Title
The Current State of Software Assurance Tools and Techniques

Author
Peisert, Sean

Publication Date
2022-10-24

Peer reviewed

BERKELEY LAB

The Current State of Software Assurance Tools and Techniques

Sean Peisert
Lawrence Berkeley National Laboratory*
sppeisert@lbl.gov

October 24, 2022

Contents

List of Figures

*This work is supported by the "AOSCSWAP: A Study of Academic, Open Source, and COTS Software Assurance Products" project, sponsored by the Department of Homeland Security's Science and Technology Directorate.

1 Introduction

"*Assurance* is confidence that an entity meets its requirements, based on specific evidence provided by the application of assurance techniques" [1]. *Security assurance* is confidence that an entity meets its *security requirements*. *Software assurance* is confidence that software meets its requirements, including both functional, security, and otherwise.

Confidence is, of course, a spectrum, not a binary property, and therefore, assurance is a spectrum as well. There are many paths to achieving degrees of assurance, and the elements of those paths take place throughout the *software development life cycle (SDLC)*, including during requirements specification, design, implementation, operation, and maintenance. Ideally software assurance will in fact involve all of these elements, and there will be continuous cycles and feedback loops between each one. Of course, this process often does not always happen, as software development has often shifted into *Agile Software Development* or *DevOps*, which can sometimes, though not always, be seen as "write now, fix later."

Regardless of the approach used, software tools can often assist in assurance. For example, software analysis tools can help to identify implementation bugs that might lead to crashes, incorrect results, and exploitable vulnerabilities. Tools cannot find every security problem in software, but they can help increase the security assurance of software.

Tools come in a variety of forms, and operate on different things and in different ways. For example, some tools operate on source code, others operate on binaries, and others operate during execution. Still others don't operate directly on source code at all but rather operate on mathematical logic during the specification phase.

In this document, we examine numerous different categories of these tools and discuss what they do, give examples of the most popular tools in each category, and, in some cases their users. We also discuss what their their various benefits and drawbacks are. We conclude with an analysis and summary of the strengths, weakesses, opportunities, and threats for tool-based approaches to software assurance.

2 Dependency Analysis

Dependency analysis tools identify which libraries, packages, or modules are used by a program and then looks up in databases (such as the National Vulnerability Database) to see if there are previously disclosed vulnerabilities in the versions of the libraries, packages, or modules used by the code. In the current parlance of discussions about supply chain security issues, this list of dependencies can be thought of as a *software bill of materials (SBoM)*. Dependency tools have the simplest output of any of the tools that this document discusses: just a list of dependencies that contain concerns identified in the databases that are looked up. The easiest dependency analysis tools to use are typically integrated with the code repository as they are run automatically. *Dependabot*, integrated with GitHub, is one such example. It is very commonly used throughout software development community, and is often seen as being as effective as separately-purchased commercial tools.

3 Static Code Analysis

Static analysis tools (often called "SAST") scan a program's source, byte, or binary code in the search of stylistic problems or mistakes that might be weaknesses. Static analysis tools can include hundreds of different kinds of reports, often with quite technical descriptions, so can be intimidating

on first use. One of the biggest complaints about static code analysis tools is that they generate false positives. Since no widely used tool is currently *sound*, the reports produced by any tool, open source or commercial, can be an over approximation of the actual problems to be found. Such "noise" in the results can frustrate programmers on their initial use of a tool, especially on code bases where no tool has been previously used. Static code analysis is widely run throughout software development processes.

Static Stylist Analysis Tools can help enforce an organization's coding standard to make coding more understandable, avoid error prone practices, and to give the code a more uniform look.

Static Code Analysis Tools can help detect a wide variety of weaknesses. These weaknesses found by such a tool include buffer overruns, injections, cross site scripting, cross site request forgery, memory leaks, improper input validation, path traversal errors, hard coded credentials, serialization errors, and many others. Such tools cannot detect all occurrences of weaknesses, but they can find many would otherwise go unnoticed by the programmer. These tools typically require that the code compiles but not necessarily executes. As these tools report potential problems by looking at the code, they are subject to false positives.

Some static analysis tools include both stylistic and code analysis features.

There are huge numbers of popular static analysis tools, including Clang,[1] Synopsys' Coverity,[2] CPAchecker,[3]. Facebook's Infer [2],[4], Google's Error Prone,[5] and even Lint.

Coverity has been used by the software on CERN's LHC and NASA JPL's Mars rover *Curiosity*.

Some static code analysis tools use *abstract interpretation* to enable *soundness* (no false negatives) and to generate invariants. Abstract interpretation involves the partial execution of a program to derive insight about the program's control flow and data flow. It does this without performing all calculations, and as a result, produces only an approximation, as some precision is lost. However, it over-approximates the possible state space as a result. In this way, while abstract interpretation achieves soundness, it cannot be guaranteed to be *correct* (it may give false positives).

