Automatically Proving Microkernel Security

(doctoral work)

Olivier Nicole
Département d'informatique, ENS, CNRS, PSL University, Paris, France
Software Safety and Security Laboratory, CEA List, Paris-Saclay, France
olivier.nicole@ens.psl.eu

***Abstract*—Operating system kernels are the security keystone of most computer systems. While prior OS kernel formal verifications were performed either on source code or crafted kernels, with manual or semi-automated methods requiring significant human efforts in annotations or proofs, we aim at computing such kernel security proofs automatically, starting from the executable code of an existing microkernel without modification, thus formally verifying security properties with a high confidence for a low cost. First results include a method to verify *absence of privilege escalation* in an industrial embedded microkernel, and we are currently working towards stronger properties, such as functional properties or separation between tasks.**

I. Introduction

Context. The security of many computer systems builds upon that of its operating system kernel. We define a kernel as *a computer program that prevents untrusted code from performing arbitrary actions*, in particular performing arbitrary hardware and memory accesses[1]. As vulnerabilities can be of the highest severity, formally verifying that kernels cannot be exploited or cannot crash is of the uttermost importance. Then, one may want to prove functional properties, i.e. properties that state the compliance of the kernel with some specification.

Scope. Besides large well-known monolithic kernels (e.g., Linux, Windows, *BSD) which are currently out of reach of formal verification, there is a rich ecosystem of small-size kernels found in many industrial applications, such as security or safety-critical applications, embedded or IoT systems.

We focus on such small-size kernels. To have a practical impact on these systems, a formal verification must be:

- **Non-invasive**: the verification should be applicable to the kernel as it is. Formal verification methods that require heavy annotations or rewrite in a new language are expensive and require developers with a rare combination of expertises (OS development and formal methods);
- **Automated**: the cost and effort necessary to perform a formal verification should be minimized.
- **Close to the running system**: Verification should be performed on the machine code in order to remove the build chain (compiler, linker, compilation options, etc.) from the trust base. This is especially true for kernels, as they contain many error-prone low-level interactions with the hardware which are not described by the source-level semantics.

Despite significant advances in the last decades [1]–[11], existing kernel verification methods do not address these issues. In most cases, verification is applied to microkernels developed or rewritten for the purpose of formal verification (except [6]), and is performed only on source [5], [6] or assembly [4], [8], [10], [11], using highly expensive manual [2], [3], [5], [6] or semi-automated [4], [7]–[9] methods. For example, the functional verification of the SeL4 microkernel [3] required 200,000 lines of annotations and still left parts of the code unchecked (boot, assembly).

Challenges. Besides the well-documented difficulty of analyzing machine code, a major challenge is that most kernels are *parameterized* systems designed to run an *arbitrary* number of tasks. This is also true for microkernels in embedded systems: even if the number of tasks, size of scheduling tables and communication buffers often do not vary during execution, they depend on the application using the kernel. A flat representation of memory (enumerating all memory cells) is no longer sufficient in such a setting, and we need more complex representations able to precisely *summarize* memory.

Contributions so far. We propose BINSEC/CODEX, a novel static analysis for proving *absence of privilege escalation* in microkernels from their executable. Our contributions include:

- A *formal model* suitable for defining privilege escalation attacks on parameterized kernel code and allowing to reduce the proof of absence of privilege escalation to a standard program analysis problem (finding *non-trivial state invariants*), hence reusing the standard program analysis machinery.
- A *3-step methodology* for inferring such state invariants, featuring 1. automated extraction of most of the analysis configuration; 2. a parameterized fully-automated binary-level static analysis by abstract interpretation, inferring an *invariant of the kernel under a precondition on the user tasks*, and 3. a fully-automated method to check that the user tasks satisfy the inferred precondition;
- A *novel weak shape abstract domain* able to verify the preservation of memory properties in parameterized ker-

PhD supervised by Xavier Rival (ENS–CNRS–PSL) and Matthieu Lemerre (CEA List). Special thanks go to Sébastien Bardin (CEA List) for useful comments and discussion. This PhD (ending in Oct. 2021) is supported by a CEA grant.

[1]This definition includes kernels and hypervisors with software-based or hardware-based isolation, but excludes the hardware abstraction layers of operating systems without memory protection.

nels, which is *efficient*, *easily configurable* (based on the memory layout of C types), and *suitable to machine code verification* (e.g., addressing indexing of data structures using numerical offsets);
- A case study (Section II) where we apply our method to a *concurrent industrial microkernel*. Our analysis is able to *identify a bug* in a beta version of the kernel, and to *verify the absence of privilege escalation* in a later version, in less than 450 seconds and with only 58 lines of manual annotations — several order of magnitudes less than prior verification efforts (Table II).

