A Survey of Formal Verification Approaches for Practical Systems

Qiao Zhang, Danyang Zhuo, James Wilcox
University of Washington

1 Introduction

The development of any large scale software systems often involves the discovery and elimination of an enormous amount of bugs. Linux kernel bug tracker currently tracks 2830 known bugs as of April 2015, with many bugs that are likely still unknown [3]. A bug in Google LevelDB prevents users from storing block chains and participating in the Bitcoin network [1]. The absence of checksum in an internal state caused Amazon S3 to become unavailable for hours in 2008 [2]. To ensure the reliability of systems, best practices in industry often rely on frequent peer code reviews, extensive test suites and occasionally static and dynamic analysis. However, despite significant effort spent in eliminating programming errors, practical systems deployed today still suffer from frequent errors, sometimes leading to catastrophic consequences [2].

Formal verification is the only known way to ensure that a system is completely free of bugs and that the system is behaving correctly according to a set of high-level specifications. However, to formulate a precise program specification and to provide a formal proof that an implementation meets the specification is often a significant undertaking. The conventional wisdom in the systems community is that the costs of producing a formal proof far outweigh its benefits.

The reality is that the technology in program verification is becoming mature enough that many recent projects have formally verified systems of scale, hitherto thought to be impractical. The variety of existing approaches, however, are diverse in their goals, tools and end-to-end guarantees.

We provide a systematic survey of existing formal verification approaches while critically examining each approach to answer our key question: how can practical systems programming effectively use formal verification to ensure reliability?

Specifically, we give a taxonomy of existing formal verification approaches, classifying each based on the following aspects:

1. What system components are verified?
2. What are the verification goals?
3. What verification tools are used?
4. How much proof effort is required?
5. What is the Trust Computing Base (TCB) of the verified system?

Based on our survey, we identify a few challenges with existing formal verification approach for practical systems building. Moreover, we identify some system components that would be interesting to formally verify that have not yet been attempted and set down the challenges that we would face using the state of the art approaches.

2 Formally Verified Systems

This section overviews a few projects that use formal verification to build reliable systems, ranging from OS components, to full OS kernels, to distributed systems.

2.1 Operating Systems

2.1.1 Jitk

Modern operating systems run multiple interpreters in the kernel, which enable user-space applications to add new functionalities or specialize system policies. It is very challenging to ensure that both the in-kernel interpreter and the user-supplied code are bug-free. In fact, existing in-kernel interpreters suffer from a wide range of bugs such as jump target off by one, incorrect division-by-zero check, buffer overflow and incorrect translation of high-level policies to low-level bytecode.

Jitk is a new infrastructure for building verified in-kernel interpreters that guarantee functional correctness through high-level policy language to native code as well as guarantee safety in executing native code in-kernel [8]. Jitk translates policy rules to BPF instructions that are transmitted across the user-kernel boundary in bytecode format. Jitk decodes the BPF bytecode and translates the BPF instructions to Cminor. CompCert compiles Cminor instructions to native assembly, which gets translated to native binary code using a conventional assembler. Jitk uses Coq to prove the semantic preservation from policy language to BPF language, the correctness of BPF encoder/decoder, and the semantic preservation from BPF language to Cminor. Jitk trusts that CompCert correctly compiles Cminor to assembly and that a conventional assembler correctly translates the assembly to binary code.