Examples of static code analysis tools that use abstract interpretation include Inria's Astrée,[6] Inria's Frama-C [3],[7] and MathWorks' Polyspace.[8]

All three tools have seen commercial and government use – notably Astrée has been used by Airbus for the A340 and A380 fly-by-wire system, and also for the automatic docking software of the European Space Agency's Jules Vernes Automated Transfer Vehicle (ATV), and Polyspace has been used by Nissan, Alenia Aermacchi, and Miracor.

Infer [2] is a somewhat special case static code analysis tool in that it combines elements of separation logic, a theorem prover that leverages inference to guess assumptions (to enable automation). It also performs *continuous reasoning* in a way that enables it to analyzes diffs (differences between source code versions) when a change occurs without having to re-analyze the entire codebase, thus improving speed dramatically.

Error Prone is a tool for Java that automatically identifies mistakes in the IDE (Integrated Development Environment) while the developer is typing and also automatically produces suggested fixes.

[1]Clang (built into LLVM): `https://clang.llvm.org/`
[2]Coverity: `http://synopsys.com/software-integrity.html`
[3]CPAchecker: `http://cpachecker.sosy-lab.org`
[4]Infer: `https://fbinfer.com`
[5]Error Prone: https://errorprone.info
[6]Astrée: `https://www.astree.ens.fr`
[7]Frama-C: `https://www.frama-c.com`
[8]Polyspace: `https://www.mathworks.com/products/polyspace.html`

4 Symbolic Execution

Symbolic execution [4] is the automated process of following the execution of every conditional branch of a program by assuming symbolic inputs rather than actual user inputs. In the process, symbolic checking can also determine which inputs would cause each conditional branch to actually execute. Symbolic execution has elements of both dynamic static dode analysis but is typically thought of as distinct. On one hand, it begins with source code and must be able to compile it, but it also executes the program symbolically. Symbolic execution is currently seen as a middle ground between actually executing a program and using traditional static analysis by giving both better coverage and accuracy. However, symbolic execution has a similar problem as model checking with state space explosion in large programs, and so with large programs, code coverage must either be limited or constrained with heuristics. Related to this it will under-approximate the set of possible states.

Concolic testing is a portmanteau of *concrete* and *symbolic* and is intended to indicate a hybrid analysis approach that leverage both concrete and symbolic approaches. The approach was developed as a way of avoiding the issue of a program making system calls or calling external library functions.

Abstract interpretation and symbolic execution are related on the surface in that both seek to abstaractly explore the state space of programs. However, the constraints surrounding their use, including over-approximation (abstract interpretation) vs. under-approximation (symbolic execution) and the process of compiling vs. actually executing make their strengths and weaknesses distinct.

Some commonly used symbolic execution tools include UIUC's KLEE,[9] ForAllSecure's Mayhem,[10] and University of Washington's Rosette.[11]

5 Model Checking

Model checking [5] is an automated process that checks whether a certain finite state model of a system, typically written in temporal logic, adheres to a set of specified safety and liveness properties associated with the model. Where the properties do not hold, and a violation occurs, many model checkers produce counterexamples.

The advantage of static analysis over model checking is higher accuracy. However, model checking carries at least two problems. The first is the state space explosion problem. The second is the issue of undecidability [6] in that the model checker cannot know if any arbitrary program will ever terminate. *Bounded model checkers* can help address the latter issue at the cost of limiting code coverage. On the flip side, model checking looks at a somewhat idealized version of a formal model whereas static analysis, where code is involved (rather than a binary) actually requires compilation. Further, the while the formal model can be augmented with descriptive elements of the surrounding environment, these are typically simplified in comparison to the raw messiness associated with a typical computing system [7]. As a result, while model checking should theoretically be both more sound and more complete than static analysis, this is not always the case.

Model checking can be combined with static code analysis by using static analysis to extract a finite state skeleton from the program and then model checking the result. There can be challenges to this approach, however, since model checking is fully automatic, due to the ambiguities associated

[9] KLEE: `https://klee.github.io`

[10] Mayhem: `https://forallsecure.com`

[11] Rosette: `https://emina.github.io/rosette/`

with high-level programming languages (in contrast to temporal logic).

Popular model checking tools include CMU's CBMC,[12] SoSy-Lab's CPAchecker,[13] NASA's Java Pathfinder,[14] Fondazione Bruno Kessler's NuSMV,[15] P,[16] University of Birmingham and Oxford's PRISM,[17] SPIN,[18] Leslie Lamport's TLA$^+$,[19] and Uppsala and Alborg Univerities' UPPAL.[20]

Sometimes static analysis and model checking are even combined in the same tool. Examples of this include CPAchecker (which recognizes C) and Java Pathfinder (Java).

