Lessons from Formally Verified Deployed Software Systems

LI HUANG, Constructor Institute of Technology, Switzerland
SOPHIE EBERSOLD, IRIT, University of Toulouse, France
ALEXANDER KOGTENKOV, Previously at Constructor Institute of Technology, Switzerland
BERTRAND MEYER, Constructor Institute of Technology, Switzerland
YINLING LIU, CRAN CNRS UMR 7039, University of Lorraine, France

The technology of formal software verification has made spectacular advances, but how much does it actually benefit the development of practical software? Considerable disagreement remains about the practicality of building systems with mechanically-checked proofs of correctness. Is this prospect confined to a few expensive, life-critical projects, or can the idea be applied to a wide segment of the software industry? To help answer this question, the present survey examines a range of projects, in various application areas, that have produced formally verified systems and deployed them for actual use. It considers the technologies used, the form of verification applied, the results obtained, and the lessons that the software industry should draw regarding its ability to benefit from formal verification techniques and tools.

CCS Concepts: • **General and reference → Surveys and overviews**; **Verification**; • **Software and its engineering → Correctness**; *Formal methods*; **Formal software verification**.

ACM Reference Format:
Li Huang, Sophie Ebersold, Alexander Kogtenkov, Bertrand Meyer, and Yinling Liu. 2024. Lessons from Formally Verified Deployed Software Systems . Preprint, 2023-2025.

1 INTRODUCTION

The ever more central role that software plays in all processes of the modern world brings to the forefront the critical question of program correctness. How do we know that software systems perform as expected? *Formal verification* is the task of proving that a program fulfills its specification. It is a long-established research area of software engineering, but disagreement persists on its relevance to mainstream system development. Many practitioners have not even heard of formal methods, and those who have often dismissed them as too hard to apply to mainstream software projects. Against this view, proponents of formal verification argue that the technology has now reached a high level of maturity and applicability. Which of these two views is correct? In other words, how realistic is the prospect of applying formal methods to production projects? To help answer this question, it is important to have an objective basis: an assessment of existing attempts to apply formal methods in practice. The present survey provides such an assessment, by reviewing

Authors' addresses: Li Huang, li.huang@constructor.org, Constructor Institute of Technology, Switzerland, Schaffhausen; Sophie Ebersold, Sophie.Ebersold@irit.fr, IRIT, University of Toulouse, France, Toulouse; Alexander Kogtenkov, kwaxer@mail.ru, Previously at Constructor Institute of Technology, Switzerland, Schaffhausen; Bertrand Meyer, Bertrand.Meyer@inf.ethz.ch, Constructor Institute of Technology, Switzerland, Schaffhausen; Yinling Liu, Yinling.Liu@univ-lorraine.fr, CRAN CNRS UMR 7039, University of Lorraine, France, Epinal.

software systems fulfilling two properties: they have actually been deployed; and they were the subject, during their development, of formal verification.

Two or three decades ago, one could legitimately phrase the underlying question simply as: "*Can formal verification be applied in industry?*". In that form, it is no longer open: a number of well-publicized industrial projects have used formal verification, even if initially for relatively small programs in mission-critical and particularly life-critical areas such as transportation and defense, where the consequences of incorrectness in programs are so great as to justify any difficulties and extra costs that formal verification might imply. The contemporary version of the question does not ask any more about feasibility (which has been established) but about practical aspects, such as: *How large a system* can formal verification handle? *What special qualifications* or training does it require for the development team? *What extra cost*s, if any, does it imply?

While it is beyond the scope of this survey to provide definitive answers to these and other questions on the practicality of formal verification, it introduces a factual basis for discussing them. It analyzes a number of deployed, formally verified systems according to a set of criteria listed in section 3.2. The lessons learned from this analysis appear in section 7.
The present article, which does go into technical details, resulted from studying a significant set of formally verified and deployed software systems of widely different kinds, extending across a variety of implementation languages and verification techniques. The projects were selected using a combination of literature review and responses to a questionnaire widely circulated by the authors. This article helps answer the following questions:

- In what areas of the industry have formally verified IT systems been deployed?
- What are the properties of the formally verified systems' projects in terms of required initial developer expertise, learning effort and effect on the software process?
- What approaches (programming languages, mathematical basis, verification techniques, verification tools, verification schemes) have been applied to verify deployed systems?
- What are the potential of and obstacles to generalizing the results to the software industry as a whole? Are there specific kinds of systems that do not lend themselves to formal verification?

The rest of the article is structured as follows:

- Section 2 relates the findings to those of previous surveys on neighboring topics.
- Section 3 presents the selection criteria for the analyzed systems, and the analysis criteria.
- Sections 4 and 5 provide a short tutorial on formal verification, focusing on the methods actually used in the projects under study.
- Section 6 is the core of the article; it describes and assesses the individual projects and their use of formal verification, according to the criteria of section 3.2.
- Section 7 draws the general lessons of the analysis and examines its limitations as well as its significance for the generalization of formal verification in industry.

2 PREVIOUS SURVEYS

Table 1 provides a point-by-point comparison with earlier surveys covering part of the scope of the present one. These previous articles and some of their main results are the following:

- Surveys of systems verified with a specific approach (Event-B in [2], SPARK in [14]) conclude that regulators seem skeptical on the use of strong static analysis and theorem proving in critical software, and that the primary determinant of the practical success of B-style formal methods is the quality and power of the supporting tools.
- Surveys of verified systems in a specific application area or of a specific kind (separation kernels [98]; distributed systems [29]) show that full formal verification has attracted large efforts in recent years. They conclude that verification effectively prevents protocol bugs.

Verification relies on assumptions that must be tested, typically through testing toolchains. These studies do not cover multi-core verification or automatic code generation.

- The survey of approaches based on the concept of proof assistant [73] has as its main finding that proof engineers succeed in continually increasing productivity by adapting the results of research on programming practices and development tools.
- A 2009 survey [90] presented a questionnaire-based summary of projects that had applied formal methods to some degree, not necessarily at the level of formally verified code.
- Recent surveys [93, 94] present (with less detail than here) a number of formally verified systems. They conclude that formal methods can be deployed in many industrial applications.
- The most recent survey [82] provides a broad scope of successful formal methods applications, including lithography manufacturing and cloud security in e-commerce.

Table 1. Comparison with major prior surveys

Systems selection methods: L — Literature review, Q — Questionnaires // Approaches: B — B-method, I — abstract Interpretation, M — Model checking, P — theorem Proving, R — Refinement, S — axiomatic Semantics, T — Testing // Project discussion aspects: S — Scope, C — Components, V — Verified properties, X — project conteXt, D — Decisions, T — Tool stack, Y — stYle, W — softWare characteristics, P — Project characteristics, L — Lessons

Ref.	Selection	Deployability	Areas covered	Formally verified	Approaches	Projects discussion aspects									
[2]	L	Yes	Transport	Yes	B	S	C	V	X		T		W	P	L
[14]	L	Yes	Navy, Aeronautics	Yes	P	S		V	X		T		W		L
[29]	L	No	Distributed systems	Yes	M, P	S		V			T				L
[73]	L	Yes	Industry	Yes	P	S			X				W		L
[90]	Q	Yes	National infrastructure, Computer microcode, Electronic finance, Security applications	No	M, P, R, T	S	C		X	D	T			P	L
[94]	L	Yes	Transport, Military, Defense	Yes	B, P	S			X		T			P	L
[93]	L	No	Mobile, Web servers	Yes	P	S					T		W	P	L
[98]	L	Yes	Aeronautics, Military, Medicine	No	B, M, P	S		V	X		T		W		L
[82]	L Q	Yes	Transport, Automotive, Aeronautics, OS, Network, Hardware, Manufacturing	Yes	B, M, P, T	S				D	T				
Current	L Q	Yes	Compiler, Library, OS, Aeronautics, Transport, Nuclear, Network, Database	Yes	B, I, M, P, S	S	C	V	X	D	T	Y	W	P	L

Some points of comparison between the present survey and these earlier articles are the following.

- It is, to our knowledge, the most comprehensive as to the number approaches reviewed and goes into more detail about the surveyed projects' features and results.
- It highlights two major ways of dealing with unverified parts.
- It analyzes the reasons behind the performance impact of verification.
- It highlights obstacles to widespread adoption beyond the difficulty of verification itself.

3 SELECTED SYSTEMS

The software systems under review must be both "formally verified" and "deployed". A system is *formally verified* if it has been mathematically proven to possess properties specified as part of its requirements. A system is *deployed* if it is either:

- Publicly available on the Web (in which case the authors of this article were able to use it).
- Not publicly available, but with strong evidence that it is used in production by several users.

Omitting the second category would have excluded commercial, proprietary systems, which account for some of the most significant applications of formal verification. The downside is that analysis of these systems has to rely on available documents often written by the authors of the respective systems, or interviews with these authors, rather than direct examination of the software.

3.1 Selection process

The selection of relevant systems used a combination of literature search and a questionnaire. While this article is not a Systematic Literature Review, it follows the recommended review protocol for SLRs [45]. A key reason for including a questionnaire (which remains available [71]) is that many formally verified systems from industry, relevant to the questions listed in Section 1, do not appear in the scholarly literature. The questionnaire was distributed to numerous communities and Internet channels starting in December 2020 to identify and document systems that have received little or no attention in the literature. It also asked for contacts to enable further interviews with authors. It yielded responses on 20 systems, of which 10 were deemed relevant. Examination of the documentation on these systems, suggestions from various sources, a list of companies that use formal verification methods in software engineering [18], and the general literature on formal verification led, by transitive closure on the references, to the identification of 65 potentially relevant systems from December 2020 to October 2021.

After application of the selection criteria described below, 32 systems remained (Table 2). 21 out of the 32 systems (highlighted in gray) are confirmed to be active — they are either still being actively used or their new features are under development. Systems that come from the replies of the questionnaire are underscored in Table 2. The systems were classified according to their application domain, resulting in 9 categories. For reasons of space, this article focuses on 11 of them, chosen for their representativeness (in terms of activeness, scale of project, level of verification and deployment), with the objective of representativeness of application fields and formal verification approaches. The other systems listed in the table are discussed in the extended version of this article [39]. A list of the relevant sites of the surveyed systems, including software release pages and software repositories, is accessible at https://sites.google.com/view/verified-and-deployed-software.

3.2 Criteria for the analysis

The description of the selected systems (in section 6) uses the following criteria:

- **Scope**: What system was verified, in what application domain and for what purpose?
- **Components**: What are the verified components and their roles?
- **Verified properties**: What properties of the system were verified?
- **Project context**: What is the motivation behind the project?
- **Decisions**: What key decisions were made to help the verification process?
- **Tool stack**: What tools were used for developing and verifying the software?
- **Style**: What is the underlying approach to specification?
- **Software characteristics**: What results did the verification effort produce?
- **Project characteristics**: What amount of resources was spent to verify the software?
- **Lessons**: What are the conclusions about the verification experience?

Blank entries express that the corresponding information was not available.

4 VERIFICATION: BASIC CONCEPTS

4.1 Definitions

The term "verification" covers techniques that ascertain the correctness of programs. It has both broad and narrow meanings:

- In practice, the software industry mostly uses "verification" to mean testing. In the programming research literature, the term is used instead to denote techniques for *proving* correctness mathematically; the meaning also retained in the present article.
- Software engineering textbooks often distinguish verification from *validation* (using "V & V" for their combination): verification is internal (checking consistency, "the systems does

Table 2. Deployed and Verified Systems Surveyed

Category	Name	Verified properties	Input PL	Output PL	Proof engine	Spec/ Imp	KLoC	Ef- fort (py)	References
Compiler	**CompCert**	semantics preservation	Coq	OCaml, C	Coq	1	135	6	[44, 54, 55]
	CakeML	semantics preservation	CakeML	CakeML	HOL4	–	100	–	[33, 49, 80]
	Vellvm	semantics preservation	Coq	OCaml, C	Coq	1	32	–	[95–97]
	Vericert	semantics preservation	Coq	OCaml	Coq	3.63	2.5	1.5	[36]
	Vermillion	functional correctness	Coq	OCaml	Coq	–	8	0.75	[50]
	Q*Cert	functional correctness	Coq	OCaml	Coq	–	–	–	[4]
Library	**Eiffel-Base2**	functional correctness	Eiffel	Eiffel	AutoProof, Boogie, Z3	–	9.37	6	[67]
	HACL*	security, functional correctness, memory safety	F*	C	Z3	3.3	31	< 1	[34, 99]
	DICE*	security, functional correctness, memory safety	F*	C	Z3, Meta-F*	4.7	29	-	[81]
	Signal*	security, functional correctness, memory safety	F*	Web-Assembly	Z3, ProVerif	-	4	-	[69]
	Amazon s2n	functional correctness, security	C	C	Coq, SAW	0.86	0.7	> 3	[16, 74]
OS	**seL4**	functional correctness, security	C	C	Isabelle, Z3, Sonolar	100	> 10	31.2	[30, 48]
	Proven-Core	security	C	C	Proven-Tools	–	–	3	[56]
	Verve	safety	Beat	C#, TAL	Boogie, Z3	3	7.55	0.75	[91]
	mCerti-KOS	security, functional correctness	Coq	ClightX, LAsm	Coq	6	3	1	[21, 31, 32]
	mC2	security, functional correctness	Coq	ClightX, LAsm	Coq	17	6.5	2	[32]
	Hyper-V	functional correctness, safety	C	Assembly, C	Boogie, Z3	5	105	1.5	[51]
	PikeOS	functional correctness, security	C	Assembly, C	Boogie, Z3	5	–	1.5	[9]
	Express-OS	security	C#, Dafny	C#	Z3	0.028	–	–	[59]
Aeronautics	SHOLIS	functional correctness, security, timing/memory constraints	SPARK	SPARK	Simplifier, Proof Checker	3.07	27	19	[46]
	Lockheed C130J	functional correctness	SPARK	SPARK	Simplifier, Proof Checker	–	350	–	[22]
	NATS iFACTS	absence of run-time exceptions, functional correctness, memory safety	SPARK	SPARK	Simplifier	0.3	250	> 50	[14, 15]
	EuroFighter Typhoon	functional correctness	Ada	Ada	Supertac, Proof-Power	–	35	1.5	[66, 79]
Transport	**Roissy Shuttle**	security	B, Ada	Ada	EDiTh B, Bertille	1.16	158	–	[6]
	Dutch Tunnel CS	safety and liveness	mCRL2	Java	mCRL2, VerCors	0.15	37	8	[65]
Nuclear Power	**Sizewell B**	functional correctness	B	PL/M-86, Assembly	MALPAS	–	150	250	[84]
Network	Ironclad Apps	functional correctness, security	Dafny	Dafny	Boogie, Z3	0.5	85	3	[28]
	Quark	security	Coq	OCaml	Coq, Ynot	0.2	976	0.83	[41, 42, 70]
	CoCon	security	Isabelle/ HOL	Scala	Isabelle/ HOL, BD-Security	0.67	15.2	0.25	[43, 68]
	CoSMed	security	Isabelle/ HOL	Scala	Isabelle/ HOL, BD-Security	1 (ker) 0.33 (app)	7.4	0.33	[8]
Database	**Ynot**	functional correctness	Coq	OCaml	Coq	1	3.03	8	[60]
Verification Tool	Flover	soundness with respect to standard semantics	Coq	HOL4	Coq, HOL4	10	3	2	[10, 27]

things right") and validation is external (checking the program against its specification, "the system does the right things"). In line with the usual sense of "formal verification" in the programming research community, this article uses "verification" to cover both parts.