System	Type	Proof Goals	Proof Tools	Proof Effort	Implementation	TCB
Jitk	in-kernel interpreter	implementation functional correctness and safety	Coq, CompCert	2300 lines of Coq proof	JIT code extracted from Coq to OCaml	Coq, CompCert, Assembler, OCaml compiler/runtime
ExpressOS	mobile OS	security invariants for mobile apps	code contracts, Dafny	2.8% annotation ratio	C# and Dafny	hardware, L4 microkernel, C# and Dafny compilers/runtime
Verve	OS kernel	Memory- and Type-safety	Boogie, safe C#, and TAL	2-3× annotation ratio	C# and Boogie compilers	Compilers, the linker
seL4	OS microkernel	functional correctness	Isabelle/HOL, Haskell	33k Isabelle LOC, 14k Haskell/C LOC, 20 py	high performance C	spec, GCC, assembler, hardware
VCC	20% of Hyper-V	functional correctness	Boogie, Spec#	13K lines of annotation	C	Boogie, translation from C to Boogie
Ironclad Apps	remote services	Secure remote equivalences	Dafny, Boogie and Z3	implementation to spec ratio is 2×, proof annotation is 4.8×	Dafny compiled to assembly	spec, hardware, Verve, assembler, linker
Quark	Browser kernel	security properties	Coq, Ynot	8KLOC	Extract to Ocaml	Coq, OCaml, sandbox
Reflex	Reactive systems	temporal logic assertions	Coq, Ynot, custom automation	13KLOC (no manual proofs for systems!)	DSL compiler	Coq, DSL frontend, Ynot
Verdi	distributed systems	safety properties	Coq	30KLOC	Extracted to OCaml, linked with shim	Coq, OCaml, shim, fault model
Compcert	C compiler	semantic equivalence	Coq	150KLOC	Extracted to OCaml	Coq, OCaml, reader, printer
Bedrock	low-level code	functional correctness	Coq (embedded DSL)	36KLOC	x86	x86 semantics

Table 1: Taxonomy of Formally Verified Systems

Safety properties such as termination and absence of kernel stack overflow are proven using Coq too. At a cost of 2300 lines of Coq and a total of 3510 lines of code for the Jitk/BPF prototype, Jitk is shown to prevent commonly known bugs in in-kernel interpreters.

2.1.2 *ExpressOS*

ExpressOS is a new OS architecture that provides formally verified security invariants to mobile applications [7]. In contrast to prior OS verification effort that aimed at full functional correctness, ExpressOS only focuses on guaranteeing security invariants covering secure storage, memory isolation, user interface isolation and secure IPC. In order to further simplify verification effort, ExpressOS reduces the amount of code in the kernel by pushing functionality into microkernel services as well as leveraging programming language type safety to isolate control and data flows within the kernel. ExpressOS kernel is implemented in type-safe C# and Dafny. To guarantee the security invariants, the implementation makes extensive use of code contracts and Dafny annotations, e.g. pre- and postconditions and object invariants. Complex specifications that are not always expressible in terms of pre- and post condition assertions on local-scope program state necessitate the use of ghost variables to aid verification. While code contracts are verified using abstract interpretation techniques and require low annotation overhead, they cannot reason about complex properties and therefore are used for verifying simpler security invariants. The more expressive Dafny annotations which use SMT solver and require heavy annotation burden and deep expertise in formal methods are used to reason about more complex properties. The combination of code contracts and Dafny annotations allow high productivity and a low annotation/code ratio of 2.8%.

ExpressOS trusts the hardware, the L4 microkernel that interfaces the hardware, the compilers and the language runtime. All system services are however not part of the TCB since the security invariants for the applications are still maintained even if the system services are compromised.

2.1.3 *Verve*

Verve is an operating system that is verified to have type safety and memory safety [10]. The aim is to substantially reduce the human effort of proving operating system. The traditional OS is separated into a Nucleus and a kernel.

The Nucleus consists of all the abstractions of the underlying hardware and memory. It is written completely in assembly language. Each function is annotated with preconditions, postconditions and loop invariants. The correctness is verified by Boogie. Boogie is a Hoare style program verifier. The kernel implements all the fully-fledged OS services. It is written in safe C# and compiled to Typed assembly language(TAL) which is then checked with existing TAL checker.

Verve does not trust high level language compiler or any unverified library code. It only has to trusts Boogie, TAL checker, the assembler and the linker. In contrast to a script-to-code ratio of 20 of sel4, Verve only requires 2-3 annotations per executable statement.

2.1.4 *seL4*

seL4 [6] is a fully verified L4 microkernel. Its C implementation is verified to satisfy the specification. The abstract specification of what the kernel does is precisely modeled in Isabelle/HOL, for example, argument formats, encodings and C integer type width. A Haskell prototype of the kernel (also called an executable specification) is implemented based on the abstract specification to determine the low level details on data structures and algorithms. Finally, a high performance C implementation is manually translated from the Haskell prototype.