6 Dynamic Analysis

Dynamic analysis tools (often called "DAST") monitor a program's execution to detect execution-time errors. Dynamic code analysis tools test programs while they are running. They typically instrument a program and then monitor a program's execution to detect if it has any execution errors such as an array reference or pointer access out of bounds. These tools require that a program is able to run and that they have reasonable test inputs. Many such tools are built into compiler frameworks, including the various memory safety detectors built into LLVM, for example. However, some might require that the program be built with a special compiler, such as HCL AppScan,[21] or Parasoft Insure++.[22]

Dynamic testing is a widely performed method for enabling useful unit testing and integration testing, and also has the benefit of testing not just the source code or the binary but the actual binary as it interoperates with its environment. As a result, certain details that may not show up earlier in the process because they are introduced during compilation or through environmental interaction, such as via dynamic linked libraries, will only show up at runtime. At the same time, dynamic analysis requires that sufficient test inputs are generated and that as much code is covered as possible, which may otherwise be limited. Performing static analysis or symbolic execution prior to dynamic analysis can help identify and guide code coverage to help provide the best of both worlds. Dynamic analysis tools can also have a downside that they can produce similarly intimidating reports as static analysis tools. [8]

7 Fuzzing

Fuzz testing or *fuzzing* is an approach to feeding random inputs to program during execution. The term *fuzz* was coined by Barton Miller in 1988 [9] and later elaborated on [10].

Approaches leveraged by fuzz testing tools vary across a number of spectra, including the degree of involvement of a human tester, the availability of the original source code vs. just a binary, the degree of code coverage that can be tested, and the degree of advanced automation techniques that are used for testing (at the cost of time and CPU to perform the tests).

At one end of the spectrum, *black box testing* uses purely random inputs, involves no guidance from a human tester, and use take the least time to ron. If a program hangs or crashers, then a

[12]CMBC: `https://www.cprover.org/cbmc/`
[13]CPAchecker: `http://cpachecker.sosy-lab.org`
[14]Java Pathfinder: `https://github.com/javapathfinder/`
[15]NuSMV: `http://nusmv.fbk.eu/`
[16]P: `https://p-org.github.io/P/`
[17]PRISM: `http://www.prismmodelchecker.org/`
[18]SPIN: `http://spinroot.com/`
[19]TLA$^+$: `https://lamport.azurewebsites.net/tla/tla.html`
[20]UPPAL: `http://www.uppaal.org`
[21]AppScan: `https://www.hcltechsw.com/wps/portal/products/appscan`
[22]Insure++: `https://www.parasoft.com/products/insure`

bug has been found. On the other hand, black box testers have lower code coverage and typically provide less insight than other fuzz testing approaches. Examples of classic fuzz testing tools like this that are still in use include Miller's fuzz[23]

A the other end of the spectrum is *white box testing* that leverages program analysis techniques such as static analysis or symbolic execution to identify the most important paths to examine to improve code coverage, to understand the most effective sets of inputs to use for testing. However the time it takes to perform white box fuzz testing can be very high. Microsoft's SAGE [11] is an example of a white box fuzz tester.

In the middle is of course *grey box testing*. Rather than program analysis, grey box testers use lightweight program instrumentation of either the source code or the binary to obtain information about a program's basic block transitions while a program is executing. Grey box testers have lower performance overhead than white box testers and increased code coverage than black box testers. They are, based on current computational power, often seen as an efficient middle ground between the two extremes for detecting vulnerabilities and other bugs. The most common examples of grey box testing tools include American Fuzzy Lop plus plus (AFL++),[24] Honggfuzz,[25] and libFuzzer.[26] Code Intelligence's Jazzer[27] is also targeted at fuzzing Java programs.

Grey box fuzzing tools are widely used in software development. Apple fuzz tests all of its products. Google's OSS-Fuzz,[28] developed in 2016, is widely used for continuous fuzzing of several security-critical open-source projects and is also used extensively for internal testing at Google. It is the backend for Google's ClusterFuzz,[29] which is a cloud-based fuzzing infrastructure used to test all of Google's products. Microsoft's OneFuzz is a self-hosted fuzzing service platform and is also used by Microsoft itself on all its products.[30]

Grey box testing has also had numerous noteworthy successes, as well. AFL detected the Shellshock bug in the UNIX Bash shell in 2014, and also the Heartbleed bug in 2015. ForAllSecure's Mayhem[31] — mentioned earlier under symbolic execution, because it is a fuzzer that leverages symbolic execution to enhance code coverage — famously won the DARPA Cyber Grand Challenge in 2016.[32] It is also becoming widely adopted as a commercial tool in industry.