- Verification is "dynamic" if it needs to execute the program, as in the case of testing. It is "static" if it works only from the program text. This article focuses on static techniques (which, since they do not perform any execution, require neither a compiler nor input data).
- Program proving is the most ambitious form of static verification, meant to ascertain that the program satisfies its specification. If the specification covers all relevant aspects of the program behavior, the proof will demonstrate "full functional correctness".
- Other forms of static analysis only analyze the program for the presence or absence of specific faults, such as deadlock or memory overflow.

The following characteristics apply to both static and dynamic forms of verification:

- Verification is not debugging. It may find faults ("bugs"), but correcting them is not its job.
- The words "succeed" and "fail" mean something else for verification tools and for programs. A verification tool *succeeds* if it either finds faults or ascertains that the program contains no faults of the specified kinds. The tool *fails* if it is unable to decide either way. (In contrast, a program succeeds if it completes its job according to its specification, and fails otherwise. Verification can succeed on a failing program — by identifying the fault — and conversely.)

4.2 Inherent theoretical and practical limitations

It is well known that a passing test, or any number of them, do not demonstrate that the program is correct. (A non-passing test does prove that the program is incorrect.) In principle, then, mathematical proofs are superior to tests. Limitations remain, however. First, a failed proof (as defined above) does not indicate that the program is incorrect, only indicates that the proof tool is unable to decide either way (correct or incorrect). This case is frequent for both theoretical and practical reasons:

- Theoretically: program correctness for any useful programming language is an undecidable problem, in the sense that it is not possible to produce a tool that will always prove or disprove the correctness of any program in finite time. (This property does not preclude the production of tools that will yield proofs for *some* programs.)
- Practically: a failed proof attempt is not the end of the story, just a step. ("Losing a battle, not the war".) It is in fact the common experience in the day-to-day practice of verification: try to verify; find that the tool either reveals a fault or produces no result; in either case, tweak the program, the specification or the proof to try to correct the problem; repeat.

Such an iterative scheme is reminiscent of the "run a test, fix a bug, repeat" of traditional non-formal development using tests. The difference is that, in static approaches, the basic iterative step involves analyzing the text of the program (rather than executing it). But in both cases, the process is iterative (as opposed to an ideal two-step process of writing the program then running a tool to prove its correctness). As a result, even though modern verification tools are described as permitting "automatic" or "mechanical" verification, their actual use still involves human effort. How much effort is an important practical factor to consider when assessing verification tools and techniques. This survey attempts, whenever information is available, to provide assessments of the effort that was involved in the verification of the reviewed systems.

4.3 The annotation issue

It is only meaningful to verify a program against specified properties. As noted, these properties can specify the entire relevant behavior of the program ("full correctness") or only some of its characteristics, for example that the computation will not produce an arithmetic overflow.

The first approach obviously yields more benefits, but it requires an extra *annotation* effort in addition to the program, since it needs a specification of the intended properties. The programmer or whoever is performing the verification must equip the program with such properties and often, in practice, intermediate “proof obligations”. The choice between verification approaches is in part a tradeoff between the amount of annotation effort and the extent of properties proved.

4.4 Maintining realistic expectations

Any verification success is relative: we verify a certain program element against a certain specification under certain hypotheses. The *specification* may be wrong (in the sense of not correctly expressing the desired behavior) or incomplete. The *hypotheses* may be violated; for example a program written in a high-level language, even if certified correct by a verifier for that language, can still fail if the compiler does not respect the semantics of the language. Only a few compilers have themselves been certified correct (one example is CompCert, is the topic of section 6.1), subject to the same observations. These limitations generically apply to all descriptions of verification successes and to any statement that a program has been proved “bug-free” (a qualification which does not appear in this article). To guarantee that a system is “bug-free” one would have to prove that the specifications are correct and complete, the models are consistent with the code, and the compilation preserves the exact semantics of the source code without introducing any bugs.

5 VERIFICATION APPROACHES

The verification of the systems reviewed here relies on a variety of approaches and frameworks.

5.1 Axiomatic semantics

Axiomatic semantic (also called Hoare logic or Floyd-Hoare-Dijkstra semantics) is one of the most widespread formal frameworks. It uses the most annotations and addresses full correctness.

```
sqrt (x: REAL; epsilon: REAL): REAL
    -- Non-negative square root of 'x' with precision 'epsilon'
  require
    x ≥ 0
  do
    . . . Algorithm to compute into Result the square root approximation . . .
  ensure
    Result ≥ 0
    abs (Result ^ 2 − x) ≤ epsilon
  end
```

Axiomatic semantics assumes that the program is equipped with “verification conditions”, also called “assertions”, which may include routine preconditions and postconditions, loop invariants, class invariants (in object-oriented programming), as well as conditions included at arbitrary program places.

A verification condition is a boolean-valued function on program states (or, in the case of postconditions, on two program states, initial and final). Proofs of termination of loops and recursive routines additionally require integer “variants”. An example of a routine annotated with a precondition and a postcondition is, in the notation of the Eiffel programming language
The precondition (**`require`**) expresses a property of `x` in the original state, at the time of a call; the postcondition (**`ensure`**) expresses a property of the **`Result`**, in relation to the value of `x`. The combination of the precondition and postcondition of a routine is its specification, or “contract”.

Verification consists of proving that the implementation matches this specification. It relies on a set of rules (axioms and inference rules) associated with the programming language. An example inference rule (with {P} A {Q} stating that if P, a verification condition, is satisfied prior to the execution of A, then Q will hold afterwards) characterizes the sequencing of instructions as usually represented by the ";" symbol in programming languages:

$$\frac{\{P\}A\{Q\}\{Q\}B\{R\}}{\{P\}A;B\{R\}} \tag{1}$$

The part above the line is a hypothesis; if that hypothesis is satisfied, the inference rule makes it possible to infer the conclusion below the line. The rule states that the sequence A ; B, assuming P on start, will produce R on end if there is an intermediate condition Q such that A ensures Q from P and B ensures R from Q. This sequencing rule is typical of how the rules of axiomatic semantics enable reasoning formally about programs. They can be automated and fed into a proof tool. They make it possible to specify the effect of a program, typically through preconditions and postconditions, and to use the proof tool to prove that the program actually produces that effect. To achieve this result, it is necessary to provide enough verification conditions to guide the proof tool.

5.2 Abstract interpretation

Abstract interpretation is a mathematical framework for performing static analyses of programs, based on mapping concrete domains of execution values onto more abstract domains, so that analyses of important properties (such as arithmetic overflow, pointer nullness or safety constraints), which would be prohibitive on a concrete domain, can be performed on the abstract domain with its results still valid at the concrete level. Abstractions satisfying such properties are so-called *Galois connections*, enjoying conservation properties both ways (concrete to abstract and back).

Determining specific properties of relevant program properties involves writing a set of equations applying on the variables of the program, based on its control flow graph, data flow graph or both. These equations are mutually recursive, implying that the way to obtain a solution is to compute a *fixpoint* of the equations through an iterative method. Such fixpoint computation, and prior to it the construction of the graph and its conversion into a set of equations, are usually impractical or even impossible for the program being verified, if only because the "concrete domains" in which program variables take their values are very large or infinite. Abstract interpretation will instead perform the computation on an abstracted version of the program, in an abstract domain. To ensure that the fixpoint on the abstract domain is reached after a finite set of iterations, it may be necessary to simplify the abstracted computation further through *narrowing* and *widening* operations.

A simple example of abstraction could serve to determine whether a variable x can ever be zero at a certain program point, where the instruction involves a division by x. Assuming for simplicity that the original values (concrete domain) are integers, the abstract domain will only include five values, representing zero (Z), Positive (P), Negative (N), Bottom (representing impossible values) and Top (representing all possible values), with a partial order relation "less defined than" defining a lattice structure. The operations on numbers transpose in the abstract domain: for example Z + Z = Z - Z = Z, P + P = P - N = P, N + N = N - P = N, but P + N = N + P = N - N = P - P = Top (since the last operations may yield a result of any of the categories). The static analysis in the abstract domain can determine whether the abstract value of x can ever be Z, much more easily than if we attempt to perform a similar analysis on the concrete program and domain. Under the appropriate conditions, results obtained in the abstract domain can be mapped back to the original.

5.3 Model-checking

Model-checking is the result of a "why not?" reaction to the accepted wisdom about the impossibility of certain tasks. Exhaustive testing is well known to be impossible in practice, since the number of cases would be infinite. On further analysis, however:

- Computers are finite automata; integers as represented by computers, for example, do not form an infinite set but one limited to (typically) 2^{32} or 2^{64} values. Similarly, the number of times the body of an ordinary "while" loop can be executed is unbounded in principle; but in practice the number of iterations is, in any program execution, not only finite but bounded (since a loop taking 100 years to execute would be of no interest).
- These sizes are extraordinarily large at the human scale, making the number of possible states for any realistic program appear, at first sight, intractable. But modern computers are very powerful, executing billions of operations a second, which may make it possible to achieve the seemingly absurd goal of exhaustive state-space search, perhaps not for the program itself (except in elementary cases) but for a simplified version known as a model.

An elementary example of a program model is a "boolean model" which replaces every integer variable by a boolean one, with False standing for 0 and True for any other integer value, dramatically reducing the number of possible states (an integer variable now yields two states instead of e.g. 2^{64}). Another example of a technique for fighting "state explosion" is loop unfolding, which replaces every loop by a scheme executing the body 1 to N times, often for a small N (typically 2 or 3).

The model-checking algorithm will then verify a specified property, such as liveness (non-deadlock) or non-starvation for a concurrent program, by constructing the set of all possible states of the model program and trying to find a "counter-example": an execution path that leads to a state violating the desired property. This step may or may not be the end of the story:

- If there is a violation of the property (a fault) in the original program, it will (for the appropriate kinds of property) persist in the model. Then, if the state space exploration does not produce a counter-example, it definitely proves that the original program is free from the fault.
- Finding the fault in a state of the model is, however, not conclusive: the fault might be in the original, or it might be an artifact of the reduction to a model. For example, $m \neq n$ between integer variables holds for $m = 1$ and $n = 2$, but in a boolean model both values map to True, allowing the model-checker to find a counter-example. Such cases require refining the model.

5.4 B

Most approaches to verification analyze a program to determine whether it is correct, regardless of how it was produced in the first place. "Correctness by construction" denotes a different process: build software so that it is correct, intertwining the construction and verification efforts. The B method [37] applies this idea, combined with the notion of *refinement*. Most systems using B rely on the "Event B" variant, which adds to the basic framework the notion of event.

A refinement process starts with a very high-level view of the program, involving variables defining a state, abstract events affecting that state, and invariants. As an example, in a system controlling access to a road segment undergoing repair work, an invariant could state that all cars in the system are either stopped or traveling one-way (all East-West or all West-East). The events might be "Let a car enter East", "Let a car exit East", the same for West, "Car arrives East" and "Car arrives West". Variables include: numbers we and ww of cars waiting on the East and West sides, current direction d of travel (boolean), number n of cars currently traveling on the road segment, maximum number N on it. Each event is defined by its effect on the variables (and hence the state); for example each "Enter" event increases n by one, decreases the respective waiting variable (we or ww) by one, and leaves the other variables unchanged. Invariants in this example include $n \geq 0$ and $n \leq N$. Events can have guards, meaning conditions that must be satisfied for an event to occur; for example, the guard for "enter east" is that the direction of travel is East to West, $we > 0$ and $n < N$. The fundamental correctness rule is that every event, when executed with its guard satisfied as well as the invariants, must ensure that the invariant is satisfied again.

The B method works by stepwise refinement. Each step, refining an existing ("abstract") model into a new ("concrete") one, may introduce new events, new invariant properties, and more specific versions of the abstract events. The refinement is correct if the new events preserve both the concrete and abstract invariants. For example, we might refine the road construction model by introducing traffic lights on both ends, with events such as going from green to orange, orange to red etc. on either of them. New invariant properties appear (for example, if the East light is green the West light must be red). We may refine "Let a car enter East" by including the change of light to green. All new versions must satisfy the preceding properties and the new ones.

Refinement proceeds until it has reached a level of detail where the result is explicit enough to be directly implemented in a programming language. With the possible exception of this last translation step, the result is correct by construction, since every step has been proved correct in the sense of invariant preservation.

