe some taint propagation logics. We analyze the LLVM IR code through the program's call graph as shown in Figure 2. When there is any tainted instruction, we extract all required information from the code to generate assertion for the next stage.

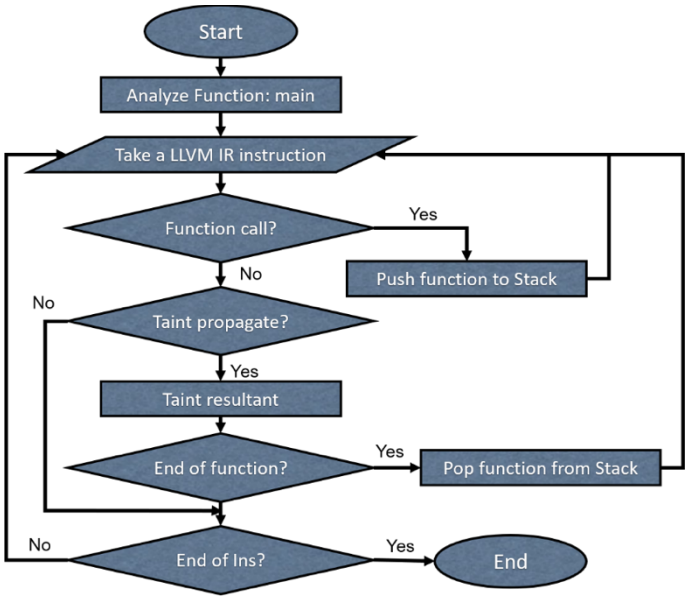

Figure 2: Information Flow Tracking (IFT) analysis workflow.

2.2 Automatic Assertion Insertion (AAI)

We developed a tool, *automatic assertion insertion* (AAI), to generate and insert assertions to the program in an automated fashion. This tool takes the user LLVM IR program under verification and inserts assertions for checking array and arithmetic boundary. The workflow of AAI is shown in Figure 3. AAI takes an LLVM IR instruction one by one and detects the external input source, which is considered untrusted data, and therefore it becomes tainted and made symbolic. When AAI finds a vulnerable instruction determined in IFT analysis phase, it inserts the assertion for checking the buffer boundary or arithmetic boundary based on the type of instruction. For inserting an assertion, AAI first inserts a *call* instruction and then provides a function definition in the LLVM IR code, which holds the assertion statement. The outcome of AAI is the instrumented program with necessary assertions of a user C program.

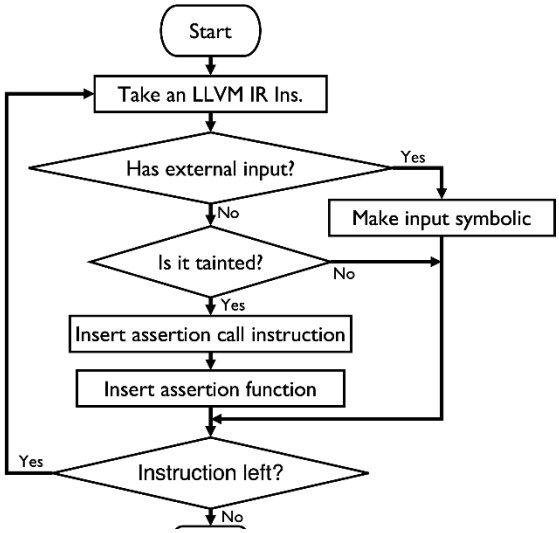

Figure 3: Workflow of automatic assertion generation and insertion.

2.3. Formal Verification and Vulnerability Detection

In this stage, the final instrumented LLVM IR program is provided to a symbolic execution engine for verifying the assertions. We utilize KLEE symbolic execution engine [3] to run the compiled binary of the instrumented LLVM IR code. During instrumentation, we define some external inputs as symbolic for which KLEE traverses all paths symbolically and propagates the taints from the sources to the resultant operand. KLEE generates constraints for all possible control paths and provided assertions, which are verified by an SMT solver (Z3) integrated into KLEE. We conclude whether a particular array, pointer, or API may cause an exploitable buffer overflow, or an arithmetic overflow may cause exploitable integer overflow from the violation of correspondent boundary assertions.

3. Benchmark Results

We evaluate the framework implemented for buffer overflow detection using the SAMATE Juliet benchmark suite from NIST [4]. The experimental result summary is shown in Table 1, which describes the number of successful detections, false alarm, and run-time, and memory overhead for the proposed framework. We observe that the framework can detect most of the buffer overflows successfully (37 out of 39 cases) with an average of ~94.87. The framework does not cause any false positive, which reflects the accurate assertions generation and taint propagation.

Table 1: Benchmark results for proposed framework.

Metrics	**Vulnerability ID**				
	CWE121 (#TC- 10)	CWE122 (#TC- 10)	CWE124 (#TC- 7)	CWE126 (#TC- 7)	CWE127 (#TC- 5)
Successful Detection	9	10	7	6	5
False Positive	0	0	0	0	0
Avg. Run-time (Assertion Violation)	0.72s	0.67s	0.71s	0.65s	0.64s
Avg. Instrumented Code Size (Bin)	1.62x	1.79x	1.68x	1.72x	1.65x

NB: Currently, we are doing experiments extensively for integer overflow and heap-based buffer overflow (Results to be shown in next sponsor meeting).

4. Challenges

So far, we have faced several challenges for providing a comprehensive solution against exploitable buffer and integer overflow using information flow tracking. Most of the technical challenges have been resolved using novel approaches and efficient modeling of the bound checking assertions. The regular updates and meetings with sponsors have been very helpful in this regard.

We list a few pending challenging items below that we plan to resolve with continuous development.

- Coverage of all LLVM IR instructions for information flow tracking. (Status: requires continuous development.)
- Detection of all corner cases for assertion generation. (Status: requires continuous development.)
- State space explosion for symbolic execution engine. (Status: work in progress.)

5. Current and Future work

We have completed the implementation for both buffer and integer overflow detection methodologies. Currently, we are doing extensive benchmark testing for both buffer and integer overflows and bug fixing for the developed tools. Our future work includes (1) establishing a higher coverage for LLVM IR instructions targeting information flow tracking, (2) provide more test and corner cases for assertion generation, and (3) ensuring control and data flow integrity, with a minimal state-space explosion for symbolic execution.

6. References

1. Lattner, Chris. "LLVM and Clang: Next generation compiler technology." The BSD conf. 2008.
2. https://llvm.org/docs/LangRef.html
3. Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." OSDI. Vol. 8. 2008.
4. https://samate.nist.gov/SARD/testsuite.php