There is no "right answer" for fuzz testing except that it is generally seen as valuable. In many cases, black box testing may well be "enough." [12] On the other hand, where time is available, as demonstrated by the widespread use throughout the commercial software development industry, grey box testing has significant advantages.

Fuzz testing tools are an active area of research and numerous other lists of academic tools in development exist.[33]

8 Theorem Proving and Proof Assistants

Theorem proving is the process of developing a mathematical proof that a program that has been developed adheres to a specification [13]. Theorem proving has the advantage over other techniques

[23] fuzz: `http://pages.cs.wisc.edu/~bart/fuzz/`
[24] AFL++: `https://github.com/AFLplusplus/AFLplusplus`
[25] Honggfuzz: `https://github.com/google/honggfuzz`
[26] libFuzzer: `https://llvm.org/docs/LibFuzzer.html`
[27] Jazzer: `https://github.com/CodeIntelligenceTesting/jazzer`
[28] OSS-Fuzz: `https://google.github.io/oss-fuzz/`
[29] Clusterfuzz: `https://github.com/google/clusterfuzz`
[30] OneFuzz: `https://www.microsoft.com/en-us/research/project/project-onefuzz/`
[31] Mayhem: `https://forallsecure.com/mayhem-for-api`
[32] Cyber Grand Challenge: `https://www.darpa.mil/about-us/timeline/cyber-grand-challenge`
[33] Academic Tool List: `https://github.com/cpuu/awesome-fuzzing`

in that it is proving *program correctness* rather than lightweight formal methods like model checking that look for safety or liveness properties, or the presence of specific stylistic errors or bugs, like static code analysis [14].

Why does Amazon use proof assistants for formal verification?

> Byron Cook: "If someone comes along and says, 'for very cheaply, I can tell you this thing does not happen,' [they say], 'you have me.' But, if you just say, 'we're going to find 30% more bugs,' [they say], '[great], we should find more bugs. But now I've found 30% more bugs. Do I know anything more?' Not really, from a leader's perspective, and they are the ones that pay the bills."
>
> Zhendong Su: "But making such a statement that you can say something is 100% true is very, very difficult."
>
> Byron Cook: "Yes, oh yes, but that's why we get the big money. I hear that from time to time people say, 'but it's so hard to be sound,' I say, 'go do testing. There are plenty of people doing testing.' But if you want to fly under the automated reasoning proof flag, then strap in, here we go." [15]

Examples of tools that have been formally verified using proof assistants include the seL4 separation kernel, [16, 17] the CompCert C compiler [18], and Cogent [19].

The use of theorem provers in software development is still rare. However, Amazon has recently particularly dramatically increased its use. While Amazon originally used TLA$^+$ [20, 21] for model checking, it has now expanded its use of formal methods to include tools across the spectrum up to and including theorem provers:

Tools that asist with theorem proving generally fall into categories of being *fully automated* or *semi-automated* (or *interactive*) [22, 23]. Semi-automated approches typically require substantially more expertise in understanding and writing logical, mathematical proofs, but at the same time, can offer the increased power that comes with not having to rely on automation to fully understand everything. The additional expertise provided by a human can enable a semi-automated *proof assistant* to potentially prove a wider range of specifications correct.

In either case, theorem proving first requires developing a mathematical specification. In some cases, such as using Galois's SAW, discussed earlier, it can be possible to derive a specification from source code. However, this has at least two downsides. The first is that high-level programming languages can be ambiguous and so need to be made unambiguous. The second is that in doing so, one would really only be proving that a program does what it does rather than independently comparing a specification to a program.

Amazon Web Service's Byron Cook has indicated:

> "External tools that we use include Boogie, Coq, CBMC, CVC4, Dafny, HOL-light, Infer, OpenJML, SAW, SMACK, Souffle, TLA$^+$, VCC, and Z3." [24]

Fully automated solvers include Alt-Ergo,[34] University of Manchester's Vampire,[35] Microsoft's Z3.[36] These tools can be extremely valuable but can have more limited use than interactive proof assistants due to the limits of fully automated reasoning.

[34] Alt-Ergo: `https://alt-ergo.ocamlpro.com`

[35] Vampire: `http://vprover.github.io/`

[36] Z3: `https://github.com/Z3Prover`

Commonly used interactive proof assistants include Boyer, Moore, and Kaufmann's ACL2,[37] Chalmers and Gothenburg Universities' Agda,[38] Inria's Coq,[39] CVC4,[40] John Harrison's HOL Light,[41] Cambridge and TUM's Isabelle,[42] Microsoft's Lean,[43]

A number of approaches exist to make formal verification easier than writing in specialized higher-order logic (see Section 8) by leveraging limited languages or other specialized approaches to translate source code into formal models that can be formally verified. For example, Boogie[44] is programming language and also a tool that takes a model written in the Boogie language as input and generates conditions to pass to an SMT solver to verify. Systems that can support inputs into the Boogie tool other than the Boogie language itself include Dafny,[45] Microsoft's VCC.[46] and SMACK,[47] which can also translate from the LLVM's intermediate representation to the Boogie intermediate language.