B-specific proof tools support the method, to prove at each step that new events preserve invariants and that the new invariants imply those of the preceding refinement level.

5.5 SMT solvers

One way to prove a "universal" property, stating that all elements of a set E satisfy a property P, is to prove the absence of a counter-example; in other words, to prove that it is impossible to find an element of E that satisfies $\neg P$, the negation of P.

The properties P of interest, and their negations, are boolean formulae. Disproving P means finding a variable assignment that satisfies $\neg P$. The boolean satisfiability problem is NP-complete, potentially requiring unrealistic computation time, but for theories meeting specific criteria, known as satisfiability modulo theories (SMT), effective algorithms are possible. SMT solvers apply this discovery and lie at the basis of many modern proof tools.

5.6 Z

The Z specification language, a predecessor of B, is a formal specification language based on set theory. Z makes it possible to specify systems in terms of sets and operations on them, represented by mathematical functions and relations. Associated tools support both consistency proofs of the specifications themselves and proofs that programs in specific languages satisfy the specification.

5.7 HOL

HOL (Higher-Order Logic) is a mathematical-logic framework underlying by two of the frameworks used by systems in this survey, HOL4 [38] and Isabelle/HOL [40, 64]. As reflected by the name, it can include logic of several increasing orders: propositional (zero-order), predicate calculus, second-order (with quantification over relations), third-order (with quantification over sets of sets). It also supports typed λ-calculus with object-level polymorphism [35].

5.8 Coq

The Coq framework [20] relies on a different logical basis: constructive intuitionistic type theory. The Coq language serves as both a specification language and a programming language. A formal proof in Coq, according to the Curry–Howard correspondence, has an associated program with a suitable type. It is possible to extract a program, guaranteed correct, directly from the proof.

5.9 Labeled Transition Systems

A Labeled Transition System (LTS) [55] is a mathematical relation $\textit{current state} \xrightarrow{trace} \textit{next state}$ that describes one step of execution of a program and its effect on the program state. The sequence of transitions from an initial state defines the observable behavior. A finite sequence describes a terminating program execution, where the program terminates either normally or with a run-time error. An infinite sequence of transitions describes a program execution that runs forever.

5.10 Other verification approaches

The approaches covered above do not represent the full extent of verification techniques. In particular, an interesting approach has garnered attention in recent years: runtime verification (RV) [7] [25] [75], a light-weight formal verification approach that analyzes execution traces of systems. RV usually involves running the system under analysis, monitoring the system's executions, and checking the system's conformance to formal specifications based on the runtime behaviors. A typical RV process consists of three stages:

- Monitor synthesis: generate, from a formally specified property, a *monitor*, a decision procedure that receives *events* (describing system actions or states) during execution and emits associated *verdicts* (indicating whether the property is satisfied).
- System instrumentation: modifying the system to enable production of events during execution; those events are the inputs to monitors.
- Execution analysis: analyze execution, either step-wise or a posteriori.

Compared to static formal verification approaches (such as model checking and theorem proving), RV is more applicable, as most of the systems are inherently executable; RV may, however, produce false negatives if not all execution traces are monitored.

6 THE SYSTEMS: DESCRIPTIONS

The present section describes the selected systems and reviews them through the ten criteria of the columns in table 2 (3.2). The criteria are mostly self-explanatory, but note the following:

- Verified Properties: properties being verified, e.g. some properties only (such as termination or liveness), or full functional correctness.
- Programming language(s): languages used for the implementation (not the specification).
- Specification/implementation (Spec/Imp) ratio: estimate of ratio between lines of specification/annotation and lines of implementation code.
- Effort (py): estimate of development and verification effort, in person-years.

6.1 CompCert

6.1.1 Scope. CompCert is a compiler for the C programming language, intended for compiling life- and mission-critical software that must meet high levels of assurance.

6.1.2 Components. The CompCert project focuses on compilation, excluding preprocessor, assembler, and linker. The latter components are unverified and come from a legacy compilation tool chain. The compiler supports almost all of the C language (ISO C99) [55], generating code for the PowerPC, ARM, RISC-V and x86 processors.

6.1.3 Verified properties. The compiler includes multiple passes. It formally verifies semantic preservation between the input and output of every pass. To that end it provides formal semantics for every source, intermediate and target language, from C to assembly; such semantics defines the set of all possible program behaviors [55], including termination (normal or abnormal).

These formally verified properties only apply to the correctness of the compiler itself, without guaranteeing the correctness of the compiled software and the absence of harmful events such as null-pointer de-referencing.

CompCert's formal semantics for C and assembly were hand-crafted; while there is a high degree of confidence that it accurately describes these languages, this is not formally guaranteed since the languages themselves are not formally defined.

6.1.4 Project context. The goal of CompCert is to avoid incorrect compilation, which would generate incorrect machine code from a correct source program. Many production compilers have bugs due to the complexity of code generation and optimization. Then, even if the source code of a program has been proved correct, the version actually executed may produce the condition of violations that the program's formal proof was supposed to have rooted out.

6.1.5 Decisions. The compiler is split into 20 passes using 11 intermediate languages [44]. The unverified parts deal with initial preprocessing, type refinement and language-specific simplifications. The formally verified passes follow the conventional multi-pass compiler structure and include parsing, front-end compiler, back-end compiler and assembling. The final assembling and linking steps involve standard non-verified tools.

The staged implementation makes it possible to reason in a modular way about each pass and postpone the verification of certain passes (such as parsing, assembly and linking) to a later step. As long as the intermediate language between two successive passes is the same, the proof of semantics preservation for both passes guarantees semantics preservation for their combination.

The verified compiler code is extracted automatically from the proofs in Coq. The code of the extractor has been proved only on paper, not mechanically [57]. The Coq documentation explicitly states that in some cases the translation from Coq to OCaml may be unsound, for example out of difference between the type systems, and that CompCert developers are responsible for making sure no bad events (such as integer overflow) occur during compiler runs.

A validation tool called Valex compensates for current absence of formal proofs for assembling and linking. It reads and disassembles the generated executable, then compares it with the output produced by CompCert to ensure that there are no injections or other changes.

6.1.6 Tool stack. The verified components of the compiler are written in Coq to guarantee their correctness. Given the formal specification and the associated constructive proof, Coq extracts a certified program from the proof in OCaml. The extracted code becomes part of CompCert.

6.1.7 Style. The semantics of every language is specified in small-step operational style as a labeled transition system (section 5.9). Because the behavior of a program can be nondeterministic, the specification relies on a refinement of the allowed behaviors, replacing every non-deterministic behavior with a more deterministic one (by proving 15 simulation diagrams for each intermediate translator independently and then composing them to establish semantic preservation for the whole compiler).

The specifications and proofs are written in Coq. The soundness theorem [54] states that if the source code S satisfies the specification $Spec$ (written $S \models Spec$): all observable behaviors B of S satisfy $Spec$ ($\forall B.\ S \Downarrow B \implies Spec(B)$), so does the compiled program C:

$$S \models Spec \implies C \models Spec$$

This result comes from two other theorems. One states that every intermediate translator guarantees that the target program P_{tgt} simulates the source program P_{src}: $P_{src} \gtrsim P_{tgt}$. Another one [78] shows that in this case, the behavior of the target program (the set of its execution traces) reproduces the one of the source program: $\mathsf{Beh}(P_{src}) \supseteq \mathsf{Beh}(P_{tgt})$.

6.1.8 Software characteristics. In 2016, authors reported that the source code has 100 000 lines of Coq [55]. CompCert is empirically more reliable than GCC and LLVM: the test generation tool Csmith [92] found 79 bugs in GCC and 202 in LLVM, but none in the verified parts of CompCert.

6.1.9 Project characteristics. The project started around 2005 [19] and resulted in more than 30 publications (conferences, journals and books), as well as 3 PhDs. The first public version of

CompCert (1.2) was released in 2008. The first commercial version has been available since 2015 [55]. As of 2016, the project had required 6 person-years (code + verification).

6.1.10 Lessons. The project demonstrates the feasibility of writing a formally verified compiler for a popular programming language. Correctness comes at a cost: generated programs run 10–20% slower compared to GCC4 with optimizations turned on. Verified compilation alone, however, turns out to be insufficient for industrial applications. Established tool chains require further extension of the compiler (e.g., to produce debug information [55]), as well as refactoring of existing source code (to move compiler-dependent pragmas and hand-coded inline assembly code to the run-time environment [44]). The pipelined nature of compilation supports almost independent development and verification of more than 10 compilation passes. Unverified parts of the compiler rely on additional validation tools to guarantee correctness of the output.

6.2 EiffelBase 2

6.2.1 Scope. EiffelBase 2 is a library of fundamental data structure implementations. The project described here provided a proof of full functional correctness.

6.2.2 Components. EiffelBase 2 covers core data structures of everyday programming such as arrays, lists, queues and stacks. It is a replacement for the EiffelBase library (hereafter EiffelBase 1 to avoid confusion) widely used by Eiffel programmers; both roughly cover the same scope as the Java Collections and .NET Collections library in respectively the Java/JVM and C#/.NET environments.

6.2.3 Verified properties. The verification covers full functional correctness in the following sense: it applies to Eiffel programs equipped with "contracts" (assertions expressing preconditions and postconditions of routines and invariants of loops and classes), and verifies that the code satisfies the specification expressed by the contracts. Hence, it verifies that the code fulfills its stated purpose, not just that it meets specific consistency properties.

6.2.4 Project context. EiffelBase 1 was already designed with correctness concerns in mind; it may have been the first general-purpose library equipped with extensive contracts. These contracts are widely used for dynamic verification, that is to say testing; preconditions, in particular, express conditions imposed on the caller and as a result make it possible, when run-time assertion monitoring is turned on, to find errors in application code using the library.

Dynamic assertion monitoring in a library that is richly equipped with contracts provides an effective way to find bugs, both in client code and in the library itself (in the case of postcondition and invariant violations). It falls short, however, of guaranteeing correctness, which requires static proofs of full functional correctness and is the goal of the EiffelBase 2 verification project.

6.2.5 Decisions. The initial intent of the EiffelBase 2 project was to extend the assertions of EiffelBase 1 to permit static mechanical verification. EiffelBase 2 was devised as a new implementation of the same structures, with a simpler inheritance hierarchy. While not identical, the API and the general style of the new library remain close to those of EiffelBase 1.

6.2.6 Tool stack. The verification uses the AutoProof verification environment for Eiffel, relying on the Boogie proof engine from Microsoft Research, itself based on the Z3 SMT solver (Fig. 1).

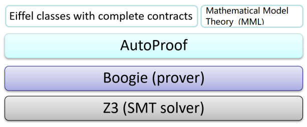

Eiffel classes with complete contracts
Mathematical Model Theory (MML)
AutoProof
Boogie (prover)
Z3 (SMT solver)

Fig. 1. Verification tool stack of EiffelBase 2

6.2.7 Style. The need to extend the assertions of EiffelBase 1 arose for two reasons: ensuring complete functional specification; specifying frame properties; and addressing the "dependent delegate dilemma" of class invariants.
Full functional specification: a typical postcondition of EiffelBase 1, for the remove routine, assuming the structure is a hash table and k is the argument representing the key of the element to be removed, would read

```
ensure
  not has (k)     -- There is no longer any element of key k.
  (old has (k)) implies (count = old count - 1) -- Number of elements decreased (by 1)
  (not old has (k)) implies (count = old count) -- if and only if key k was present
```

This specification is not complete since it does not say anything about elements associated with keys other than k. An implementation could modify some of these elements and still meet the precondition. EiffelBase 2 classes make the specifications complete by using mathematical models; for example the mathematical model for a hash table is a map (a partial function from keys to elements), so that the postcondition can be given as **ensure model** = (**old model**).`deprived` (k), where `deprived` yields a map whose domain has been deprived of k. The original postcondition clauses are consequences of this new one. Models serve specification purposes only and are provided by an auxiliary library, MML (Mathematical Model Library), with classes representing such mathematical objects as sets, functions (including maps) and sequences. They are purely functional in style (for example, no procedure deprive that modifies a map, but a function deprived that yields a new map with the key removed).
Frame conditions: postconditions describe how properties of objects change. For practical proofs, it is also necessary to specify what does not change (and prove that it does not). "Modify" clauses, added to the contracts, specify the frame of a routine: the objects that it is allowed to change.
Dependent delegate dilemma: a delicate issue of the semantics of object-oriented programs is the precise role of class invariants, which are only expected to hold on routine entry and exit but can be violated in-between, making the underlying object "inconsistent". The "Dependent Delegate Dilemma" is the case of a routine r calling another s in a state where the source object O is consistent (as permitted), but s calling back into a routine t of O, whose correctness requires a consistent object on entry. To address this issue, AutoProof and EiffelBase 2 rely on a semantic collaboration model, which requires further annotations.

6.2.8 Software characteristics. While EiffelBase 2 is fully functional, the majority of Eiffel users continue to use EiffelBase 1 for reasons of compatibility with existing code.

6.2.9 Project characteristics. The experience of the EiffelBase 2 project shows that a naive view, "write fully specified code, then run it through the verifier", does not reflect the reality of proving program correctness. The process is an iterative one: write code with contracts, attempt to verify, see which parts do not verify, fix the problem (by modifying either the code or the contracts), repeat. It can be compared to the traditional process of debugging a program, except that the debugging is static (running the prover) rather than dynamic (running the program). The process requires significant effort; the principal author of EiffelBase 2 spent 6 months after the completion of her PhD thesis finishing the EiffelBase 2 verification, applying the process described.

The verification required about 5 person-years of work. It was part of the more general AutoProof project, consuming around 10 person-years. Project members were researchers.

6.2.10 Lessons. Libraries are keys to modern programming, with its emphasis on reuse; their correctness is paramount since so many applications will rely on a given library. The verification of EiffelBase 2 is, consequently, a milestone.