This work is the first OS verification effort to specifically address *absence of privilege escalation*. It is also the first to perform formal verification on *an existing* operating system kernel *without any modification*, on *machine code*, and the first to do so using a *fully-automated technique* able to handle *parameterization*.

Limitations. Like any sound static analyzer, BINSEC/CODEX may be too imprecise on some code patterns, emitting *false alarms*. Also, currently, our analysis cannot handle dynamic task spawing nor dynamic modification of memory repartition, as well as self-modification or code generation in the kernel. Still, many microkernels and hypervisors fall in our scope.

II. VERIFYING ABSENCE OF PRIVILEGE ESCALATION ON AN INDUSTRIAL MICROKERNEL

Implementation. Our static analysis technique has been implemented in BINSEC/CODEX, a plugin of the BINSEC [12] framework for binary-level semantic analysis. We reuse the intermediate-representation lifting and executable file parsing of the platform, but implemented a whole static analysis on top of it. Development is done in OCaml, the plugin counts around 41 kloc.

Case study. We applied our method to a microkernel, part of an industrial solution for implementing security- and safety-critical real-time applications, used in industrial automation, automotive, aerospace and defense industries. The kernel is being developed by AnonymFirm[2] — an SME whose engineers are not formal method experts — using standard compiler toolchains. The system is parameterized: the kernel and the user tasks are compiled separately and both are loaded at runtime by the bootloader.

We have analyzed a port of the kernel to a 4-cores ARM processor. It relies on the ARM MMU for memory protection. The executable file contains about 4,500 instructions. We have analyzed two versions:

- BETA is a preliminary version, where we found a bug;
- V1 is a more polished version where AnonymFirm fixed the bug and removed some debug code.

We ran our analysis on a laptop with an Intel Xeon E3-1505M 3 GHz CPU with 32 GB RAM.

[2]For confidentiality reasons, we are not free to disclose the company's name.

TABLE I
MAIN VERIFICATION RESULTS

		Generic configuration		*Specific* configuration	
# shape annotations	generated	1057			
	manual	57 (5.11%)		58 (5.20%)	
Kernel version		BETA	V1	BETA	V1
invariant computation	status	✓	✓	✓	✓
	time (s)	647	417	599	406
# alarms in runtime		1 **true error** 2 false alarms	1 false alarm	1 **true error** 1 false alarm	**0** ✓
user tasks checking	status	✓	✓	✓	✓
	time (s)	32	29	31	30
Proves APE?		N/A	∼	N/A	✓

Protocol. We consider both kernel versions and two configurations for annotations:

- ***Generic*** contains types and parameter invariants which must hold for all legitimate user tasks;
- ***Specific*** further assumes that stacks of all user tasks have the same size. This is the default for applications on this kernel, and it holds on our case study.

Results. The main results are given in Table I. The ***generic*** configuration requires only 57 lines of manual annotations, in addition to 1057 lines that were automatically generated (i.e. 5 % of manual annotations). When analyzing the **BETA version** with this configuration, only *3 alarms* are raised in the runtime:

- One is a **true vulnerability** (an *off-by-one error* in manually written assembly code that sanitizes the system call number), potentially allowing a malicious user task to (at least) crash the system;
- One is a false alarm caused by *debugging code* temporarily violating the shape constraints (by writing `0xdeadbeef` in a pointer, before correcting it);
- The last one is a false alarm caused by an imprecision in our analyzer when user stacks can have different sizes.

When analyzing the **V1 version**, the first two alarms disappear (no new alarm is added). Analyzing the kernel with the ***specific*** configuration makes the last alarm disappear. In all cases task checking succeeds.

Analyzing the V1 *kernel with the* specific *configuration allows to reach 0 alarms, meaning that we have* a fully-verified invariant *and a proof of absence of privilege escalation.*

Computation time is *always very low*: less than 11 minutes for the analyzer and 35 seconds for the checker.

Our analysis also computes other useful properties (absence of runtime error, respect of call/return conventions) and relevant information that can be verified against the expectations of kernel developers (control-flow graph, read and written addresses, memory shared between processors, etc.).

III. CONCLUSION

Table II presents a comprehensive overview of prior OS verification efforts.

