The New Era of Semantic Operating Systems

Charly Castes
EPFL
Switzerland

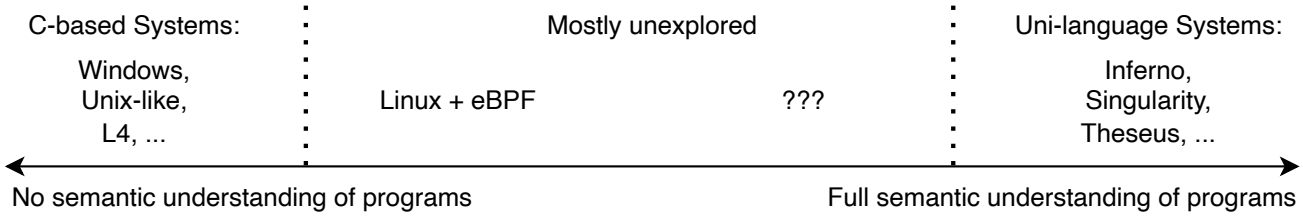

Figure 1. The semantic operating systems spectrum

Abstract

The design of operating systems in wide use today is rooted in systems originally developed in the seventies, and hence inherits design decisions constrained by the available tooling of that time. In particular, today's operating systems lack understanding of programs, leading to the need for costly isolation, unstructured interfaces between software components, and lack of flexibility of execution. Yet, programming language design and compiler toolchains made stunning progress during the last two decades and opened the door to radically different designs for a new generation of operating systems. In fact, such designs already emerge in both research and widely used legacy systems, achieving better performance with increased reliability.

We claim that the understanding of code semantics will continue to make its way into current and future systems. In this proposal, we look at existing system through the lenses of their understanding of program semantics and highlight a mostly unexplored design area. We identify crucial operating system components required to leverage program semantics and propose extensions that could benefit both commodity operating systems and experimental uni-language systems.

1 Introduction

The original Unix system was developed more than 50 years ago, initially in assembly and then rewritten in C in 1973 [10], marking the beginning of the fascinating co-design between the language and the operating system [6]. Nowadays, C is the *lingua franca* of programming languages and often the only interface to the system, while the way programs are assembled, executed, and isolated is largely identical to the original design. A fertile ecosystem has emerged out of our current generation of systems, that has lead to the development of game-changing theories and technologies for building software. Yet, these technologies have not been integrated back into the systems that saw their birth, leading to an awkward feeling that our systems are lagging behind our tooling in terms of software distribution, safety, reliability, and achievable performance. We argue that at its core, the gap is due to the lack of understanding of program semantics in our current operating systems.

Looking at existing operating systems through the lenses of their understanding of program semantics reveals two broad families: traditional C-based systems that view programs as opaque execution contexts and uni-language systems that leverage full understanding of program semantics.

Semantically, C programs have full access to the machine resources, including memory and the whole instruction set. For this reason, program-isolation on C-based systems, such as Windows and Unix descendants, can only be achieved by leveraging hardware mechanisms such as virtual memory and privilege levels. The micro-kernel design, in which some of the kernel services are outsourced to hardware-isolated servers, is a consequence of the unrestricted semantics of C. Because of the cost associated with context switches, the isolation boundaries (*e.g.* processes) are often coarser than code-modularity or trust boundaries (*e.g.* libraries).

An interesting alternative is adopted by uni-language (or *unilang*) systems, *i.e.* systems managed by a unified runtime or built from a single compiler to provide higher level abstractions, such as strongly typed interfaces and memory safety. Example of such systems include Inferno [3] and the Dis virtual machine, Singularity [4] and the Sing# compiler, and Theuseus [1] which leverages the Rust language. Unlike C-based systems, unilang systems can only execute programs in very restricted environments, but leverage those semantic restrictions to reap additional benefits such as running sandboxed programs without any hardware support and thus avoiding expensive context or mode switches. In fact, Inferno, Singularity, and Theseus all execute their processes in the same address space as the kernel itself.

We argue that C and unilang systems are the two ends of a semantic understanding spectrum (Figure 1), and that there

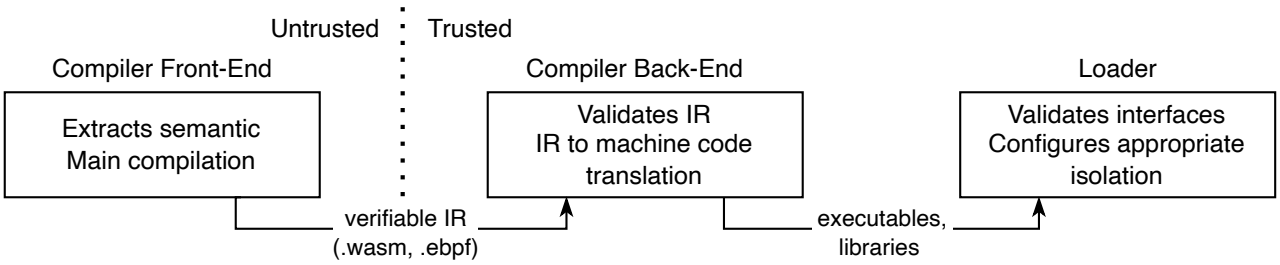

Figure 2. The semantic operating systems workflow

is room for more nuanced designs for both upcoming and already existing operating systems. In fact this transformation is already happening, as evidenced by the integration of eBPF in the Linux kernel. Indeed, Linux is now able to leverage program semantics to safely sandbox untrusted program execution, unlocking a whole range of applications through unmatched performance-security trade-off [13].

