Research Report: Analysis of Software for Restricted Computational Environment Applicability

Jacob I. Torrey and Jonathan Miodownik
Assured Information Security
Greenwood Village, CO, USA
{*torreyj, miodownikj*}*@ainfosec.com*

***Abstract*—Preliminary experiment design and research goals are presented to measure the applicability of restricted computational complexity environments in general purpose development efforts. The Linux kernel is examined through the lens of LangSec in order to gain insight into the make-up of the kernel code vis-à-vis the complexity class of recognizer for input to each component on the Chomsky Hierarchy. Manual analysis is assisted with LLVM Passes and comparison with the real-time Linux fork. This paper describes an on-going effort with the goals of justifying further research in the field of restricted computational environments.**

***Keywords*-Walther recursion, Linux, programming languages, parsing, LLVM, language-theoretic security, LangSec.**

I. Introduction

Language-theoretic security (LangSec) provides two fundamental guides for reducing the prevalence and impact of software vulnerabilities: formal parsing of input and restriction on computational complexity for the environments that perform operations on untrusted data. LangSec sketches a unified view of software exploitation, with the concept of a "weird machine": an *ad hoc*, emergent virtual machine that converts input data into execution flow with unexpected states and state transitions [1].

This leads to the realization that the data passed to the application is in fact the program that executes on the machine instantiated by the code. If these machines can be limited in their complexity and power, reference monitors and formal verification can improve the security when exposed to attacker-controlled input. In a previous work [2], the authors have shown empirical evidence of the improvement of verification; this paper aims to outline the on-going experiments to determine the applicability of restricted computational environments and categorize software components by their challenge to constrain.

A. Contributions

In [2], the sub-Turing complete programming language *Crema* was shown to ease verification and ensured programmers better provided their intent into implementation, reducing the risk of certain classes of attack [3] [4]. This effort aims to determine how applicable these programming design patterns are to general purpose programming, and which classes of software modules require more computational power than others. This research report describes an on-going effort, and as such is not complete at time of writing.

II. Problem Specification

A. Background

The following sub-sections provide a brief background to the concepts built-upon in this work.

1) Software Tools for Our Case Study:

LLVM and Clang: The LLVM compiler framework [5] is a tool-chain of modular components to analyze, optimize, compile, and execute programs via a standardized byte-code intermediate-representation (IR). Front-ends parse an input language, construct an abstract syntax tree (AST), and emit LLVM IR, which then can leverage the existing optimization passes, a cross-platform just-in-time compiler, and static analysis tools to allow for rapid compiler development. One such front-end parser for the C and C++ programming languages is Clang, which transforms source files into an LLVM AST for generation of IR, bit-code or target binaries.

Once an input program has been converted to the IR, there are a number of tools and libraries that can then be used to optimize the IR for faster execution or smaller memory footprint. Chiefly of interest to readers of this paper is the *Pass*, a construction for modules that can navigate the AST either for analysis or modifications. Passes provide a powerful API into the program's AST and can specify which AST elements to inspect or modify: viz., loops, functions, modules, etc.

Linux Kernel: As a representative software project, the Linux kernel handles a large number of tasks and is highly complex. In 2001 the kernel had approximately 2.4 million source lines of code (SLOC) and has been grown rapidly in order to support multiple CPU architectures and a multitude of hardware devices. For this research a pared-down version[1] of the kernel was chosen, one supporting only the Intel x86 architecture. Additionally, device-specific drivers were deemed out of scope, whereas those for a general interface (e.g., USB 2.0 or PCI) were left in.

[1]The kernel was configured and built with the target `tinyconfig`

B. LangSec-inspired Challenges

LangSec highlights the risks involved with parsing input into a program's internal type-system and demonstrates how a poorly designed or implemented parser creates a risk of unintended computations. The theory goes further and calls for input languages to be as restricted as possible, urging programmers to utilize the minimum amount of computational expressiveness necessary to validate inputs. This paper specifically examines the feasibility for software developers to incorporate these restricted computational environments into general purpose applications or to refactor existing code bases into safer programs. A future area of study is the ability to rapidly assist developers with such refactoring attempts in order to aid them in shipping more robust and less vulnerable applications.