In addition, the Software Analysis Workbench (SAW)[48] enables formal verification of properties C, Java, and Cryptol source code using automated SAT and SMT solvers. The Frama-C WP ("Weakest Precondition") plugin[49] enables allows proving that ACSL annotations in C programs. OpenJML[50] is a verification tool for Java that to verify annotations of specifications of Java programs. Oracle's Systematic, Ontological, Undiscovered Fact Finding Logic Engine (Soufflé)[51] is a programming language that enables static analysis of source code written in Java. Liquid Haskell[52] is enables reasoning and verification of Haskell programs.

Certified Abstraction Layers,[53] RefinedC,[54] and the Verified Software Toolchain (VST),[55] are all tools that provide support for verifying C programs with Coq specifications. BedRock Systems' BriCk[56] is a tool for extracting C++ programs for verification with Coq. hs-to-coq[57] [25] provides similar functionality for Haskell.

9 Analysis and Summary

9.1 Strengths

Software assurance tooling is improving in many places. For example, dependency analysis is now built into tools such as GitHub, making automation as part of a commit a trivial step. Grey-box fuzz testing and symbolic execution (and their combination) have both made amazing strides

[37] ACL2: `http://www.cs.utexas.edu/users/moore/acl2`
[38] Agda: `https://wiki.portal.chalmers.se/agda/`
[39] Coq: `https://coq.inria.fr/`
[40] CVC5: `https://cvc5.github.io`
[41] HOL Light: `https://www.cl.cam.ac.uk/~jrh13/hol-light/`
[42] Isabelle: `https://leanprover.github.io`
[43] Lean: `https://leanprover.github.io`
[44] Boogie: `https://github.com/boogie-org/boogie`
[45] Dafny: `https://github.com/dafny-lang/dafny`
[46] VCC: `https://github.com/microsoft/vcc`
[47] SMACK: `https://smackers.github.io`
[48] SAW: `https://saw.galois.com`
[49] Frama-C/WP: `https://frama-c.com/fc-plugins/wp.html`
[50] OpenJML: `https://www.openjml.org`
[51] Souffle: `https://souffle-lang.github.io`
[52] Liquid Haskell: `https://ucsd-progsys.github.io/liquidhaskell-blog/`
[53] `https://flint.cs.yale.edu/flint/publications/dscal.html`
[54] RefinedC: `https://plv.mpi-sws.org/refinedc/`
[55] Verified Software Toolchain: `https://vst.cs.princeton.edu`
[56] BriCk: `https://github.com/bedrocksystems/BRiCk`
[57] hs-to-coq: `https://hs-to-coq.readthedocs.io/en/latest/`

forward, both in terms of algorithmic improvements, as well as overall capability improvements due to processing power and memory. Perhaps most importantly, Rust has emerged as a type and memory safe language, and thus when used without its "unsafe" abilities provides proven security properties to developers.

At the most rigorous end of the spectrum, numerous success stories for formal verification also show its value. Among the systems deployed in practice include AWS LibCrypto (continuous formal verification of cryptographic primitives), AWS's s2n-tls (formally verified TLS implementation), Google's BoringSSL (formally verified elements of SSL implementation), Google's OpenTitan (formally verified cryptographic hardware elements of silicon root of trust), and Microsoft's Project Everest (formally verified communication protocols, including the EverCrypt formally verified cryptographic library and the EverParse generator for verified parsers and serializers). The demonstrated success of the seL4 formally verified separation kernel in the HACMS competitions is also reaching near-term planned deployment by Apple, NASA, and Sandia, among others. These demonstrations are of significant importance to the community: verified SSL/TLS libraries means never having a "heartbleed" vulnerability again, and in general, verified software means actually knowing what your software does and doesn't do. Further, as AWS has demonstrated, formal verification lowers the cost of maintenance over time.

9.2 Weaknesses

The ideal software assurance tools are *sound* (no false positives) and *complete* (no false negatives) for a rich and expressive set of functional and security properties, with minimal manual additional work by developers beyond existing development approaches. To this end, in an ideal world, perhaps software engineers would write specifications and programs in Coq and prove them correct all in one environment. However, most software engineers will never write functional or security specifications at all let alone ones in Coq, or do formal proofs. Nor are software engineers typically even realistically going to program in functional languages that would make proving easier. On the other hand, this will leave a gap, because the tools that will remain available to software engineers are unlikely to support developing software – particularly certain low-level software, such as compilers, drivers, and other operating system elements, that carry with them extremely intricate requirements relating to memory, typing, interaction with hardware, etc...

Even among the tools providing the lowest level of assurance such as static analyzers, unless such tools are used from the outset, they will continue to give so many false positives that they will be seen as a burden rather than an asset.

Thus our current reality is that of having many gaps. If we are to continue to make progress in software assurance, then surely the future is programming languages that have safety properties (like type and memory safety) by design; programming languages that make it easier to prove functional and security correctness, such as contract languages that make sense to programmers and support automation, even in the face of expressive specifications; and robust tools for those languages and others that support lifting models from source code as well as extracting code from higher-order logic back to code. This latter point is key. For example, Rust provides type and memory safe properties out of the box as long as "unsafe" elements are not used, but on the other hand very little tooling around verification has been built for Rust, nor verified compilers. In contrast C, has the downside of being unsafe out of the box, but has a robust set of tooling around it. Of course, that tooling must be properly used to be effective.

9.3 Opportunities

In many ways the biggest challenge to software assurance tools isn't that there is a technical gap but that there is a social gap in using them at all. This social gap is unsurprising in the event that a developer first runs such tools after several weeks, months, or years of development work only to see hundreds or thousands of issues — many of which end up being false positives — flagged. At that point, the developer just switches off. The right way to use tools is to do so from the beginning. To help support this, significantly improved *integration* of these tools in existing development pipelines is important.

In addition, there is certainly a range of assurance provided by various tools, with, understandably, the lowest level of assurance provided by the tools that are easiest to use. A superb illustration of the spectrum of tool classes and the strength of guarantee they provide vs. the level of user effort and scalability required appears in Fisher, Launchbury, and Richards' article reflecting on the DARPA HACMS program [26]. As further pointed out by Fisher et al:

> "There are a whole range of different kinds of formal methods, as shown in the notional graph ..., including type systems, model checkers, sound static analysers, verified runtime monitoring, automatic theorem provers and interactive proof assistants. The horizontal axis of the graph shows how much effort is required to use a particular tool, with automatic techniques that can scale to as much code as we can write on the left and labour-intensive tools that require PhD-level expertise and currently scale to programs on the order of 100K lines on the right. The vertical axis shows the strength of the guarantees, ranging from simple type safety properties at the bottom to full functional correctness at the top. Not surprisingly, the most scalable techniques (type systems) provide the weakest guarantees, and the most labour-intensive techniques provide the strongest (interactive proof assistants).
>
> ...
>
> "There is a continuum of formal-method techniques, however. Tools requiring lower levels of effort can be useful to a much broader audience. For example, Facebook has built and deployed INFER, which is a sound static analyser. INFER can process millions of lines of code and thousands of diffs per day. It requires 4 h to analyse the complete Facebook Android/iOS code base. More importantly, it takes less than 10 mins to process a single diff, which allows the tool to be integrated into the standard Facebook development process. When developers try to check in modifications, INFER runs and the developers are required to address any issues INFER finds before they can complete their check-in, which ensures that certain kinds of bugs never enter the production code base. In exchange for this speed, the properties that INFER proves are relatively weak: only the absence of null pointer exceptions and resource leaks." [26]

Fisher et al's point that if something can be automatically identified before it is checked in, the chances that it will be fixed increase dramatically. Thankfully languages like Rust and tools like INFER now exist to provide this technique, and their impact on memory errors by its users are therefore tremendous. The opportunity is to increase the use of these approaches more broadly, and to increase their capabilities to minimize other errors.

Deep specification is essential [27] and more is needed [28]. To put it simply:

> "Formal methods are the *only* reliable way to achieve security and privacy in computer systems. Formal methods, by modeling computer systems and adversaries, can prove that a system is immune to entire classes of attacks (provided the assumptions of the

models are satisfied). By ruling out entire classes of potential attacks, formal methods offer an alternative to the 'cat and mouse' game between adversaries and defenders of computer systems." [29]

To address the gap in enabling formal verification at scale, we must advance the state of the art of both low-level proof assistants (libraries, interfaces) to help scale the abilities of developers of proof libraries.

Adding formal methods to most systems likely will require "bringing formal methods to where the engineers are." [30] In the near/medium term, this probably means proving simpler properties via adding "contract" annotations to imperative languages rather than anything that looks like writing in higher-order logic.

Or consider the use of safer programming languages:

> "When you are programming in Rust you are essentially proving the absence of memory corruption in a type system (assuming you're not using "unsafe" regions)"[58]

Thus we must help software engineers write specifications more easily and write programs that are more easily verified. We need easy-to-use languages to for programmers to write annotations and mechanisms to extract those annotations from existing C/C++/Rust code into formal specifications usable by proof assistants We need formally verified mechanisms (e.g., Certified Abstraction Layers, Refined C, SAW, BriCk, and Verified Software Toolchain) to extract existing C++ or Rust code into models usable by proof assistants. And we need improved support for writing models and specifications in easy-to-read logical languages (TLA+2, Alloy), domain-specific languages (Cryptol/cryptography, Sail/hardware) and domain-specific modeling languages (Simulink/stateflow), or C-like languages (PlusCal, Dafny, Frama-C, OpenJML, Bedrock2 [31, 32]) that are easily verified in a theorem prover.

On the other hand, at the other end of the spectrum, it is essential that we advance the state of the art of tools and languages that in reality, software engineers will write with that help enable formal specification and verification at scale. This will likely require focus on imperative languages — typically "reduced" versions of languages that can be more easily verified, may also leverage contract annotations (aka design-by-contract) to derive specifications. Of course, there will be situations where software engineers are not involved in writing specifications at all, and a method for extracting program logic from existing code and marrying that with specifications written by a proof engineer are likely also necessary.

To address this, we must help proof engineers prove correctness more easily, or reduce the need for proof engineers. To achieve this, we need continued enhancement to proof assistant tooling (beyond CoqIDE, emacs, and limited VSCode support), continued enhancement to proof automation (e.g., via better annotation and programming language design that enables extraction of specifications and models), and a substantially enhanced set of proof libraries to improve proving automation.

At the same time, numerous open research questions exist: How "expressible" do languages used for formal specifications need to be, with SPARK/Ada on one end, for example, and Coq on the other? What's in the middle and how close to one end or the other do many developers need to be? What are the consequences for formal specification expressibility, execution performance, TCB complexity/size, and manual proving effort required for translating models from C/Rust and extracting specifications from annotated C/Rust code vs. re-implementing in Bedrock? How

[58] AWS Blog: "A gentle introduction to automated reasoning" - `https://www.amazon.science/blog/a-gentle-introduction-to-automated-reasoning`

much easier is formal verification of Rust code than C code since memory issues with pointers don't need to be addressed? How can programming languages be designed to make proving easier by automatically removing large classes of problems like Rust (memory safety), Ada/SPARK, or OCaml/Haskell (already functional)? What are the UI/UX implications for software engineers of learning a C-like language built on HOL (Bedrock, PlusCal) vs. (heavily) annotating C/C++/Rust code? How can we improve tooling to make it more accessible to specify and verify systems spanning hardware, architecture, and software?

We also have some simple engineering that needs to happen, including C/C++ to Rust conversion (converts what it can automatically), a "Rust 2.0" that is easier to develop in and doesn't include "unsafe" features, a Rust runtime that is rewritten in Rust, and a vastly expanded set of useful Rust libraries.

We note in passing that software assurance is not the only way to gain safety from software bugs. Hardware also can have a significant role. Microsoft estimates leveraging CHERI [33, 34, 35] for spatial and temporal memory safety would have deterministically prevented 70% of bugs submitted to the MSRC in 2019.[59] *Trusted execution environments* or *confidential computing* can also provide software isolation [36, 37, 38].

9.4 Threats

Software assurance tools can make securing software easier but exploit tools make finding exploitable bugs easier. If we don't find a better way to develop secure software faster than attackers can develop tools to find exploits, attackers will continue to have the edge. The *threat* is that complacency for developing the right tools and adopting those tools will enable adversaries to have that edge.

The technologies that would protect our computing infrastructure are generally not presentin that infrastructure. It is a fantasty to think that our existing and incremental approaches will protect us – they are grounded in unsafe properties. Indeed, existing technologies are hopeless — they fail to provide adequate security properties and/or are impossible to secure.

We note that over time, the threat of unsafe software to the United States will continue to grow. In addition to technical development, procurement is an additional area that needs to be addressed. This could begin on the government side, including requiring type-safe languages and formal methods.

10 References

[1] Elisabeth Sullivan and Michelle Ruppel. Chapter 19 (Introduction to Assurance). In Matt Bishop, editor, *Computer Security: Art and Science, Second Edition*. Addison-Wesley Professional, Boston, MA, 2019.

[2] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. Scaling Static Analyses at Facebook. *Communications of the ACM*, 62(8):62–70, 2019.

[3] Patrick Baudin, François Bobot, David Buhler, Loıc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. *Commun. ACM*, 64(8):56–68, jul 2021.

[4] Lori A. Clarke. A Program Testing System. In *Proceedings of the Annual Conference*, pages 488–491. ACM, 1976.

[59] An Armful of CHERIs: `https://msrc-blog.microsoft.com/2022/01/20/an_armful_of_cheris/`

[5] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[6] Henry Gordon Rice. Classes of Recursively Enumerable Sets and their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[7] Dawson Engler and Madanlal Musuvathi. Static Analysis Versus Software Model Checking for Bug Finding. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 191–210. Springer, 2004.

[8] Andrew Adams, Kay Avila, Elisa Heymann, Mark Krenz, Jason R. Lee, Barton Miller, and Sean Peisert. Guide to Securing Scientific Software. Trusted CI Report – `https://doi.org/10.5281/zenodo.5777646`, December 14, 2021.

[9] Barton P. Miller. Fuzz Testing of Application Reliability. `http://pages.cs.wisc.edu/~bart/fuzz/`, 1988.

[10] Barton P Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of Unix Utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[11] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[12] Barton P. Miller, Mengxiao Zhang, and Elisa R. Heymann. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Transactions on Software Engineering*, 2020.

[13] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1910–1913.

[14] Zhong Shao. Certified software. *Communications of the ACM*, 53(12):56–66, 2010.

[15] Byron Cook. On the Business of Proof (Interview). *Workshop on Dependable and Secure Software Systems* — `https://www.youtube.com/watch?v=g-DH_b5bFd4`, 2021.

[16] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, 2009.

[17] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[18] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[19] Gerwin Klein, June Andronick, Gabriele Keller, Daniel Matichuk, Toby Murray, and Liam O'Connor. Provably Trustworthy Systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150404, 2017.

[20] Chris Newcombe. Why Amazon Chose TLA+. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 25–39. Springer, 2014.

[21] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, 2015.

[22] Allen Newell and Herbert A. Simon. The Logic Theory Machine: A Complex Information Processing System. Technical Report P-868, RAND, June 5, 1956.

[23] Robert S Boyer, Matt Kaufmann, and J Strother Moore. The Boyer-Moore Theorem Prover and its Interactive Enhancement. *Computers & Mathematics with Applications*, 29(2):27–62, 1995.

[24] Byron Cook. Formal Reasoning About the Security of Amazon Web Services. In *Proceedings of the International Conference on Computer Aided Verification*, pages 38–47. Springer, 2018.

[25] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, Joshua Cohen, and Stephanie Weirich. Ready, Set, Verify! Applying hs-to-coq to Real-World Haskell Code. *Journal of Functional Programming*, 31, 2021.

[26] Kathleen Fisher, John Launchbury, and Raymond Richards. The HACMS Program: Using Formal Methods to Eliminate Exploitable Bugs. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150401, 2017.

[27] Andrew W Appel, Lennart Beringer, Adam Chlipala, Benjamin C Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. Position Paper: The Science of Deep Specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20160331, 2017.

[28] Adam Shostack and Mary Ellen Zurko. Secure Development Tools and Techniques Need More Research That Will Increase Their Impact and Effectiveness in Practice. *Communications of the ACM*, 63(5):39–41, 2020.

[29] Stephen Chong, Joshua Guttman, Anupam Datta, Andrew Myers, Benjamin Pierce, Patrick Schaumont, Tim Sherwood, and Nickolai Zeldovich. Report on the NSF Workshop on Formal Methods for Security. *arXiv preprint arXiv:1608.00678*, 2016.

[30] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards Making Formal Methods Normal Meeting Developers Where They Are. *arXiv preprint arXiv:2010.16345*, 2020.

[31] Zygimantas Straznickas. *Towards a Verified First-Stage Bootloader in Coq*. PhD thesis, Massachusetts Institute of Technology, 2020.

[32] Andres Erbsen and Samuel Gruetter. Bedrock2: Language and Compiler for Verified Low-Level Programming. `https://github.com/mit-plv/bedrock2`.

[33] Robert N. M. Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[34] Robert NM Watson, Simon W Moore, Peter Sewell, and Peter G Neumann. An Introduction to CHERI. Technical report, University of Cambridge, Computer Laboratory, 2019.

[35] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020.

[36] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. Toward Confidential Cloud Computing: Extending Hardware-Enforced Cryptographic Protection to Data While in Use. *Queue*, 19(1):49–76, February 2021.

[37] Ayaz Akram, Anna Giannakou, Venkatesh Akella, Jason Lowe-Power, and Sean Peisert. Performance Analysis of Scientific Computing Workloads on General Purpose TEEs. In *Proceedings of the 35th IEEE International Parallel & Distributed Processing Symposium*, 2021.

[38] Sean Peisert. Trustworthy Scientific Computing. *Communications of the ACM (CACM)*, 64(5):18–21, May 2021.