The major challenges in generalizing the technology pioneered by the verification of EiffelBase 2 are the level of competence required (project members all obtained PhDs in computer science) and the extensive annotation effort required for frame specification and, particularly, semantic collaboration, the hardest to teach to newcomers. Research efforts are in progress to address both of these needs by analyzing the programs to produce the corresponding specifications automatically.

6.3 HACL*

6.3.1 Scope. HACL* [99] is a verified portable C library of cryptographic primitives for a new mandatory cipher suite in TLS 1.3 [72]. It is also intended as the main cryptographic provider for the miTLS [12] verified implementation and integrated in Mozilla NSS cryptographic library.

6.3.2 Components. All cryptographic algorithms in HACL* are correct by construction.

6.3.3 Verified properties. The developers of HACL* have verified the following properties:

- Memory safety. The code never reads or writes memory at invalid locations, such as null or freed pointers, unallocated memory, or out-of-bounds of allocated memory. In addition, any locally allocated memory is eventually freed (exactly once).
- Functional correctness. The code of each primitive conforms to its published standard specification on all inputs.
- Mitigations against Side-Channel Attacks. The code does not reveal any secret inputs to an adversary, even if the adversary can observe low-level runtime behavior such as branching, memory access patterns, cache hits and misses or power consumption.
- Cryptographic security. The code of each cryptographic construct implemented by the library is (with high probability) indistinguishable from a standard security definition.

6.3.4 Project context. The project's primary goal was to build a reference implementation in C and to prove that it conforms to computations described in the corresponding RFC standards (Request For Comments — a publication in a series, usually for the Internet).

6.3.5 Decisions. The main implementation vehicles for HACL* are the F* functional language and Low*, an integration into F* of a safe subset of C. HACL* code never allocates memory on the heap; all temporary states are stored on the stack to simplify proofs of memory safety and avoid explicit memory management. The Low* source code is broken into many small functions to improve readability, modularity and code sharing and reduce the complexity of each proof. Consequently, the default translation of this code to C would result in a set of small C functions, which can be verbose and hurts runtime performance with some compilers like CompCert (Section 6.1). For better control of the generated code, program annotations direct the KreMLin compiler to bring certain functions online and unroll certain loops. A large chunk of the bignum verified code is shared across three different encryption algorithms: Poly1305, Curve25519 and Ed25519, meaning that this code is verified once but used in three different ways. The sharing has no impact on the quality of the generated code because KreMLin puts the generic code online and specializes it for one particular set of bignum parameters. The result is that Poly1305 and Curve25519 contain separate, specialized versions of the original Low* bignum library.

6.3.6 Tool stack. The developers first write a high-level specification of the primitive in a higher-order purely functional subset of F* (Pure F*). They then write an optimized implementation in Low*, a low-level subset of F* that can be efficiently compiled to C. They then verify the code using the F* Z3-based type checker, for conformance to the specification and to the logical preconditions and type abstractions required by the F* standard library. If type checking fails, there may be a bug in the code, or the type checker may require more annotations. Finally, KreML translates the Low* code to C. The translation is actually to Clight, a subset of C that can be compiled with CompCert (Section 6.1), although in practice this last step uses GCC for performance reasons.

6.3.7 Style. Fig. 2a contains an example of a pure F* executable specification. To implement prime field arithmetic for the Poly1305 algorithm on 64-bit platforms, one strategy is to represent each 130-bit field element as an array of three 64-bit limbs, where each limb uses 42 or 44 bits and so has room to grow. When adding two such field elements, one can simply add the arrays point-wise, ignoring carries; the *fsum* function performs this task. Fig. 2b contains a contract of the future implementation (fadd) of the abstraction (fsum) expressed in pure F* (Fig. 2a).

```
type limbs = b:buffer uint64_s{length b = 3}
let fsum (a:limbs) (b:limbs) =
a.(0ul) ← a.(0ul) + b.(0ul);
a.(1ul) ← a.(1ul) + b.(1ul);
a.(2ul) ← a.(2ul) + b.(2ul)
```

(a) The pure F* abstraction.

```
val fsum: a:limbs → b:limbs → Stack unit
(requires (λ h0 → live h0 a ∧ live h0 b
∧ disjoint a b
∧ index h0.[a] 0 + index h0.[b] 0 <MAX_UINT64
∧ index h0.[a] 1 + index h0.[b] 1 <MAX_UINT64
∧ index h0.[a] 2 + index h0.[b] 2 <MAX_UINT64))
(ensures (λ h0 _h1 → modifies_1 a h0 h1
∧ eval h1 a = fadd (eval h0 a) (eval h0 b)))
```

(b) F* contract

Fig. 2. The specification style of HACL*.

The contract is proven to correctly implement the F* abstraction; then the Low* implementation is proven to correctly implement the contract. After that, the KreMLin tool converts Low* to Clight.

6.3.8 Software characteristics. The library's code base totals 801 lines of pure F* code (specification), 22926 lines of Low* code (code and proofs), and 7225 lines of C code (translated from Low*) [99], and gets verified in 9127 seconds.

6.3.9 Project characteristics. Symmetric algorithms like Chacha20 and SHA2 do not involve sophisticated mathematics, and are relatively easy to prove. The proof-to-code ratio hovers around 2, and each primitive took around one person-week. Code that involves bignums requires more advanced reasoning. While the cost of proving the shared bignum code is constant, each new primitive requires a fresh verification effort. The proof-to-code ratio is up to 6, and verifying Poly1305, X25519 and Ed25519 took several person-months. High-level APIs like AEAD and SecretBox have comparably little proof substance, and took a few person-days.

6.3.10 Lessons. Although formal verification can significantly improve confidence in a cryptographic library, it relies on a large trusted computing base. The semantics of F* has been formalized and the translation to C has been proven to be correct on paper, but the results still rely on the correctness of the F* type checker, the Z3 SMT solver, the KreMLin compiler, and the C compiler (when GCC is used instead of CompCert).

One goal of the design of such systems is to achieve "secret independence", a necessary but not complete mitigation of side-channel attacks. Secret independence provably rules out certain classes of timing side-channel leaks that rely on branching and memory accesses, but does not account for others such as power analysis. The current model only protects secrets at the level of machine integers; it treats lengths and indexes of buffers as public values, so that it is impossible to verify implementations that rely on constant-time table access or length-hiding constructions.

The verification tool chain stops with verified C code. The research team did not consider the problem of compiling C to verified assembly, which is an important but independent challenge.

HACL* performs well for symmetric algorithms on 32-bit platforms, and suffers a penalty for algorithms that rely on 128-bit integers because they are represented as pairs of 64-bit integers. If CompCert (Section 6.1) supports 128-bit integers in the future, the penalty will disappear.

Verification may help optimize code. In the process of verifying HACL*, the researchers found that some algorithms are too conservative. They optimized them and then successfully verified the correctness of the optimization.

Taking an RFC and writing a specification for it in F* is straightforward, as well as translating existing C algorithms to Low*. Proving that the Low* code is memory safe, secret independent, and that it implements the RFC specification is the bulk of the work.

6.4 seL4

6.4.1 Scope. In an operating system (OS), a kernel is a central component that runs in a privileged mode to manage the communication between software and hardware. seL4 [30] is a microkernel that provides essential kernel services.

6.4.2 Verified System. As a microkernel, seL4 only undertakes the minimum tasks required in privileged mode, leaving others to user mode. Core OS functions include isolation through sandboxes, in which a program can execute without interference from others, and inter-process communication (IPC) to transport function inputs and outputs between programs. seL4 is also a hypervisor that creates and executes virtual machines (VMs) running mainstream OSes (such as Linux). This enables provisioning system services, borrowing services from guest OSes running in seL4's VMs.

6.4.3 Verified properties. The verification of seL4 covers the following aspects [13, 48, 76, 77]:

- Functional correctness: the C implementation of seL4 is free of defects (e.g., system crashes, unsafe operations).
- Security properties: confidentiality, integrity and availability.
- Worst-case execution-time (WCET) analysis by obtaining hard upper bounds for all system call latencies.
- Semantics equivalence: the binary is a correct translation of the verified C implementation.

6.4.4 Project context. A fault in an OS kernel can compromise the safety and correctness of the rest of the OS. In other words, a kernel is a trusted computing base (TCB) that must operate correctly for the OS to be secure. SeL4 is one of the L4 microkernels [88] whose TCB is minimized to reduce its exposure to attacks. They aim to achieve greater confidence and better performance. The objectives of the project include: 1) having its implementation proved correct; 2) solving problems on resource management in microkernels and OSes in general; 3) making the system suitable for real-world use.

6.4.5 Decisions. The design of seL4 conforms to the principle of least authorities (POLA) [89] —the rights given to a component are restricted to an absolute minimum. To perform access control in line with POLA, seL4 uses capabilities, which are access tokens that delegate to components the rights (read or write) to use certain objects.

6.4.6 Tool stack. The proof tool is Isabelle/HOL [64]. Two SMT solvers — Z3 and Sonolar — are used to validate the semantic equivalence between C code and binary. The WCET analysis was performed using the Chronos tool [13].

6.4.7 Style. Functional specifications of seL4 are given in HOL. Fig. 3 depicts an example of HOL specification for scheduling: the scheduler is modelled as a function picking one `thread` from all active runnable `threads` in the OS or from the idle threads; the function *all_active_tcbs* returns the abstract set of all runnable threads in the system; the **`select`** statement picks any element from the set `threads`. The `OR` operator indicates non-deterministic choice between the first block (embraced by **`do`** and **`od`**) and `switch_to_idle_thread`.

```
schedule ≡ do
   threads ⟵ all_active_tcbs;
   thread ⟵ select threads;
   switch_to_thread thread
od OR switch_to_idle_thread
```

Fig. 3. HOL specification for scheduler

6.4.8 Software characteristics. seL4 documentation reports exceptionally good performance, complementing the guarantees of safety and security. IPC (interprocess communication) performance is 2 to 10 times faster than some existing microkernels [30] [62]: seL4 is more than five times faster than CertiKOS for a round-trip IPC operation, over twice as fast as Fiasco.OC and 9 times faster than Zircon. An example of practical deployment of seL4 is the DARPA High-Assurance Cyber Military Systems (HACMS) [26], where seL4 guarantees non-interference between components and protects from tampering the computer in the Boeing ULB autonomous helicopter.

6.4.9 Project characteristics. The total development efforts is 31.2 person/years; the effort for the seL4 functional correctness proof, about 20.5; for the security proof 4.1 and the binary verification 2. Re-verification of full functional correctness is feasible [48]; adding a significant feature (for example, a new data structure) to the kernel would cost 1.5 to 2 p/y.

6.4.10 Lessons. The cost for development of seL4 and its functional correctness proof is $362/LOC [48] — inexpensive compared to the industry's widely accepted baseline of $1000/LOC for the EAL6 low level ("Common Criteria" standard [86]), while the assurance level of seL4 is higher than EAL6. These results suggest that formally-verified software can be less expensive than traditionally engineered high-assurance software, while providing stronger assurance [48].

6.5 mCertiKOS

6.5.1 Scope. mCertiKOS is a fully verified single-core operating system kernel that can serve as a hypervisor capable of booting unmodified Linux guests. It is deployed on the Landshark Unmanned Ground Vehicle (UGV) and other American-built car platforms [61].

6.5.2 Components. The project included the development and verification of several variations of the mCertiKOS kernel. An OS kernel is the portion of the OS code that is always resident in memory, facilitating interactions between hardware and software components [87].

6.5.3 Verified properties. The research team has verified security [21], as well as behavioral and strong contextual correctness properties (described below) [31].

6.5.4 Project context. Modern computer systems consist of abstraction layers, such as OS kernels, device drivers, network protocols. Each layer defines an interface that hides the implementation details of a particular functionality level. A client program built on top of a given layer should be able to understand the layer functionality solely from the layer's interface, independently of its implementation. The mCertiKOS project formalizes this methodology through an *abstraction layer calculus*, relying on *deep specifications*, to specify, implement and verify several OS kernels. A deep specification is supposed to capture the precise functionality of the underlying implementation.

6.5.5 Decisions. The researchers wanted deep specifications to satisfy the *implementation independence* property: two distinct implementations M_1 and M_2 of the same deep specification L should have *contextually equivalent* behaviors. Specifically, given any *whole-program* client P built on top of L, running $P \oplus M_1$ (P linked with M_1) should lead to the same observable result as $P \oplus M_2$. This requirement influenced the following project decisions:

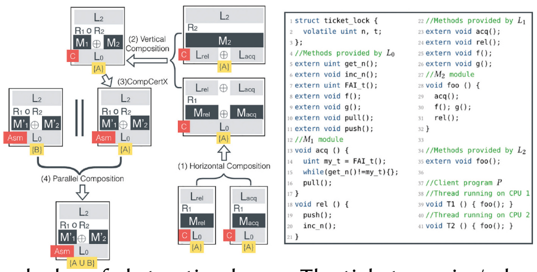
- Focus on languages whose semantics are *deterministic* relative to external events [83].
- Consider only interfaces whose primitives have deterministic specifications.
- Prohibit a client program from calling primitives defined in two different interfaces that reflect conflicting abstract states of the same abstract state.

6.5.6 Tool stack. The project relies on the following technologies:

- ClightX, a variant of the CompCert (Section 6.1) Clight language [21].
- LAsm, an x86 assembly language.
- Coq for development and refinement of abstraction layers' interfaces.
- CompCertX — a verified compiler for compiling ClightX abstraction layers in LAsm layers.

Outside the verified kernels, there are 300 lines of C and 170 lines of x86 assembly code that are not verified yet: the preinit procedure, the ELF loader used by user process creation, and functions such as `memcpy` which currently cannot be verified due to a limitation of the CompCert memory model. Device drivers are not verified because LAsm lacks device models for expressing the correctness statement. The CompCert assembler for converting LAsm into machine code remains unverified. LAsm does not cover the full x86 instruction set and many other hardware features.

