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, the authors took very thorough care when designing seL4's programming model to minimise complexities introduced into verification. A three-level refinement process was used – a Haskell-based executable model implemented the full functionality of the microkernel, as defined by an abstract specification. The executable model was then optimised for performance by reimplementing in C. Isabelle/HOL was used in an interactive verification process to ensure the model's functional-correctness. The resulting microkernel is comparable in performance to other L4 kernels, and imposes a reasonable materialistic cost in verification. As a potential improvement to the paper, more discussions could be included on the impact of the programming model's design decisions on the programmer's ability to write concise and efficient code for the kernel.

The article by Bessey et al. [2] described notable observations and unique challenges faced by the authors in the process of commercialising a static analysis-based bug-finding tool. Their product applied an unsound traversal on the target codebase to find as many errors as possible. The authors identified some caveats in the process, including integrating with customised build systems, dealing with compilers and runtime environments not in compliance with language standards, and working with misinformed users as well as commercial demands. They also introduced some attempts their team made to address these challenges. A possible extension to this paper is discussions 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. For each mitigation technique, they described associated performance overheads, compatibility issues, and attack vectors. Starting by introducing common types of memory corruption attacks, the authors gave an overview on protection systems in use based on both probabilistic protection and deterministic protection principles. After discussing important metrics and tradeoffs in cost and compatibility, they described in more detail how current systems perform in these aspects. They also covered defences against generic attacks and control-flow hijack attacks. The authors concluded by highlighting performance and compatibility constraints remain in current systems, which are preventing their wide adoption. A critique for this paper concerns on its lack of clarity in structure. For example, description and discussion of stack smashing protection were repeated twice [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, inhibiting its wide adoption. An assumption made during seL4's verification process concerns the uniprocessor runtime environment [1, 3.3], which was still dominating the market at the time. With modern processors featuring multiple cores, seL4's applicability becomes severely constrained. Bessey et al. [2, pp. 69-70] noted how compiler deviations from language standards caused difficulties in their static analysis model, requiring significant adaptation. Szekeres et al. reasoned that slow industrial adoption of memory corruption defences was caused by significant performance penalties and incompatibilities associated [3, I, IV-C].

2.2 Security versus cost

Strong resistance to wider use of software verification and error mitigation comes from the potential cost involved for stakeholders. In the case of the verified microkernel by Klein et al. [1, 5.3], fundamental redesigns to existing kernel features are very costly to reverify. Kernel redesigns are unfortunately a common occurrence in the industry [4]. Bessey et al. [2, p. 73] noted how upgraded systems leading to more bugs being identified can be bad for managers, as this can damage their bonuses. For performance-related 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), which would allow greater protection against memory corruption.

2.3 Benefits of using an intermediate representation

A more subtle theme seen across the three papers involves potential benefits of using intermediate representations in analysis. Klein et al. [1, 2.2, 5.2] used a Haskell prototype as an executable model prior to optimisation in C, which saved significant cost in verification and allowed them to take advantages of existing toolchains and experience. When analysing C# and Java bytecode compiled from the source, there was no need to navigate compiler deviations, therefore Bessey et al. [2, p. 72] had a much easier time. Techniques providing the highest level of memory corruption protection (SoftBound+CETS) represented new objects with unique identifiers in processing to improve memory performance [3, VI-C].

3 Ideas in current context

In order to address the limitations of seL4's uniprocessor model, a follow up research by Tessin [5] presented the *clustered multikernel* approach for constructing verifiable multiprocessor kernels. In addition, Tessin developed a "lifting framework" to add multiprocessor support to verified uniprocessor microkernels with only minor modifications. This approach could save a large amount of effort required for developing a verifiable multiprocessor kernel from scratch.

Intermediate representations have been further utilised to improve program analysis when searching for errors and security violations, as demonstrated by the Static Analysis Intermediate Language (SAIL) [6]. The two-level intermediary design of SAIL preserve syntactical error reporting capabilities, while allowing semantical analysis to be performed for error finding. 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 failed to strictly observe language standards, creating false-positive error reports. An alternative error-finding approach has since emerged, combining the classical Bounded Model Checking (BMC) approach with LLVM intermediate representation to offset complexities of modern language features in C/C++, which previously made applying BMC difficult [8]. This approach is also able to find bugs introduced by compilers – addressing another complaint by Bessey et al.

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 were also 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] when designing their scalable multicore kernel, albeit without formal verification. In order to address the security problems caused by compiler errors when compiling verified C code such as seL4, Yang et al. [11] developed a method for detecting compiler bugs through randomised test-case generation.

Considering the difficulties faced by Bessey et al. [2] in performing static analysis on unseen codebases, Chipounov et al. [12] proposed an alternative method of performing multi-path analysis directly on binaries. While their approach solved problems in accessing source code, it lacks error reporting capabilities due to the compiled nature of binaries. Johnson et al. [13] further investigated reasons behind 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 for unanalysed reflective calls.

Szekeres participated in a more recent work [15], which introduced new design concepts of code-pointer integrity and code-pointer separation, which supplemented 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, as full memory safety protection is very computationally-expensive as measured by Szekeres et al. [3].

(1189 words according to texcount.)

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.