III. Approach

The goals of this effort, when abstracted are equivalent to the Halting Problem [6] — determining whether or not a general algorithm (equivalent to a Turing Machine) will halt on a given input, or if there are inputs that the algorithm would never halt on. The scope of this effort is such that perfect certainty is not expected, instead components will be grouped into three categories: 1) provably sub-Turing complete (i.e., in the Walter Recursion complexity class [7]), 2) possibly sub-Turing with human-in-the-loop refactoring to remove edge cases that should never be hit [2], and 3) components that require the full expressiveness of Turing-completeness. The second group is probabilistic, the component has the design patterns of others than are sub-Turing or could be made so, but without a human-in-the-loop knowledgeable of that sub-system or module, no guarantees can be made.

In order to side-step the Halting Problem, there are a number of methods the authors foresee utilizing to gather a reasonably complete picture of the Linux kernel's computational power needs. Each of these is described below in more detail:

A. Low-Hanging Fruit

The fastest method to rapidly classify a large number of components is with an LLVM Pass designed to mark every function as sub-Turing that either: does not contain any looping construct, recursive call or co-recursive call; or only contains looping constructs that fall within the bounds of Walter recursion — meaning termination is provable. These loops may already be candidates for unrolling during optimization or have fixed bounds. This low-hanging fruit will quickly separate the kernel code into the "easy" sections that can be ignored by later steps and those that warrant additional research.

B. RT-Linux Analysis

Once the obvious functions have been classified, an examination of the Linux real-time (RT) fork will be performed [8]. The RT fork aims to produce a Linux kernel that can support hard real-time demands for use in safety-critical applications. Through considerable developer effort, the RT kernel has been modified where the core scheduling loop passes execution to tasks given a fixed amount of time to complete their processing. Through examination of the core scheduling loop and associated routines, a set of components that should demand unbounded looping and can confidently be placed in the third grouping of components.

C. Assisted-Manual Analysis and Heuristics

Once the low-hanging fruit have been sorted into the first and third category, the remaining routines will be those that require more intensive analysis. From this point, human-in-the-loop analysis is expected to manual classify the functions remaining. As each function is classified, the abstract syntax tree (AST) patterns for the function or component will be combined the classification to train a binary classifier to make an approximate "guess" for the remaining components not clearly in first or third group. It is expected that as the human-in-the-loop process continues, new patterns of sub-Turing complete functions will be detected, and these patterns fed into a LLVM Pass to classify all similar functions into the first group.

IV. Experiment Setup

The modern Linux 4.x kernel will be utilized as well as LLVM and clang-3.8. A number of patches to the build settings and Makefiles must be made in order to coerce the kernel to build with clang. Additionally there are a few assembly files with GNU assembler-specific macros that must be expanded in order for the build to succeed. The kernel is configured with `make tinyconfig` to generate the target minimal set for processing and analysis.

With a kernel building with clang-3.8, the build scripts will be modified to ensure the compiler outputs LLVM bitcode intermediate representation of the AST to disk to be analyzed by the LLVM Passes developed under this effort. As the kernel has a complex build process, each LLVM Pass writes the findings as machine-readable JSON files to be consumed by a final pass to collate and perform analytics on the output from multiple sub-systems and build directories.

A. Results and Discussion

As this is a research report on an on-going effort, the results are not ready at this time. A camera-ready report would contain the latest findings. Interpreting the findings of this effort will be aided through the use of the following metrics:

[2] A concrete example of this is the portable document format in which normal operations of parsing should always terminate, however the specification for the format allows for unbounded looping. If refactored, a PDF parser would be able to correctly parse well-behaved inputs, and reject the edge-cases indicative of malicious behavior.

Raw Percentage SLOC By Group: The most apparent measurement for this effort is the raw percentage of the source lines of code (SLOC). The build scripts for the kernel have been modified to track dependencies for all compilation steps, then de-duplicate across every object built, followed by analysis of total SLOC by the `sloccount` utility [9]. Once total SLOC has been calculated, the functions in each group have their respective SLOC counts analyzed.

This metric is valuable in making a case for considering LangSec during design of development projects; if the percentage of functions in the third group is minimal, architects and developers can write most of their code with sub-Turing environments in mind, with full Turing-completeness as a outlier left to senior developers. Raw percentage does not provide the full picture to truly make a case for changing software design paradigms, the benefits of such a shift must be taken into careful consideration.

Profiled Execution Percentage: To ensure any effort to change software design is a valuable contribution w.r.t. the safety of developed applications, a stronger case must be made through another metric. If the findings from this effort show that a sizable majority of the code analyzed could be executed in a constrained computational environment, but these regions of code are seldom executed compared to the full Turing-complete functions, the safety benefits would be diminished. In order to measure this, the kernel will be profiled under a few typical workloads [10] to categorize which routines are most commonly executed and this mapped to the aforementioned groupings. This measurement provides an approximation of the impact of constraining functions from groups one and two towards the safety of the entire program.

The well-known $Risk = Threat \times Vulnerability \times Consequence$ formula [11] is used in analyses to attempt to objectively categorize risks facing an organization or security posture. By using a profiled view of the kernel, the authors aim to provide a better metric for risk of software compromise (assuming the consequence of remote code execution in the kernel is constant across the code). This metric is imperfect in that it fails to determine whether the code that may have vulnerabilities or is in a more powerful computational group is *reachable* by attacker-controlled input. In future efforts, exploring methods to better categorize attacker access would provide a better dataset than scheduling time on the CPU. An example could be a driver that only responds to certain types of device messages (e.g., USB packets) and may not be scheduled to run very often, though if an attacker could craft a malicious input, it would be able to compromise the kernel.

V. Conclusion

This work aims to measure the feasibility of developing general purpose applications with a closer eye towards the computational complexity of each component. By measuring the make-up of a representative software program traditionally considered "complex" (the Linux kernel), insight into the trade-offs for further research and changing development practice are realized. The authors posit that, for many components of software projects, the fully expressive, Turing-complete environment is overly powerful and carries a significant (and realized) risk of compromise. Through the use of a restricted execution model, software safety can be increased and software design paradigms altered to consider the benefits of a LangSec-inspire approach.

Acknowledgments

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) on the STAC program[3].

References

[1] J. Vanegue, "The weird machines in proof-carrying code," in *Proc. First Annual Langsec Workshop*, May 2014.

[2] J. Torrey and M. Bridgman, "Verification state-space reduction through restricted parsing environments," in *Security and Privacy Workshops (SPW), 2015 IEEE*, May 2015, pp. 106–116.

[3] M. Dowd. (2003) Sendmail release notes for the crackaddr vulnerability.

[4] J. Vanegue, S. Heelan, and R. Rolles, "Smt solvers for software security," in *Proceedings of the 6th USENIX Conference on Offensive Technologies*, ser. WOOT'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2372399.2372412

[5] C. Lattner. (2015) The LLVM compiler infrastructure. [Online]. Available: http://llvm.org/

[6] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[7] C. Walther, "Security applications of formal language theory," *Artificial Intelligence*, vol. 70, no. 1, 1994.

[8] L. R. Team. (2015) Rtwiki. [Online]. Available: https://rt.wiki.kernel.org/index.php/Main_Page

[9] D. A. Wheeler. (2002) More than a gigabuck: Estimating gnu/linux's size. [Online]. Available: http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html

[10] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[3]The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

[11] L. A. T. Cox, Jr, "Some limitations of risk = threat vulnerability consequence for risk analysis of terrorist attacks," *Risk Analysis*, vol. 28, no. 6, pp. 1749–1761, 2008. [Online]. Available: http://dx.doi.org/10.1111/j.1539-6924.2008.01142.x