6.5.7 Style. The certified abstraction layer, $L_0 \vdash_{R_1} M_{acq} : L_{acq}$ (Fig. 4), is a predicate plus a mechanized proof showing that the implementation M_{acq} running on the underlay interface L_0, written in C, correctly implements the desirable overlay interface L_{acq}, written in Coq. The implementation relation is denoted by R_1. The layer can be (1) horizontally composed with another layer (such as the lock release operation) if they have identical state views (with the same R_1) and are based on the same L_0. The composed layer can also be (2) vertically composed with another that relies on its overlay interface. The Coq code extraction engine produces an OCaml program containing CompCertX, the abstract syntax trees of the kernels, and the driver functions that invoke (3) CompCertX on pieces of ClightX code and generate the full assembly file. In a concurrent setting, these layers can also be (4) composed in parallel.



```
struct ticket_lock {
  volatile uint n, t;
};
//Methods provided by L0
extern uint get_n();
extern void inc_n();
extern uint FAI_t();
extern void f();
extern void g();
extern void pull();
extern void push();
//M1 module
void acq () {
  uint my_t = FAI_t();
  while(get_n()!=my_t){};
  pull();
}
void rel () {
  push();
  inc_n();
}
//Methods provided by L1
extern void acq();
extern void rel();
extern void f();
extern void g();
//M2 module
void foo () {
  acq();
  f(); g();
  rel();
}

//Methods provided by L2
extern void foo();

//Client program P
//Thread running on CPU 1
void T1 () { foo(); }
//Thread running on CPU 2
void T2 () { foo(); }
```

Fig. 4. The calculus of abstraction layers. The ticket acquire/release example.

6.5.8 Software characteristics. The project resulted in a **calculus of abstraction layers** and a series of operating system kernels developed with the help of this calculus:

- The baseline version of the mCertiKOS kernel.
- The mCertiKOS-hyp kernel adds core primitives to build user-level hypervisors by supporting the AMD SVM hardware virtualization technology.
- mCertiKOS-rz augments mCertiKOS-hyp with support of ring 0 processes.
- mCertiKOS-emb is a minimal operating system suitable for embedded environments.

The research team reports 3000 lines of ClightX and LAsm code for mCertiKOS-hyp, calling it "the most realistic kernel". The authors do not share the exact number of lines of Coq code, but we estimate around 18500 lines based on the information distributed throughout the papers.

6.5.9 Project characteristics. The planning and development of mCertiKOS took 9.5 person-months plus 2 person-months on linking and code extraction. It took additional 1.5 person-months to finish mCertiKOS-hyp, after which mCertiKOS-rz and mCertiKOS-emb took half a person-month each.

6.5.10 Lessons. The layered architecture facilitates reuse: the certified abstraction layers resulting from the development of the base version of the mCertiKOS kernel were heavily reused when developing the three variations. The authors, however, report up to 2x slowdown of the mCertiKOS-hyp kernel in the lmbench benchmarks [58]. The certified abstraction layers approach has demonstrated high applicability for verifying strong properties, such as termination and contextual correctness. The specification/implementation ratio of 6, however, looks high compared to the other approaches discussed in this survey.

6.6 ExpressOS

6.6.1 Scope. ExpressOS is a mobile operating system kernel featuring the interface of Android that enforces seven security invariants. Those proven invariants provide a foundation for applications to isolate their state and run-time events from other applications running on the same device.

6.6.2 Components. The scope of verification is limited to the ExpressOS kernel. The ExpressOS kernel is responsible for managing the underlying hardware resources and providing abstractions to applications running above.

6.6.3 Verified properties. The verification effort included checking 7 security invariants that guarantee storage security, memory isolation, UI isolation, and secure IPC.

6.6.4 Project context. The distinguishing characteristic of the project was its focus on preserving security invariants, rather than establishing full functional correctness. It allowed the researchers to fully verify the resulting system against these invariants, whereas attempts to undertake the cumbersome full functional verification only partially verified other Unix-based kernels. The project was motivated by the lack of formal guarantees that mobile applications do not cross each other's security boundaries as they are constantly (re)installed and removed, on smartphones that retain a near-constant Internet connection.

6.6.5 Decisions. ExpressOS relies on four main techniques to simplify verification effort: [59]:

(1) It pushes functionality into microkernel services, reducing the amount of code that needs to be verified.
(2) It deploys end-to-end mechanisms in the kernel to defend against compromised services; for example, the kernel encrypts all data before sending it to the file system service.
(3) It relies on programming language type-safety to isolate control and data flows.
(4) It makes minor changes to the IPC system-call interface to expose explicitly IPC security policy data to the kernel.

6.6.6 Tool stack. The ExpressOS kernel is implemented in C# and Dafny [53]. The verification uses Boogie proof engine [52] and Z3 SMT solver [23] from Microsoft Research. The researchers combined two specification and verification techniques to keep the annotation burden under control: (i) using Code contracts in C# code for specifying lightweight properties, then verified using abstract interpretation based techniques, (ii) using Dafny for specifying, implementing, and verifying data structures that are less amenable to straightforward verification, such as doubly linked lists. Both C# and Dafny code compile to object code for native execution.

6.6.7 Style. The following property, which promotes memory isolation, illustrates well the specification style of ExpressOS:

Property 5- *When the page fault handler serves a file-backed page for a process, the file has to be opened by the same process.*
Fig. 5 contains a fragment of the C# method that handles page faults and includes an assertion enforcing the required property.

```
static uint HandlePageFault(Process proc, uint faultType, Pointer faultAddress, Pointer faultIP) {
  Contract.Requires(proc.ObjectInvariant());
  ...
  AddressSpace space = proc.Space;
  var region = space.Regions.Find(faultAddress);
  ...
  if (shared_memory_region) {...} else { ...
    if (region.File != null) {
      // Assertion of Property 5
      Contract.Assert(region.File.GhostOwner == proc);
      var r = region.File.Read(...); ... } ... } ... }
```

Fig. 5. Property 5 in code contracts.

The invariant of class *MemoryRegion* (Fig. 6), implemented in Dafny, ensures the assertion in the C# client code. The *GhostOwner* built-in ghost field of an object denotes the object's owner. In this specific case, it has type *Process* because every memory region is owned by a process. The invariant of a memory region states that the file backing the region is owned by the same process owning the region, which is what the required property says.

```
class MemoryRegion { ...
  var File: File; var GhostOwner: Process;
  function ObjectInvariant() : bool ... { File != null ==> File.GhostOwner == GhostOwner ... }}
```

Fig. 6. Dafny code ensuring the C# assertion of Property 5.

The verification results from Dafny are expressed as properties of ghost variables, such as the *GhostOwner* field, exported to code contracts as ground facts with the *Contract.Assume()* statements. *GhostOwner* fields also appear read-only C# annotations to ensure soundness.

6.6.8 Software characteristics. The resulting system's performance is comparable to that of Android: it only adds 16% overhead on average to the page load latency time for nine popular web sites. Focusing on security invariants and combining specification and verification techniques made it possible to achieve around 2.8% annotation overhead. To evaluate the security of ExpressOS, the authors have built a prototype system and have examined 383 vulnerabilities listed in CVE [17]. The ExpressOS architecture successfully prevents 364 of them [59].

6.6.9 Lessons. During the verification process, the specification using ghost code was sometimes too closely intertwined with the implementation. The authors believe that an alternate mechanism for writing specifications that is a bit more code-independent, resilient to code changes, and yet facilitates automated proving would make the developers' work more robust and productive.

Two other observations made by the authors are that combining code contracts and Dafny reduced the code to annotation ratio down to about 2.8% (implementing the annotations' specification) and that the redesign of the data structures and the implementation of the property using ghost variables made verification easier.

6.7 NATS iFACTS

6.7.1 Scope. With the ever-increasing capacity of air traffic over the UK, the National Air Traffic Service of UK developed a set of electronic tools to assist air-traffic control. The tool set is called interim Future Area Control Tools Support (iFACTS), which was proven free of run-time errors.

6.7.2 Components. iFACTS provides functions to help human controllers to handle air traffic, including electronic management of flight progress strip, trajectory prediction and conflict detection. iFACTS enables controllers to look up to 18 minutes ahead. This allows them to test the viability of various options available for a maneuvering aircraft, and thus controllers have more time for decision-making. As the flights progress, the tool updates predictions information (positions, altitude and headings of all aircrafts in their sectors of interest) and the controllers issue new clearances based on the real-time predictions.

6.7.3 Verified properties. The proof of iFACTS concentrates on type-safety properties: the absence of run-time exceptions such as buffer overflow, arithmetic overflow or division by zero. The verification also covers functional correctness and memory usage constraints [71].

6.7.4 Project context. The integration of iFACTS into industrial systems requires the system to be certified according to Civil Aviation Authority SW01 standards [5]. iFACTS was the first system developed under these new regulations.

6.7.5 Tool stack. Implementation of iFACTS used the SPARK framework; the verification tool set includes the Examiner and Simplifier; the Examiner generates verification conditions (VCs) in FDL language (a typed first-order logic) from input SPARK programs; the Simplifier is a theorem prover for discharging the VCs.

6.7.6 Style. In the requirement and design phases, the specifications of iFACTS are Z notations, the basic building block is a schema, which defines variables, constraints and operations in terms of variables.

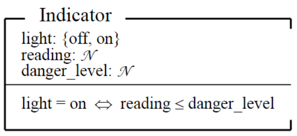

Indicator
light: {off, on}
reading: $\mathcal{N}$
danger_level: $\mathcal{N}$
light = on $\Leftrightarrow$ reading $\leq$ danger_level

Fig. 7. The graphical layout of Indicator schema

Fig. 7 shows an example of a schema (from SHOLIS [46]), describing a *state entity* named `Indicator`. The upper section (or *signature*) declares three variables and their types; the lower section (*predicate*) defines logical relationships between the three variables. In the implementation phase, specifications are given as SPARK annotations, which describe data- and information-flow properties as well as functional properties. Those SPARK annotations were generated from the Z specifications in the previous design phase. Annotations are encoded in comments via the prefix `-#`. For example, "`-#global`" indicates the use of certain global variables; "`-#derives`" specifies the dependency between the variables; "`-#pre and -#post` " define pre- and post-conditions of a certain procedure.

6.7.7 Software characteristics. The Examiner generates 152927 VCs from the SPARK code; the Simplifier discharged 151026 VCs (98.76%) automatically and the remaining 1701 VCs were analyzed manually; a single proof run took 3 hours [14]. Owing to the tools' ability to parallelize and to exploit caching of proof results, re-verification of small changes can take only seconds. In addition, the development also established a 'proof-first' style — all code changes must be proved correct before being committed.

iFACTS have been in full operation in NATS' facility since November, 2011. The deployment of iFACTS has enhanced the capacity of airspace and the efficiency of air-traffic controllers. It was reported that there was an average 15% increase in airspace capacity in the UK in 2012 [63].

6.7.8 Project characteristics. The research and development team of the project comprised over 140 engineers from NATS and Altran and the project spanned over 10 years; they carried out a one-week formal training, including the training on Z notation and SPARK.

6.7.9 Lessons. Verification of iFACTS scaled much better than previous SPARK projects such as SHOLIS and C130J, thanks to the considerable improvement in computing resources and theorem-provers [22]. The further reduction of time consumption of the overall project, however, is limited due to the rudimentary tool support for Z; as they used MS Word documents to store Z specifications, merging different versions of large documents (over 1000 pages documents) would be difficult and time-consuming.

What makes this verification project stand out from others includes not only the scale of implementation (iFACTS is one of the most ambitious SPARK project to date) but also the application of formal methods throughout the complete development lifecycle. Additionally, the experience showed that training engineers was not a barrier — engineers of diverse programming backgrounds could easily pick up verification languages/tools [85].

6.8 Ynot

6.8.1 Scope. Relational database management systems (RDBMSs) allow application developers to have a high-level specification of the behavior of the data manager (one of the modules in the RDBMS), which is suitable for formal reasoning about application-level security and correctness properties. It is one of the systems implemented by the team of the Ynot Project.

6.8.2 Components. The Ynot RDBMS is a lightweight, fully verified, in-memory database management system. A command-line interface serves to create tables, load tuples into a table, save/restore a table to/from disk, and query tables using an SQL subset.

6.8.3 Verified properties. The main verification task focuses on the correctness of executing queries in the RDBMS regarding denotational semantics of SQL and relations.

6.8.4 Project context. RDBMSs are omnipresent in modern application software. They are used to store data whose integrity and confidentiality must be maintained. The implementation of the data manager should ensure that a bug cannot bring about accidental corruption or disclosure.

6.8.5 Decisions. The design decisions of RDBMSs seek to avoid complicated proofs and reduce computations. The introduction of ptrees and their association with B+ trees [1] as ghost states help in these simplifications. Avoiding disjunctions when something can be readily computed also helps reduce computations. A ptree is a defined functional model of the tree, simplifying the task of defining predicates that describe when a heap contains a tree with a valid shape.

6.8.6 Tool stack. The verification of the system involved two tools: the formal proof management system Coq and the Ynot extension to Coq. The proof that implementation meets the specification is written and verified in Coq. The Ynot extension to Coq is designed to support writing and proving correct pointer-based code, using a variant of separation logic.

6.8.7 Style. The specification language is Coq, which the team used to develop mathematical theories and prove specifications, relying on OCaml to create Coq plugins for novel tactics or functionality. The following Coq specification example ensures correct usage of semantic optimizations:

```
Definition accurate (m: DbInfo) (G: Ctx) (E: Env G) : Prop :=
forall s I, getRelation s I E = empty < − > isEmpty (m s I)
```

Here, `DbInfo` stands for database information; `Ctx` means context; `Env` describes environment; `s` and `I` denote a string and a schema respectively.

6.8.8 Software characteristics. The Ynot RDBMS is a proven lightweight memory system, in which the functional specification of system behavior, system implementation, and proof of achieving compliance with the specification were written and verified in Coq.

6.8.9 Project characteristics. Four people spent two years finishing this project.

6.8.10 Lessons. Successful application for real systems requires a number of tasks. Some need relatively modest extensions to the current implementation; for example, key information can be integrated to enable efficient point and range queries. Reifying the low-level query plan will help fuse operations to avoid materializing transient data (the low-level queries are those executed by the RDBMS using a sequence of operations over imperative finite maps). Other tasks demand more effort. Realizing the ACID (Atomicity, Consistency, Isolation, Durability) guarantee of concurrency, transactional atomicity and isolation, and fault-tolerant storage is a challenging task. Implementing and verifying the correctness of high-performance, concurrent B+ tree is another challenge.

6.9 Roissy Shuttle

6.9.1 Scope. The Roissy VAL shuttle is a system for running driverless shuttles in Charles de Gaulle airport in Paris. The safety-critical software, developed in this project with the B Method, controls the speed of the shuttles.

6.9.2 Components. A shuttle is an automatic light train running on the line connecting Roissy terminal 1 to terminal 2. The line is made up of 5 sections. Each section has a wayside control unit (WCU) that controls the speed of the driverless shuttle in the section, composed, in turn, of a set of blocks. WCUs should localize the shuttle by checking whether a block is occupied. WCUs receive the commands from the traffic control center and drive the shuttles based on the commands. The control logic of WCU is implemented as a safety critical software, called WCU-SCS.

6.9.3 Verified properties. The development of this system took place in three phases: abstract model construction, concrete model construction, and code generation. In the abstract model construction phase, the informal software specifications, given in natural language, were formalized into an abstract model. In the concrete model construction phase, a concrete model was built from the not implementable parts of the abstract model. In the code generation phase, the Ada code was generated based on both the abstract model and the concrete model. The verified properties include safety requirements for the abstract model, compliance between the concrete model and the abstract model, compliance between the code and the models.

6.9.4 Project context. The whole VAL system was developed by Siemens Transportation Systems and the development of WCU-SCS has been subcontracted to ClearSy. The objective of the project was to ensure that WCU-SCS satisfied a set of safety properties and that its development was within budget and schedule.

6.9.5 Decisions. ClearSy applied the Siemens B Method, a process designed for using B language to build correct software by construction. The functionalities of WCU-SCS were firstly modeled in B. After establishing their correctness, the B models were then translated into Ada code. Unit testing was not performed since most of the effort was devoted to the early specification phase, to build correct software directly.

6.9.6 Tool stack. Refinement from abstract model to concrete model was performed using two semi-automatic refinement tools, EDiTh B and Bertille. The proof of the B models was made by the automatic prover of Atelier B. Generation of ADA code uses the Digisafe-ADA technology.

6.9.7 Style. The high-level properties of the VAL system models are specified using abstract data types such as sets of scalar types, relations, partial and total functions. An example of a property for the abstract model is illustrated below.

```
∀ block (block ∈ t_block ∧ (( ctx_block_bs_up [{block}] ∪ ctx_block_bs_down [{block}] ∩ cut_beam_sensors ≠ ∅
        ∨ ctx_block_detector [{block}] ⊆ occupied_block_detectors) ⇒ block ∈ occupied_blocks)
```

t_block represents the block type and $ctx\_block\_xxx[\{block\}]$ denotes abstract variables for a given $block$. The property specifies that a block is regarded as occupied when a beam sensor located at one of the block borders is cut (denoted by $cut\_beam\_sensors \neq \emptyset$) or when the block detector is occupied (denoted by $ctx\_block\_detector[\{block\}] \subseteq occupied\_block\_detectors$).

6.9.8 Software characteristics. Automatic refinement can lower the costs of a software development, with an acceptable price of usually 10% slower code compared to the handwritten version. In addition, although the approach can ensure that the code correctly implements the B model, it is difficult for developers to link the auto-generated code with the corresponding software specification.
It was easier to prove the concrete model than the abstract model. The ratio between the effort on the abstract model phase and the effort on the concrete model phase is 2. The abstract model proof consists of complex and lengthy interactive demonstrations, making the proof difficult to reuse. The concrete model proof is broken down into small steps with the same patterns.As far as maintenance is concerned, it is guaranteed that the modified version will remain consistent as long as the proof is completely redone. The cost will also be limited since 1) the whole development environment has been set up and 2) the refinement and proof rules are likely to be reused.

6.9.9 Project characteristics. The interactive verification took about 50 person-days.

6.9.10 Lessons. The method used in this project is also suitable for other industrial domains. It is suitable for software mainly based on discrete logic description, but not for software based on continuous calculation or on floating point numbers which cannot be considered as decimals.

While compliance between the code and the model has been established by a mechanical proof, compliance between the model and the natural language specifications is checked manually, which still leaves room for potential errors.

6.10 Sizewell B

6.10.1 Scope. Sizewell 'B' is a Westinghouse-designed Nuclear Pressurised Water Reactor (PWR) built in Sizewell, Suffolk in the UK. It possesses two diverse protection systems whose role is to provide an automatic reactor trip when plant conditions reach safety limits and to actuate emergency safeguard features to limit the consequences of a failure condition.

6.10.2 Components. Nuclear Electric plc constructed and operated the Sizewell B PWR. Two systems provides overall protection (Primary Protection System or PPS) and secondary protection (SPS). The PPS uses microprocessor-based logic, while the SPS uses magnetic core logic (laddic) technology. The design requirement for the PPS was to provide reactor trip and engineered safety feature (ESF) actuation for all design faults. The requirement for the SPS was to provide reactor trip and ESF actuation, in parallel with the PPS, for faults with a frequency in excess of about once in 1000 years. The reliability targets of the two systems were therefore 1 in 10,000 for PPS and 1 in 100,000 for SPS. They could have been achieved by hardware, but it was necessary to guaranteeing the same level on the software side. A full code and compliance analysis was performed with the MALPAS analyzer; it showed that the code fully matched the requirements.

6.10.3 Verified properties. The verification ensures that the implementation of complex protective functions is accurate and that there is a high degree of confidence in this.

6.10.4 Project context. The confirmatory assessment of the software using MALPAS (consisting of static and semantic analysis tools for all safety software) was conducted under the TACIS program of the European Union. It took place in a large acceptance process. An extensive program of software verification was carried out on the PPS software to detect and remove any errors generated in the design and coding phases. All of the assessments were carried out by independent companies or independent entities of the Nuclear Electric plc.

6.10.5 Decisions. The analysis of the PPS software was conducted on a procedure-by-procedure basis in a bottom-up manner. The analysis commenced with procedures that call no others, and then progresses up the call hierarchy until the top level application code is reached. All the MALPAS analysers were run on each procedure and the results verified against two levels of specification: a higher level specification (the Software Design Requirements, SDR) and a lower level (the Software Design Specification, SDS). The SDR was the primary document against which the code was verified, with supporting information provided by the SDS. All the code that can be accessed during on-line operation was subjected to MALPAS analysis.

6.10.6 Tool stack. The MALPAS toolset comprises five specific analysis tools that address various properties of a program. The input to the analyzers needs to be written in MALPAS Intermediate Language (IL) which is produced by an automated translation tool from the original source code, assisted by an analyst which can make suitable manual changes. The MALPAS toolset consists of five analyzers:

(1) Control Flow Analyser examines the program structure, and provides a summary report drawing attention to undesirable constructs and an indication of the complexity of the program structure.
(2) Data Use Analyser separates the variables and parameters used by the program into distinct classes and identifies errors.
(3) Information Flow Analyser identifies the data and branch dependencies for each output variable or parameter.
(4) Semantic Analyser reveals the exact functional relationship between all inputs and outputs over all semantically feasible paths through the code.
(5) Compliance Analyser compares the mathematical behavior of the code with its formal IL specification to ensure that the code of each procedure:conforms to the functional aspects of its specification (SDR, supported by SDS), respects the state invariants, performs its specified functions without corrupting the computing environment nor data owned by any other module or procedure, conforms to the language in which it is written.

6.10.7 Style. We have not found an example of Sizewell B specification in the literature, so this section exceptionally includes no illustration. The IL text is fed into MALPAS, which constructs a directed graph and associated semantics for the program. For compliance with the MALPAS Compliance Analyzer, the IL specification is written as pre- and postconditions for each procedure, plus optional code assertions. The analyzer determines whether the code meets the specification.

The first part of the Compliance Analysis process is the construction of the mathematical specification from the natural language SDR and SDS. This work ensures both that the interpretation of the existing specifications is correct and that the important functionality and properties are modelled in the mathematical specifications.

6.10.8 Software characteristics. Sizewell B has required the major effort by Westinghouse, NE, the NNC and others to ensure that complex protective functions are implemented accurately in code and that there is a high degree of confidence in this. The quantity of software involved in the PPS is large, but not too large to have been verified meticulously and in its entirety. The PPS software is verified with very demanding reliability requirements (imposed by UK Nuclear Installations Inspectorate (NII)), that attest to its suitability for reactor protection. About 2000 comments have been raised during the verification process (One for 30 lines of code). 40% of them induced minor specification changes or checks, but no major issue was detected.

We didn't find any representation of this semantics in the literature

6.10.9 Project characteristics. The team grew from 15 persons at the beginning of year 1992, to 95 at the end of the project. It took four and a half years to complete the project; The Compliance Analysis was the most important part of the MALPAS analysis of the PPS software and also involved the majority of the effort.

6.10.10 Lessons. The analysis does not prove the safety of the software, but does prove formally that it meets its specifications. It also increased the integrity of the software (through code and documentation modifications that have resulted from detected anomalies) and confidence in its correctness. Five main measures were adopted to ensure its accuracy, consistency and reproducibility: (1) all work was conducted under a defined quality system and in accordance with the requirements of ISO9001[1]; (2) a detailed "standards and procedures" document defined very precisely how the analysis was to be conducted; (3) full recording of all aspects of the analysis was ensured; (4) peer reviews of all analytical work were carried out; (5) strict configuration management of all analytical documents and results was applied, both in hard copy and magnetic media.

The Malpas analysis work conducted in the scope of this project has shown the feasibility of conducting a rigorous retrospective analysis of a large software system. Even if the costs are high in absolute terms, they are low in relation with any potential software malfunction, for example. Conducting rigorous static analysis, including Compliance Analysis, has been then considered to be essential for software controlling systems with the level of criticality and potential failure consequences similar to that of the Sizewell 'B' PPS.

6.11 CoCon

6.11.1 Scope. CoCon is a conference management system, providing a web interface.

6.11.2 Components. Cocon involves five types of stakeholders, each with a specific role. It supports 45 operations, some open to some stakeholders only. They are divided into 5 categories: creation, update, nondestructive update, reading, listing. A conference goes through 7 phases: No-Phase, Setup, Submission, Bidding, Reviewing, Discussion, Notification, Closing.

6.11.3 Verified properties. Confidentiality properties of CoCon are verified through a framework, called verification infrastructure for Bounded-Deducibility (BD) security. It is a general framework for the verification of rich information flow properties of input/output automata. BD security is parameterized by declassification bounds and triggers (in a context of a fixed topology of bounds and triggers). Informally, BD Security can be summarized as follows: If trigger T never holds, then attacker Obs can learn nothing about secrets Sec beyond B. CoCon's verification also involves a form of traceback properties and several safety properties proofs. All of them are described by Isabelle scripts (A zip archive with the Isabelle formalization is available[2]).

6.11.4 Project context. A Conference Management System (CMS) presents confidentiality, integrity and security problems. In addition, a CMS must guarantee the selective availability of information.

[1]https://www.iso.org/standard/16533.html
[2]https://www.andreipopescu.uk/papers/Formal_Scripts_CoCon.zip (accessed in March 2023)

CoCon addresses information flow security problems of realistic web-based systems by interactive theorem proving—using a proof assistant, Isabelle/HOL.

6.11.5 Decisions. The architecture of CoCon follows the paradigm of security by design [24] (the guarantees do not apply to the application layer).

6.11.6 Tool stack. Cocon relies on a three layers stack as follows:

1) Formalization and verification of the kernel of the system in the Isabelle proof assistant.
2) Automatic translation of the formalization obtained in step 1) into Scala.
3) Wrapping the translated program (from 2) in a web application, using trusted components.

6.11.7 Style. To ensure that all these properties are handled, CoCon's kernel is formalized in Isabelle as an executable input/output automaton (extracted from the Isabelle specification to a Scala program using Isabelle's code generator). A state of the CoCon's I/O Automaton stores the lists of registered conference IDs, user IDs and paper IDs. For each ID, the state stores actual conference, user or paper information. For user IDs, the state also stores (hashed) passwords. In the context of a conference, each user is assigned one or more of the roles described by the following Isabelle datatype: `datatypeRole=Chair` PC| AutPaperID | RevPaperIDNat|.

The initial state of the system, *istate* ∈ *State* (Fig. 8), is the one with no conference and a single user, Andrey Voronkov, as SuperChair (the superChair is the first user of the system whose role is to approve new conference requests).

```
istate =
    confIDs = []                    conf = (λ cid.emptyConf)
    userIDs = ["voronkov"]          pass = (λ uid.emptyPass)
    user = (λ uid.emptyUser)        roles = (λ cid uid.[])
    paperIDs = (λ cid.[])           paper = (λ pid.emptyPaper)
    pref = (λ uid pid.NoPref)       voronkov = "voronkov"
    news = (λ cid.[])               phase = (λ cid.noPh)
```

Fig. 8. Specification of the initial state

6.11.8 Software characteristics. CoCon has been used so far to manage the submission and review process of two international conferences, TABLES 2015 and ITP 2016, hosting about 70 and 110 users respectively, consisting of PC members and authors.