The functional correctness of the kernel is proved by refinement in two stages. The proof demonstrates first correspondence between the abstract specification and the executable specification and then correspondence between the executable specification and the C implementation. Since correspondence is transitive, the proof shows that the C implementation satisfies the abstract specification precisely.

seL4 is the first-ever fully verified general-purpose kernel and the verified implementation is in C. The proof took 33k Isabelle/HOL LOC, 14k Haskell/C LOC and a total of 20 person-year.

2.1.5 *VCC*

The goal of VCC is to develop a set of tools that can verify commercial system software and then use those tools to verify functional correctness of Microsoft Hyper-V [4]. Hyper-V is an existing commercial hypervisor written in C and it was built without any formal verification in mind.

VCC requires developers to annotate C functions with pre- and post-conditions, assertions and other invariants. VCC does static analysis to generate a corresponding Boogie program and Boogie leverages Z3 to check if each assertion in the functions can be soundly proved. Tools, like VCC model viewer, translate counterexamples in Z3 back to a program state sequence that violates the assertions.

VCC successfully proves 20% of the Hyper-V code base with 13500 lines of annotation.

2.2 Reactive and Distributed Systems

2.2.1 *Ironclad Apps*

Ironclad apps [5] are a set of secure services, for example, notary or differential-privacy database service, that provide remote clients with the guarantee that the remote

code executed verifiably conform with the app's high level abstract state machine and that the app verifiably uses secure hardware. Ironclad apps verify the end-to-end security property of the full software stack, including not only the app, but also the OS, libraries and drivers. Moreover, the proofs cover the assembly code compiled from high level Dafny language, so Ironclad does not include the compiler or language runtime as part of its TCB.

Ironclad achieves rapid development by using state-of-the-art automated software verification tools such as Dafny, Boogie and Z3. Ironclad uses a set of techniques such as incremental verification, opaque function and also adopts software engineering disciplines such as idiomatic specification to allow scalable and stable verification through automated tools.

Ironclad has 2:1 implementation to spec ratio and 4.8:1 proof annotation to implementation ratio. The four Ironclad apps consist of 3,546 lines of spec and 7K lines of implementation in total. Ironclad took about 3 person-years to verify.

2.2.2 Verdi

Verdi is a framework for formally verifying distributed system implementations [9]. Because distributed systems run in a diverse range of environments, Verdi supports verifying systems in various fault models. Verdi serparates verifying application logic from fault tolerance mechanisms using verified system transformers. Verdi contains system transforms which handle several types of common faults with proofs that they preserve application logic in the new network semantics.

2.2.3 Quark

Quark is a verified browser kernel that enforces security properties such as tab noninterference. The key insight of Quark is that these properties can be enforced by writing and verifying a relatively small amount of code (on the order of 1000 lines). The remaining code is run in a sandbox that is controlled by the kernel.

2.2.4 Reflex

Reflex generalizes the techniques of Quark to arbitrary reactive systems. Reflex includes custom automation tactics to prove temporal logic assertions about systems expressed in the Reflex DSL. By specializing the type of system allowed and restricting the properties that can be expressed, Reflex enables the construction of a verified SSH server, browser kernel, and web server.

2.3 Compilers

2.3.1 Compcert

Compcert is a verified C compiler. Compcert includes formal semantics for C, three backends (x86, ARM, and PowerPC), and over 10 intermediate languages used as part of the compilation process. The externally visible behavior of a program is defined to be the sequence of system calls that a program makes. Compcert is verified to preserve this behavior. That is, the resulting assembly program is guaranteed to make the same system calls in the same order with the same argument. It is thus indistinguishable from the input program.

2.3.2 VST

Verified Software Toolchain (VST) verifies the assertions claimed at the top of the toolchain still hold in the machine-language program. VST consists of three parts: A static analyzer that checks the assertions in the source-language program, a proved compiler that transform the source-language program to machine-language program and a runtime to support external function calls from the machine-language program.

VST specify observable behaviors of source- and target-language programs and proves the corresponding target-language exhibit the same observable behavior as the source-language. The benefit of this approach is that the checker does not need to reason about the internal behaviors on the concurrent states. VST chooses C minor as the source language. TCB of VST is the transition from source-language to target-language and Coq.

2.3.3 Bedrock

Bedrock is a framework for writing and verifying low level code in Coq. Bedrock includes custom automation for proving separation logic formulas over x86 programs. Bedrock has been used to verify a multithreaded web-server implementation.

3 Challenges

Recent verification projects aim to verify the implementations rather than just the high level specifications of systems, e.g. VCC and seL4. However, none of those projects successfully verified an existing system. Microsoft HyperV is a hypervisor written without formal verification in mind. VCC only manages to verify 20% of the implementation, rather than the full system. Most other projects start from scratch and write a full system that are more amenable to verification. Verdi and Jitk allows an implementation to be extracted in a high level language such as OCaml or Haskell. However, for performance critical systems, the eventual implementation in OCaml is unlikely to be deployable which defeats the purpose of verifying the implementation rather than just the specifications. seL4 is a notable exception to this, but it requires manual translation from a Haskell prototype to C and require two refinement proofs due to the dual implementation. The proof effort was enormous, so it is unclear if the approach of dual implementation is optimal.

Some projects make extensive use of automated theorem prover such as Dafny in their verification, e.g. Ironclad, ExpressOS. Such tools make the verification process less manual, but at the expense of an increased TCB. Moreover, Hoare proof style is not applicable to all systems. The alternative approach is to use Coq which has an extremely small TCB. The drawback is that proving theorems in Coq require an extensive and intricate knowledge of proof tactics. While having a suite of tactics as a library alleviate the problem, proving systems in Coq is still difficult.

4 Research Problems

4.1 Verified Bitcoin Scripting Engine

Recently, Bitcoin emerges as the most important cryptographic currency. Bitcoin uses a Forth-like scripting language to enable flexible financial transactions. Bugs in Bitcoin scripting engine can result in substantial monetary loss to participating parties. The scripting engine is implemented in C as an open source project on GitHub. Similar to Jitk, we can implement a Bitcoin scripting engine in Coq and extract an OCaml/Haskell implementation with functional correctness properties verified.

4.2 An Incrementally Verifiable System

Using an automated theorem prover allows us to prove properties such as memory safety or type safety with relatively small amount of proof effort compared to using a proof assistant, as shown in the examples of ExpressOS and Verve. The low annotation ratio makes this approach appealing for building many practical systems. One can imagine that the time spent in annotation is well-rewarded by the time saving in less debugging and less testing.

While guaranteeing memory and type safety alone may be sufficient for some applications, there are applications that would require additional assurance such as functional correctness. Automated theorem prover offers less control than a proof assistant, and thus projects that benefit from the ease of proving memory safety and type safety would have to pay a steep price once they need to prove additional properties such as functional correctness. The alternative of doing all proofs in a proof assistant such as Coq from the beginning may contradict with the software engineering goals of rapid development and frequent iterations for practical system building.

We imagine that there could be a better verification toolchain that offers the ease of proving memory/type safety properties as well as the flexibility of incrementally proving more complex properties when needed.

5 Conclusion

In this paper, we provided a systematic survey of existing formal verification approaches and created a taxonomy of those approaches based on various aspects. We pointed some challenges with existing approaches for building practical systems and hopefully shed some light on what research problems are interesting to work on in the future.

References

[1] Claiming bitcoin's bug bounty. http://hackingdistributed.com/2013/11/27/bitcoin-leveldb/.

[2] Amazon S3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html.

[3] Linux kernel bug tracker. https://bugzilla.kernel.org/.

[4] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 23–42, Munich, Germany, Aug. 2009.

[5] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, Oct. 2014.

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, Oct. 2009.

[7] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in ExpressOS. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 293–304, Houston, TX, Mar. 2013.

[8] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–47, Broomfield, CO, Oct. 2014.

[9] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 2015*

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, OR, June 2015.

[10] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 99–110, Toronto, Canada, June 2010.