Nowadays the operating system semantic understanding spectrum is largely under-explored. We propose to close the ever growing gap between our aging systems and bleeding edge programming language toolchains by investigating how semantic understanding of software can be leveraged by key operating system components.

2 Proposed Work

Semantic understanding of programs can be leveraged to provide better safety, reliability, and performance, but the question of how it can be integrated into operating systems remains largely unexplored. We identify two key components that need to coordinate in order to leverage semantics: 1) the language toolchain, which understands and summarises the program's semantics, and 2) the program loader, that sets up appropriate isolation and performs adequate checks to guarantee correctness. We propose to explore the semantic operating system spectrum by both extending legacy C-based systems to leverage semantics and widening the scope of unilang systems to make them more practical.

2.1 A Trusted Language Toolchain

If semantics are used for isolation of untrusted programs, as in unilang systems, it becomes necessary to trust the software that enforces semantic restrictions. Usually, this is done by the compiler through static type-checking or injection of dynamic checks. Relying on a trusted compiler to produce valid programs is impractical as it prevents the OS from accepting the output of new compilers and languages.

Instead, we propose to draw inspiration from existing verifiable bytecode formats, such as NaCl [12], eBPF, and WebAssembly [11], where existing compilers behave as front-ends emitting a verifiable IR bytecode that gets compiled by a trusted back-end, as shown in Figure 2. The popularity of tools such as LLVM [7] makes this approach practical, creating the opportunity for unilang systems to open to more languages while enabling legacy systems to benefit from software sandboxing.

2.2 The Semantic Loader

Beside the compiler toolchain, a crucial piece on the path to program execution is the dynamic loader, *e.g.* ld.so on Linux, which can also benefit from leveraging program semantics. Indeed, the loader is responsible for craming libraries into a single address space, but could also be used to (1) enforce isolation boundaries between libraries and (2) perform extensive checks and data-layout transformations [9].

We propose to extend the dynamic loader to make use of semantic understanding to provide higher level guarantees and transformations, even on existing C-based systems. For instance, type-checking of dynamic libraries can be readily implemented by leveraging debug informations (*e.g.* DWARF), while extensive transformations such configuring hardware sandboxing [8] could be performed at runtime without program modification. Applications we have in mind include automatic partitioning with Intel MPK [5] and isolation of sensitive libraries with SGX [2].

3 Preliminary Results

We already explored how to better leverage program semantics on both legacy C-based and newer unilang systems, through the development of a Linux-compatible dynamic loader and a from scratch unilang system supporting execution of WebAssembly. Our dynamic loader is capable of type-checking dynamic libraries interfaces through DWARF debugging information stored in binaries, while our unilang kernel is able to sandbox and compose code written in any language that can output WebAssembly.

In the future, we would like to close the gap between both worlds by designing a standard format for defining interface and semantic guarantees, and porting it to standard toolchains. This will serve as a basis for demonstrating how semantic can benefits both C-based and unilang systems in terms of reliability, security, and efficiency, for both legacy binary and software built with a compatible toolchain.

References

[1] Boos, K., Liyanage, N., Ijaz, R., and Zhong, L. Theseus: an Experiment in Operating System Structure and State Management. In *OSDI* (2020), pp. 1–19.

[2] Costan, V., and Devadas, S. Intel SGX Explained. *IACR Cryptol. ePrint Arch. 2016* (2016), 86.

[3] Dorward, S. M., Pike, R., Presotto, D. L., Ritchie, D. M., Trickey, H. W., and Winterbottom, P. The Inferno™ operating system. *Bell Labs Tech. J. 2*, 1 (1997), 5–18.

[4] Hunt, G. C., and Larus, J. R. Singularity: rethinking the software stack. *ACM SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 37–49.

[5] Intel. Intel 64 and ia-32 architectures software developer's manual., 2022.

[6] Kernighan, B. W., and Ritchie, D. *The C Programming Language*. Prentice-Hall, 1978.

[7] Lattner, C., and Adve, V. S. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. pp. 75–88.

[8] Lefeuvre, H., Badoiu, V.-A., Jung, A., Teodorescu, S. L., Rauch, S., Huici, F., Raiciu, C., and Olivier, P. FlexOS: towards flexible OS isolation. pp. 467–482.

[9] Ren, Y., Zhou, K., Luan, J., Ye, Y., Hu, S., Wu, X., Zheng, W., Zhang, W., and Hu, X. From Dynamic Loading to Extensible Transformation: An Infrastructure for Dynamic Library Transformation. pp. 649–666.

[10] Ritchie, D., and Thompson, K. The UNIX Time-Sharing System. *Commun. ACM 17*, 7 (1974), 365–375.

[11] Rossberg, A., Titzer, B. L., Haas, A., Schuff, D. L., Gohman, D., Wagner, L., Zakai, A., Bastien, J. F., and Holman, M. Bringing the web up to speed with WebAssembly. *Commun. ACM 61*, 12 (2018), 107–115.

[12] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy* (2009), pp. 79–93.

[13] Zhong, Y., Wang, H., Wu, Y. J., Cidon, A., Stutsman, R., Tai, A., and Yang, J. BPF for storage: an exokernel-inspired approach. pp. 128–135.