6.11.9 Project characteristics. Verification took 3 person-months, including development of the reusable proof infrastructure and automation. Developing the web application around the kernel took longer, much of it done incrementally with the system already up and running. Two master students worked on it (6 person-months); one of them completely changed the implementation.

6.11.10 Lessons. The second application of CoCon evealed a bug that was difficult to catch. This bug made privacy violations possible, which is exactly what CoCon is trying to avoid. In reality, the bug was caused by the web interface and had nothing to do with CoCon's verification. It did highlight the need to verify the API layer and all the trusted components.

The formal guarantees provided in Isabelle must be combined with trust steps applied to the entire system. The verification targets only the system's implementation logic — attacks such as browser-level forging are out of its reach, but are orthogonal issues that should be handled.

7 DISCUSSION AND LESSONS LEARNED

The review of systems in the previous section leads, after an analysis of threats to validity (7.1), to general observations (7.2), lessons drawn by the project members themselves (7.3), insights about the penetration of formal techniques (7.4), and an overall assessment of the results (7.5).

7.1 Threats to validity

The analysis of systems leading to this survey is subject to clear limitations:

- It is not a full Systematic Literature Review, in particular because some of the industrial systems surveyed do not appear in the "literature" in the sense of scholarly journals or conferences. It follows, however, the recommended review protocol for SLRs [45].
- We may have overlooked relevant systems for lack of information. We may for example have missed systems from companies that do not think of publishing their experience.
- Not all application areas are represented; missing, for example, are desktop applications, malware, software for ML and AI and healthcare systems.
- The selected systems were from Europe or the US, and are described in English.
- A selection bias comes from the papers themselves, which usually report successful verification efforts. Note, however, that the Ironclad authors explain that the method used did not succeed in verifying security but made it possible to find a complementary method that did.
- The available information mostly concerned initial system development. It is well known that updates ("maintenance") are a key part of software engineering. It has been difficult to find information on updates and whether they have been verified.
- Contacting people to get information not immediately clear from publications has often been hard (as project members leave the organization or no contact information is available).

7.2 Projects and technology: evolution over time

Fig. 9 presents the timeline of the projects, from 1989 to 2021; note the quasi-constant rate (red line) from 2005 on. Proof-based and auto-active approaches had replaced refinement-based ones by 2007. Both kinds continue to strive, but from 2011 on proof-based overtakes auto-active. Model-checking, although not used as a single technique for verifying code, underlies some of the more complete techniques. Refinement-based systems are rarely used (Roissy Shuttle is one of example).

The mostly frequently used programming languages are Coq and C; 8 projects use Coq as their programming language, 6 of which 6 also use it as their specification languages. The most frequently used specification languages are Coq (11 projects) and Z (4 projects). 7 projects apply annotations in the programs to specify the verified properties. Connections exist between programming and specification languages: a verification using HOL seems to be suitable for systems written in C / SML / OCaml; SPARK code is specified in Z; Isabelle is used for Scala and C.

The most widely used proof engines are Isabelle, Coq, Z3, and HOL4. Correspondingly, the most widely used theoretical framework is Hoare logic.

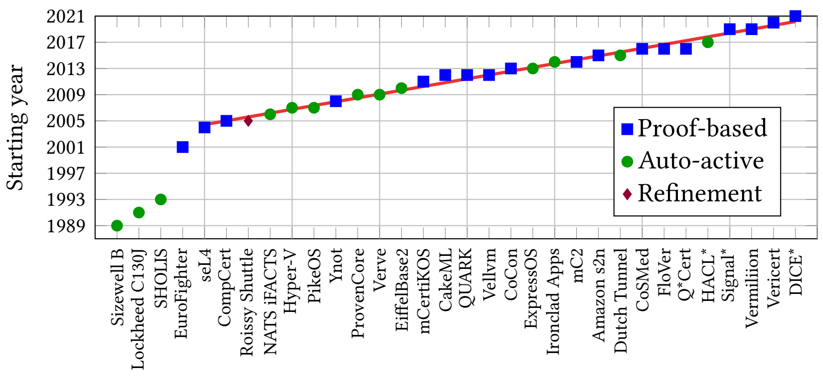


Fig. 9. Verification approaches by year

7.3 Lessons stated by project teams

Some important general lessons come from comments made by project team members themselves. On the positive side, some teams (Lockheed-Martin C130J, NATS iFACTS, EuroFighter Typhoon Aircraft FCS, Dutch Tunnel Control) reported measurable reduction of costs after introducing formal verification into the development process. The NATS iFACTS project actually recommends, as a result, applying formal methods to all phases of the software development life cycle.

Teams also reported limitations. 22 of them had to depend on an unverified code elements, as with the grep and HACL* projects which compile the resulting C code with GCC rather than the verified CompCert compiler, to get more efficient binaries. Unverified elements can include:

- Standard reusable components (in CompCert, addressing parsing, assembling and linking).
- Unverified features of the verification tool itself, as with code extraction in Coq, used by many projects. In fact, the Ironclad Apps project found an actual bug in the verification tools.
- Generated code (such as the output of the Verilog compiler) that requires further processing by trusted tools.
- Program design that makes it difficult to specify and verify properties of interest, as in HACL*.
- Insufficient computing resources to conduct exhaustive verification, as in SHOLIS.
- Trusted but unverified components, as with an API layer in CoCon.
- Reliance on the assumption that future users of the system will not attempt certain malicious actions, as in Ironclad Apps.
- Bugs in the formal specifications themselves. Manual inspection revealed 3 such bugs in Ironclad Apps specification.
- Natural language requirements, as were used in the Roissy Shuttle project.

Such recourse to unverified elements is not universal. Some projects — including CakeML, HACL*, Verve, EuroFighter Typhoon Aircraft FCS and Ynot — chose instead to sacrifice such functionality.

Members from 14 projects reported performance issues:

- Some verified systems perform more slowly than competing non-verified systems. Since verification does not by itself affect performance, one may posit two explanations: some highly efficient algorithms or implementation ideas may raise verification challenges and hence be discarded in favor of less efficient ones; and the effort devoted to verification may mean that less manpower is available for optimizations of the implementation.
- Some derived systems (such as binaries produced by a verified compiler) may perform more slowly than traditional alternatives.
- Verification may slow down the development process; in particular, failed verification attempts take a long time to terminate or do not terminate (and have to be interrupted manually).

In the other direction, the HACL* developers (section 6.3) report that the verification process helped them optimize an existing implementation of the Curve25519 algorithm and prove the optimization correct. The optimized implementation is about 2.2% faster than the formerly fastest one. Optimizations often make the code less readable and thus more likely to contain defects, encouraging developers to be more conservative in optimizing good enough code. Formal verification tools, as they prove correctness of optimizations, can make the developers more determined and confident about using such optimizations, as observed in the HACL* project.

Reports from 7 projects (CompCert, CakeML, EiffelBase2, PikeOS, Ynot, Roissy Shuttle, Ironclad) state that their verification technology would be difficult for an average developer to practice. The Roissy Shuttle reported significant effort invested into validating the formalization of initial natural-language requirements. Even though the validation was performed rigorously, it was still a manual, human-driven process. The researchers were not 100% sure that the formal specification correctly formalized the actual needs and, as noted above, did find incorrect elements.

7.4 Current use of formal verification in industry, and obstacles to further penetration

Many of the verified systems surveyed belong to the category of system software: programs (such as compilers, operating systems and hypervisors, cryptographic and data structure libraries, database management systems) providing a platform for running other software. Others are application software in safety-critical areas: aeronautics, rail, nuclear plants. Non-safety-critical applications include a web browser and few social applications: a conferencing system, a social media platform.

In spite of the potential of today's verification technology and its successes as documented here, formal verification is still not mainstream. The key obstacle is the high entry level: each project needs one or more verification experts. Companies may feel that it will be easier to find many testers than a few verification-savvy developers.

Another concern is the rate of change of modern software systems. It is now common to practice "continuous development", producing new versions once or more daily. In principle every such version requires re-verification. The good news, reported by several projects (seL4, Cocon, Amazon s2n, iFACTS, Dutch tunnel) is that re-verification costs much less than the original verification. seL4 reports that verifying a new and large feature (a new data structure for API calls) cost less than 10% of the initial effort. The re-verification of small changes in iFACTS was completed in seconds, while its initial verification run took three hours.

An alternative to re-verification is reuse of verified elements. Reviewing the projects revealed a number of cases of successful reuse, for example in OS projects (mCertiKos 6.5.10, Hyper-V, PikeOS). Particularly interesting is the reuse by VeriCert of part of the verified CompCert compiler: VeriCert's compilation from C to three-address code was adapted from CompCert's compilation from C to Verilog hardware designs, and did not need to be re-verified. Verification reuse is not, however, as widespread as it could be. It is unclear why Microsoft retired the successful and reusable VCC project tools, initially developed for the verification of Hyper-V and PikeOS.

More generally, projects using continuous development may require continuous verification. Proof-based verification favors this goal: it is possible to develop proofs and code together and keep them in sync. Tools are needed to support such incremental verification processes.

In practice it seems that many projects, once verified, do not continue the audit effort, do not transfer their results to other projects, and do not integrate the results into their tools. A large percentage of the projects initially selected were abandoned immediately or shortly after completion, as these projects, although deployed, were purely for research purposes. Only a few of them, such as Roissy Shuttle and PikeOS, were deployed in public systems for constant use. There seems not to be enough follow-up and scaling-up of even successful results.

We noticed also that some of the verified projects were not accepted by the users even though they have been proven (hence better than their predecessors) mainly due to lack of compatibility of the new verified project with legacy code. An example is ExpressOS.

Applications with potentially complicated internals and simple input-output streams seem to be easiest to verify: they involve no interaction with the (unverified) external world. In addition, systems that do involve many external interactions tend to suffer more from the just discussed issue of frequent change. Perhaps as a result of this property, we did not encounter any application with a formally verified graphical user interface, apart from research prototypes [3].

Human factors are also at play. Most of the systems studied here involved researchers as the key force behind the verification. This approach may not scale well if the demand for correct software grows. The industry would need proof-engineering professionals [47], who in addition to general SE competence possess a strong background in formal reasoning and programming theory. In the projects studied here, however, engineer training does not seem to have been an obstacle. The iFACTS project shows that engineers from different programming backgrounds can become familiar with verification languages and tools. Similarly, the SHOLIS project conducted only a 5-day Z training course and a 10-day SPARK course, for a verification effort of 19 person-years.

7.5 Perspectives for formal verification in industry

7.5.1 A realistic prospect. The preceding subsections (7.3 and 7.4) provide a frank and realistic analysis of the obstacles to generalizing formal verification in industry. It should not, however, lead to a pessimistic assessment, in particular because the projects surveyed include several striking success stories: projects that were closely attuned to market needs and achieved many installations or widespread use; examples include CompCert, Hyper-V, HACL*, the Roissy shuttle and PikeOS.

These successes demonstrate that the idea of software guaranteed correct (in the precise sense analyzed in sections 4.2 to 4.4) is no longer a heroic goal, but a distinct practical possibility.

Unlike testing (to any level of depth), formal verification *guarantees* the absence of bugs of specified kinds, dependent only on the quality of the theory and tools, not on the number of use cases or on whether all were exercised. Advances in verification tools have been enormous, enabling the verification of software with thousands of lines of code [11].

7.5.2 Positive effects of formal verification attempts. While the prime goal of verification for most projects was to eliminate bugs, many had another incentive: achieving certification for applicable industrial standards. For example, the verification projects for microkernels and ProvenCore aim at the Common Criteria EAL7 standard[3], which requires that the system design should be formally verified and tested. The Lockheed-Martin C130J and PikeOS project relied on the DO-178B [4] which helps determine whether the software will perform reliably in an airborne environment. Another potential driver is the needs of product users. In the s2n project, Amazon uses formal analysis to provide customers with concrete information about how it establishes security.

Some projects (Hyper-V, Amazon s2n, CoCon) report that formal verification revealed design defects or subtle bugs not detected by the regular quality assurance process. In the Lockheed-Martin C130J project, some errors that formal analysis could establish immediately, such as conditional initialization errors, might only have emerged after very extensive testing. The HACL* project further suggests that formal verification helps identify and verify potential optimizations.

The entry cost for formal methods may seem prohibitive to some companies. cost can, however, decrease dramatically for subsequent efforts. As noted in 7.4, re-verification can be cheaper by an order of magnitude. Verified software can be reusable and serve as a reliable component in another verified project. While we noted (7.4) that more reuse is needed, we also saw notable successes in this area, such as the reuse of the CompCert compiler and its verification in VeriCert.

8 CONCLUSION

The survey of 32 software systems, with a detailed analysis of 11 here (the rest in the extended version [39]), shows that formally verified software systems use a wide range of methods, tools, specification styles and languages. While no single solution has achieved dominance for verification (let alone for unified verification and development), some approaches have reached a significant level of successful use, including languages such as Z and B for safety critical systems such as nuclear or aeronautics, and Hoare-style proof techniques in these and other areas.

Generalizing verification faces obstacles, of which the main one, per the results of the present study, remains cost, which can be reduced in various ways; for example, training can be inexpensive, and re-verification can cost 90% less than the original effort; in addition, formal verification

[3]https://www.commoncriteriaportal.org
[4]https://www.academia.edu/24446830/SOFTWARE_CONSIDERATIONS_IN_AIRBORNE_SYSTEMS_AND_EQUIPMENT_CERTIFICATION

considerably reduces testing costs. Even the discovery of a few bugs justifies the effort. Few people who have practiced formal verification on a real system would go back.

Acknowledgements. We thank Alexander Naumchev and Aliyu Alege for their help towards an earlier version of this work.

REFERENCES

[1] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases, vol. 8. Addison-Wesley Reading (1995)

[2] Abrial, J.R.: Formal methods: Theory becoming practice. Journal of Universal Computer Science **13**(5), 619–628 (2007)

[3] Adelsberger, S., Setzer, A., Walkingshaw, E.: Declarative GUIs: Simple, consistent, and verified. In: Int. Symp. on Principles and Practice of Declarative Programming. ACM (2018)

[4] Auerbach, J.S., Hirzel, M., Mandel, L., Shinnar, A., Siméon, J.: Handling environments in a nested relat. algebra with combinators and an impl. in a verified query compiler. In: Int. Conf. on Managmt. of Data. pp. 1555–1569. ACM (2017)

[5] Authority, C.A.: SW01—Regulatory objectives for software safety assurance in ATS equipment in part B (generic requirements and guidance) of CAP670—Air traffic services safety requirements (2001)

[6] Badeau, F., Amelot, A.: Using B as a high level programming language in an industrial project: Roissy VAL. In: Int. Conf. of B and Z Users. pp. 334–354. Springer (2005)

[7] Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. Lectures on Runtime Verification: Introductory and Advanced Topics pp. 1–33 (2018)

[8] Bauereiß, T., Gritti, A.P., Popescu, A., Raimondi, F.: Cosmed: A confidentiality-verified social media platform. Journal of Automated Reasoning **61**(1), 113–139 (2018)

[9] Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification–specification is the new bottleneck. arXiv:1211.6186 (2012)

[10] Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M.O., Fox, A.: A verified certificate checker for finite-precision error bounds in Coq and HOL4. In: Formal Methods in Computer Aided Design. pp. 1–10. IEEE (2018)

[11] Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., et al.: Everest: Towards a Verified, Drop-in Replacement of HTTPS. In: Summit on Advances in Programming Languages (2017)

[12] Bhargavan, K., Fournet, C., Kohlweiss, M.: miTLS: Verifying protocol implementations against real-world attacks. IEEE Security Privacy **14**(6), 18–25 (2016)

[13] Blackham, B., Shi, Y., Chattopadhyay, S., Roychoudhury, A., Heiser, G.: Timing analysis of a protected operating system kernel. In: Real-Time Systems Symposium. pp. 339–348. IEEE (2011)

[14] Chapman, R., Schanda, F.: Are we there yet? 20 years of industrial theorem proving with SPARK. In: Int. Conf. on Interactive Theorem Proving. pp. 17–26. Springer (2014)

[15] Chapman, R., White, N.: Industrial experience with agile in high-integrity software development. In: Safety-Critical Systems Symposium. pp. 2–4 (2016)

[16] Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., et al.: Continuous formal verif. of Amazon s2n. In: Int. Conf. on Comp.-Aided Verif. pp. 430–446. Springer (2018)

[17] Common vulnerabilities and exposures (2019), http://cve.mitre.org/

[18] List of companies using formal verification methods in soft. eng. (2021), https://github.com/ligurio/practical-fm

[19] CompCert Webpage (2021), https://compcert.org/

[20] Coq Proof Assistant (2021), https://coq.inria.fr/

[21] Costanzo, D., Shao, Z., Gu, R.: End-to-end verification of information-flow security for C and Assembly programs. SIGPLAN Notices **51**(6), 648–664 (2016)

[22] Croxford, M., Sutton, J.: Breaking through the V and V bottleneck. In: International Eurospace-Ada-Europe Symposium. pp. 344–354. Springer (1995)

[23] De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)

[24] Deogun, D., Johnsson, D., Sawano, D.: Secure By Design. Manning (2019)

[25] Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. Eng. dependable soft. syst. pp. 141–175 (2013)

[26] Fisher, K., Launchbury, J., Richards, R.: The HACMS program: using formal methods to eliminate exploitable bugs. Phil. Trans. of the Royal Society A: Mathematical, Physical and Engineering Sciences **375**(2104) (2017)

[27] Flover: A certificate checker for roundoff error bounds (2021), https://gitlab.mpi-sws.org/AVA/FloVer

[28] Floyd, R.W.: Assigning meanings to programs. In: Program Verification, pp. 65–81. Springer (1993)

[29] Fonseca, P., Zhang, K., Wang, X., Krishnamurthy, A.: An empirical study on the correctness of formally verified distributed systems. In: European Conference on Computer Systems. p. 328–343. ACM (2017)

[30] Gernot Heiser: The seL4 microkernel - an introduction. Tech. rep., The seL4 Foundation (2020)

[31] Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X., Weng, S.C., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. ACM SIGPLAN Notices **50**(1), 595–608 (2015)

[32] Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertikOS: An extensible architecture for building certified concurrent OS kernels. In: USENIX Symp. on OS Design and Implementation. pp. 653–669 (2016)
[33] Guéneau, A., Myreen, M.O., Kumar, R., Norrish, M.: Verified characteristic formulae for CakeML. In: European Symposium on Programming. pp. 584–610. Springer (2017)
[34] Hacl*: A high-assurance cryptographic library (2021), https://github.com/project-everest/hacl-star
[35] Harrison, J., Urban, J., Wiedijk, F.: History of Interactive Theorem Proving, vol. 9, pp. 135–214 (2014)
[36] Herklotz, Y., Pollard, J.D., Ramanathan, N., Wickerson, J.: Formal verification of high-level synthesis. ACM on Programming Languages **5**, 1–30 (2021)
[37] Hoang, T.S.: An Introduction to the Event-B Modelling Method, pp. 211–236 (2013)
[38] HOL interactive theorem prover (2021), https://hol-theorem-prover.org/
[39] Huang, L., Ebersold, S., Kogtenkov, A., Naumchev, A., Meyer, B.: Lessons from formally verified deployed software systems (extended version of the present paper). arXiv (2023), https://arxiv.org/abs/2301.02206
[40] Isabelle (2021), https://isabelle.in.tum.de/
[41] Jang, D.: Language-based Security for Web Browsers. University of California, San Diego (2014)
[42] Jang, D., Tatlock, Z., Lerner, S.: Establishing browser security guarantees through formal shim verification. In: USENIX Security Symposium. pp. 113–128 (2012)
[43] Kanav, S., Lammich, P., Popescu, A.: A conference management system with verified document confidentiality. In: Int. Conf. on Computer-Aided Verification. pp. 167–183. Springer (2014)
[44] Kästner, D., Barrho, J., Wünsche, U., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., Blazy, S.: Compcert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In: European Congress Embedded Real-Time Software and Systems. pp. 1–9 (2018)
[45] Keele, S., et al.: Guidelines for performing systematic literature reviews in soft. eng. Tech. rep., Citeseer (2007)
[46] King, S., Hammond, J., Chapman, R., Pryor, A.: The value of verification: Positive experience of industrial proof. In: Int. Symp. on Formal Methods. pp. 1527–1545. Springer (1999)
[47] Klein, G.: Proof engineering considered essential. In: Int. Symp. on Formal Methods. pp. 16–21. Springer (2014)
[48] Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. Transactions on Computer Systems **32**(1), 1–70 (2014)
[49] Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified impl. of ML. SIGPLAN Not. **49**(1), 179–191 (2014)
[50] Lasser, S., Casinghino, C., Fisher, K., Roux, C.: A verified LL(1) parser generator. In: Int. Conf. on Interactive Theorem Proving. vol. 141, pp. 1–18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
[51] Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V hypervisor with VCC. In: Int. Symp. on Formal Methods. pp. 806–809. Springer (2009)
[52] Leino, K.R.M.: This is Boogie 2. Manuscript KRML (2008)
[53] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370. Springer (2010)
[54] Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009)
[55] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert – a formally verified optimizing compiler. In: Embedded Real Time Software and Systems (2016)
[56] Lescuyer, S.: ProvenCore: Towards a verified isolation micro-kernel. In: MILS@ HiPEAC (2015)
[57] Letouzey, P.: A new extraction for Coq. In: Int. Work. on Types for Proofs and Programs. pp. 200–219. Springer (2002)
[58] Lmbench - tools for performance analysis (2013), http://lmbench.sourceforge.net/
[59] Mai, H., Pek, E., Xue, H., King, S.T., Madhusudan, P.: Verifying security invariants in ExpressOS. In: Int. Conf. on Architectural Support for Programming Languages and Operating Systems. p. 293–304. ACM (2013)
[60] Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 237–248 (2010)
[61] mCertiKOS Hypervisor (2021), https://flint.cs.yale.edu/certikos/mcertikos.html#mcertikos
[62] Mi, Z., Li, D., Yang, Z., Wang, X., Chen, H.: Skybridge: Fast and secure inter-process communication for microkernels. In: EuroSys Conference. pp. 1–15 (2019)
[63] How technology is transforming air traffic management (2013), https://nats.aero/blog/2013/07/how-technology-is-transforming-air-traffic-management/
[64] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer (2002)
[65] Oortwijn, W., Huisman, M.: Formal verification of an industrial safety-critical traffic tunnel control system. In: Integrated Formal Methods. pp. 418–436. Springer (2019)
[66] O'Halloran, C.: Automated verification of code autom. generated from Simulink. Aut. Soft. Eng. **20**(2), 237–264 (2013)
[67] Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. In: Int. Symp. on Formal Methods. pp. 414–434. Springer (2015)

[68] Popescu, A., Lammich, P., Hou, P.: Cocon: A conference management system with formally verified document confidentiality. Journal of Automated Reasoning **65**(2), 321–356 (2021)
[69] Protzenko, J., Beurdouche, B., Merigoux, D., Bhargavan, K.: Formally verified cryptographic web applications in WebAssembly. In: IEEE Symposium on Security and Privacy. pp. 1256–1274 (2019)
[70] Quark: A web browser with a formally verified kernel (2021), https://github.com/Conservatory/quark
[71] Formally Verified Systems Questionnaire (2021), https://bit.ly/2LMxbZB
[72] Rescorla, E., Dierks, T.: The transport layer security (TLS) protocol version 1.3 (2018)
[73] Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z.: Qed at large: A survey of engineering of formally verified software. Foundations and Trends® in Programming Languages **5**(2-3), 102–281 (2019)
[74] s2n github repository (2021), https://github.com/aws/s2n-tls
[75] Sánchez, C., et al.: A survey of challenges for runtime verification from advanced application domains (beyond software). Form Methods Syst Des **54**(3), 279–335 (2019)
[76] Sewell, T., Kam, F., Heiser, G.: High-assurance timing analysis for a high-assurance real-time operating system. Real-Time Systems **53**(5), 812–853 (2017)
[77] Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified OS kernel. In: ACM SIGPLAN conference on Programming language design and implementation. pp. 471–482 (2013)
[78] Song, Y., Cho, M., Kim, D., Kim, Y., Kang, J., Hur, C.K.: CompCertM: CompCert with C-Assembly linking and lightweight modular verification. ACM Programming Languages **4** (2019)
[79] Stich, R.: Clearance of flight control laws for carefree handling of advanced fighter aircraft. In: Optimization Based Clearance of Flight Control Laws, pp. 421–442. Springer (2012)
[80] Tan, Y.K., Owens, S., Kumar, R.: A verified type system for CakeML. In: Symposium on the Implementation and Application of Functional Programming Languages. pp. 1–12 (2015)
[81] Tao, Z., Rastogi, A., Gupta, N., Vaswani, K., Thakur, A.V.: DICE*: A formally verified implementation of DICE measured boot. In: USENIX Security Symposium. pp. 1091–1107. USENIX Association (2021)
[82] Ter Beek, M.H., Chapman, R., Cleaveland, R., Garavel, H., Gu, R., ter Horst, I., Keiren, J.J.A., Lecomte, T., Leuschel, M., Rozier, K.Y., Sampaio, A., Seceleanu, C., Thomas, M., Willemse, T.A.C., Zhang, L.: Formal methods in industry. Formal Aspects of Computing (2024)
[83] Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCertTSO: A verified compiler for relaxed-memory concurrency. Journal of the ACM **60**(3) (2013)
[84] Ward, N.: The rigorous retrospective static analysis of the Sizewell 'B' primary protection system software. In: Int. Conf. on Computer Safety, Reliability and Security. pp. 171–181. Springer (1993)
[85] White, N., Matthews, S., Chapman, R.: Formal verification: will the seedling ever flower? Phil. Trans. of the Royal Society A: Mathematical, Physical and Engineering Sciences **375**(2104) (2017)
[86] Wikipedia: Common Criteria (2022), https://en.wikipedia.org/wiki/Common_Criteria
[87] Wikipedia: Kernel (operating system) (2022), https://en.wikipedia.org/wiki/Kernel_(operating_system)
[88] Wikipedia: L4 microkernel family (2022), https://en.wikipedia.org/wiki/L4_microkernel_family
[89] Wikipedia: Principle of least privilege (2022), https://en.wikipedia.org/wiki/Principle_of_least_privilege
[90] Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM Computing Surveys **41**(4) (2009)
[91] Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: SIGPLAN Conference on Programming Language Design and Implementation. pp. 99–110. ACM (2010)
[92] Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: SIGPLAN Conference on Programming Language Design and Implementation. pp. 283–294. ACM (2011)
[93] Zhang, F., Niu, W., et al.: A survey on formal specification and verification of system-level achievements in industrial circles. Academic Journal of Computing & Information Science **2**(1) (2019)
[94] Zhang, Q., Zhuo, D., Wilcox, J.: A survey of formal verification approaches for practical systems https://courses.cs.washington.edu/courses/cse551/15sp/projects/qz2013-danyangz-jrw12.pdf
[95] Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: POPL. pp. 427–440. ACM (2012)
[96] Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formal verification of SSA-based optimizations for LLVM. In: SIGPLAN conference on Programming Language Design and Implementation. pp. 175–186. ACM (2013)
[97] Zhao, J., Zdancewic, S.: Mechanized verification of computing dominators for formalizing compilers. In: Int. Conf. on Certified Programs and Proofs. pp. 27–42. Springer (2012)
[98] Zhao, Y., Sanán, D., Zhang, F., Liu, Y.: High-assurance separation kernels: a survey. arXiv:1701.01535 (2017)
[99] Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: SIGSAC Conference on Computer and Communications Security. pp. 1789–1806. ACM (2017)