



The Next Wave

The National Security Agency's Review of Emerging Technologies

Vol 19 No 1 • 2011

High Confidence Software and Systems



Letter from the Guest Editor

For years the National Security Agency (NSA) has pursued research in high confidence software and systems (HCSS) technologies to improve the assurance of security critical algorithms, protocols, software, and hardware. Along the way, NSA has been a leader in the development of a national, collaborative community of HCSS researchers and sponsors, some of whom are represented in this issue of *The Next Wave* (TNW).

HCSS research has primarily focused upon developing foundational technology and techniques, yielding components and systems that are “correct by construction.” HCSS research has also been aimed at creating analytic techniques to assess and improve the quality of existing code and specifications. