R209 Essay: Correctness vs. Mitigation

Chongyang Shi (*cs940*)

November 20, 2017

1 Summaries of research

The paper by Klein et al. [1] discussed the design and construction process of a formally verified general-purpose L4 microkernel, *seL4*. In order to ensure the functional-correctness of the microkernel, thorough care had been taken by the authors to design seL4's programming model while minimising the introduced complexities in verification. A three-level refinement process was used – a Haskell-based executable model implements the full functionality of the microkernel's abstract specification, which is then optimised for performance in a C implementation. Isabelle/HOL was used in an interactive verification process to ensure functional-correctness. The resulting microkernel is comparable in performance to other L4 kernels, while imposing a reasonable materialistic cost in verification. As a potential improvement to the paper, more discussions on the impact of design decisions on the programmer's ability to write concise and efficient code for the kernel could be included.

The article by Bessey et al. [2] introduced notable observations and unique challenges faced by the authors in the process of commercialising a static analysis-based bug-finding tool. Their product applies an unsound traversal on the target codebase to find as many errors as possible. After the product was adapted for use on large commercial codebases, caveats including as integrating with different build systems, dealing with compilers and runtime environments not compliant with language standards, and working with misinformed users as well as commercial demands were identified by the authors, along with some attempts to address these challenges. A possible extension to this paper is discussion on whether code written in languages with tighter ecosystems such as Apple's Objective-C are easier for static analysis in the field.

The survey paper by Szekeres et al. [3] summarised common mitigations against memory corruption bugs, as well as performance overheads, compatibility issues, and attack vectors associated with each mitigation technique. Starting by introducing common types of memory corruption attacks, the authors first gave an overview on protection systems in use based on both probabilistic protection and deterministic protection. After discussing important metrics as well as tradeoffs in cost and compatibility, they further described how current systems measure in these aspects. They also covered defences against generic attacks and control-flow hijack attacks. The authors concluded by highlighting performance and compatibility constraints in current systems which prevent their wide adoption. A critique for this paper concerns on its lack of clarity in structure, such as the repeated description and discussion of stack smashing protection [3, III, VIII-B].

2 Key themes of research

2.1 Laboratory theory versus real-world practice

When a system or technique evaluates well under laboratory conditions, it may not perform as well or face operational difficulties "in the field". A assumption made during seL4's development process before 2009 is the uniprocessor runtime environment, which was required in order to model correctness without concurrency constraints [1, 3.3]. With modern processors featuring multiple cores, this is no longer feasible in performance aspects. On a different aspect, Bessey et al. [2, pp. 69-70] noted how compiler deviations from language standards caused difficulties to the one-solution-fits-all research model. Szekeres et al. also described the lack of industrial adoption of memory corruption defences due to their significant performance penalties and incompatibilities [3, I, IV-C].

2.2 Security versus cost

A source of strong resistance against wider adoption of error verification and mitigation is the potential cost involved for stakeholders. Fundamental redesigns to existing kernel features are very costly to verify under the verified microkernel by Klein et al. [1, 5.3], which is unfortunately a common occurrence in the industry [4]. Bessey et al. [2, p. 73] noted how upgraded systems leading to more bugs being detected can cost the stakeholder managers their bonuses. On performance aspect of cost, Szekeres et al. [3, V-A] raised the example of a 10% performance penalty preventing most programs from being compiled as Position Independent Executables (PIE).

2.3 Benefits of using an intermediate representation

A more subtle theme seen across the three papers is the potential benefits of using intermediate representations. Klein et al. [1, 2.2, 5.2] used a Haskell prototype as an executable model prior to optimisation, which saved significant cost in verification and allowed them to take advantage of existing toolchains and experience. When analysing C# and Java bytecode compiled from the source, Bessey et al. [2, p. 72] had a much easier time without having to navigate compiler deviations. The technique providing the highest level of memory corruption protection (SoftBound+CETS) represented new objects with unique identifiers to improve memory performance [3, VI-C].

3 Ideas in current context

In order to address the uniprocessor limitations of seL4 as discussed, a followup research by Tessin [5] presented the *clustered multikernel* approach in order to provide verifiable multiprocessor kernels. In addition, Tessin developed a lifting framework to transform verified uniprocessor microkernels to support multiprocessing with minor modifications. This approach saved a large amount of effort in developing a verifiable multiprocessor kernel from scratch.

Intermediate representations have been further utilised to improve program analysis to search for errors and security violations, as demonstrated by the Static Analysis Intermediate Language (SAIL) [6]. The two-level intermediary design of SAIL preserve both syntactical error reporting capabilities and the ability to perform semantical analysis to find errors. A similar intermediate representation scheme was used by Dullien and Porst [7] to perform platform-independent security auditing and vulnerability detection.

Bessey et al. [2] faced significantly problems when a varying range of C/C++ compilers fail to observe strict language standards, causing false-positive error reports. An alternative error-finding approach has since emerged, combining the classic bounded model checking (BMC) approach with LLVM intermediate representation to offset the complexities of modern language features introduced by C/C++ which previously made BMC difficult [8]. This approach is also able to find bugs introduced by compilers – another advantage addressing their complaints.

4 Literature review

While the correctness of seL4 microkernel was proved by Klein et al. [1] on the design level, its high-level security properties of confidentiality and integrity have also been proved by Murray et al. [9] in 2013. The seL4's memory management system based on capabilities was modelled upon by Baumann et al. [10] for their scalable multicore kernel, albeit without formal verification. In order to address potential security problems created by compiler errors during the compilation of verified C code such as seL4, Yang et al. [11] developed a method for detecting compiler bugs through randomised test-case generation.

Based on the difficulties faced by Bessey et al. [2] in performing static analysis on codebases, Chipounov et al. [12] proposed an alternative method of performing multi-path analysis directly on binaries. While their approach solves the source code access problem, their approach lacks error reporting capabilities due to the compiled nature of binaries. Johnson et al. [13] further investigated reasons of developers not using static analysis to find bugs, including false positives and developer overloading, and made their recommendations to address these issues. Bodden et al. [14] improved coverage of static analysis in Java by inserting runtime checks against unanalysed reflective calls.

Szekeres was involved a more recent work [15], which introduced design points of code-pointer integrity and code-pointer separation to supplement defences against control-flow hijack attacks. Schuster et al. [16] developed an alternative protection principle of *region self-integrity* for distributed MapReduce computations, in lieu of full memory safety protection, which is very computationally-expensive as measured by Szekeres et al. [3].

References

[1] G. Klein *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 207–220.

[2] A. Bessey *et al.*, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[3] L. Szekeres *et al.*, "Sok: Eternal war in memory," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 48–62.

[4] A. Israeli and D. G. Feitelson, "The linux kernel as a case study in software evolution," *Journal of Systems and Software*, vol. 83, no. 3, pp. 485–501, 2010.

[5] M. Von Tessin, "The clustered multikernel: An approach to formal verification of multiprocessor os kernels," in *Proceedings of the 2nd Workshop on Systems for Future Multi-core Architectures, Bern, Switzerland*, 2012.

[6] I. Dillig, T. Dillig, and A. Aiken, "Sail: Static analysis intermediate language with a two-level representation," *Stanford University Technical Report*, 2009.

[7] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis," 2009.

[8] F. Merz *et al.*, "Llbmc: Bounded model checking of c and c++ programs using a compiler ir," *Verified Software: Theories, Tools, Experiments*, pp. 146–161, 2012.

[9] T. Murray *et al.*, "sel4: from general purpose to a proof of information flow enforcement," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 415–429.

[10] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 29–44.

[11] X. Yang *et al.*, "Finding and understanding bugs in c compilers," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 283–294.

[12] V. Chipounov *et al.*, "S2e: A platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[13] B. Johnson *et al.*, "Why don't software developers use static analysis tools to find bugs?" in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 672–681.

[14] E. Bodden *et al.*, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 241–250.

[15] V. Kuznetsov, L. Szekeres *et al.*, "Code-pointer integrity." in *OSDI*, vol. 14, 2014, p. 00000.

[16] F. Schuster *et al.*, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 38–54.