TABLE II
COMPARISON OF KERNEL VERIFICATION EFFORTS

VERIFIED KERNEL	VERIFIED PROPERTY		VERIFICATION TECHNIQUE					CASE STUDY			
	Verified property	Implies APE?	Degree of automation	Verif. Level	Parame-terized	Multi-core	Infers invariants	Manual Annotations (LoC)	Unproved code (LoC)	Non-invasive	Analysis time (s)
This work	Absence of priv. escalation	✓	**Fully automated**	**Machine**	✓	✓	✓	58✓	0 ✓	✓	**406**
XMHF [13]	Memory integrity	✗[b]	Semi automated	Source	✗[h]	✓	✗	N/A	422 (C) + 388 (asm)	✗	**76**
üXMHF [8]	Security properties	✓	Semi automated	Source + assembly	✗[h]	✗	✗	5,544	0 ✓	✗	3,739
Verve Nucleus [4]	Type Safety	✓	Semi automated	Assembly	✓	✗	✗	4,309	0 ✓	✗	**272**
Prosper [7]	Compliance with specification	✓	Semi automated	**Machine**	✗[i]	✗	✗	6,400 [c]	0 ✓	✗	$\leq$ 28,800
Baby hyper-visor [10]	Compliance with specification	✓	Semi automated	Source+ assembly	✓	✗	✗	8,200 tokens	0 ✓	✗	4,571
Komodo [9]	Compliance with specification	✓	Semi automated	Assembly	✓	✗	✗	18,655	0 ✓	✗	14,400
UCLA Secure Unix [1]	Compliance with specification	✓	Manual	Source	✓	✗	✗	N/A	80%	✗	N/A
Kit [2]	Compliance with specification	✓	Manual	**Machine**	✗	✗	✗	1,020 definitions + 3561 lemmas	0 ✓	✗	N/A
μC/OS-II [6]	Compliance with specification	✓[a]	Manual	Source	✓	✗[f]	✗	34,887 [d]	37%	✓	57,600[e]
SeL4 [3]	Compliance with specification	✓[a]	Manual	Source[g]	✓	✗	✗	200,000	1,200 (C, boot) + 500 (asm)	✗	N/A
CertiKOS [14]	Compliance with specification	✓	Manual	Source[g]+ assembly	✓	✓	✗	100,000	0 ✓	✗	N/A

[a] Assuming that the proof is completed to cover all the code. [b] Control flow integrity is assumed. [c] Generated from a 21,000 lines of HOL4 manual proof. [d] Plus 181,054 LoC of specification and support libraries. [e] The reported compilation time includes the support libraries. [f] The verification is concurrent because of in-kernel preemptions. [g] The translation to assembly is also verified. [h] The hypervisor supports a single guest [i] The hypervisor supports two guests

Our experiments up to now show that verifying absence of privilege escalation of an industrial microkernel using only fully-automated methods is feasible (albeit with a very slight amount of manual configuration).

We are now working to extend our results in two directions:

- Proving memory separation between tasks, not only between kernel and tasks.
- Verifying more specific properties, such as "this register is not modified by this system call".

REFERENCES

[1] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and verification of the ucla unix security kernel," *Commun. ACM*, vol. 23, pp. 118–131, feb 1980.

[2] W. Bevier, "Kit: A study in operating system verification," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1382–1396, 11 1989.

[3] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, pp. 2:1–2:70, feb 2014.

[4] J. Yang and C. Hawblitzel, "Safe to the last instruction: automated verification of a type-safe operating system," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 99–110, 2010.

[5] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, 2015.

[6] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li, "A practical verification framework for preemptive os kernels," in *International Conference on Computer Aided Verification*, Springer, 2016.

[7] M. Dam, R. Guanciale, and H. Nemati, "Machine code verification of a tiny ARM hypervisor," in *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, TrustED '13, ACM, 2013.

[8] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta, "überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor," in *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, 2016.

[9] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, ACM, 2017.

[10] W. Paul, S. Schmaltz, and A. Shadrin, "Completing the automated verification of a small hypervisor – assembler code verification," in *Software Engineering and Formal Methods*, Springer, 2012.

[11] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified os kernel," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, ACM, 2013.

[12] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion, "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis," in *SANER*, IEEE Computer Society, 2016.

[13] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *2013 IEEE Symposium on Security and Privacy*, May 2013.

[14] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "Certikos: An extensible architecture for building certified concurrent OS kernels," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association, 2016.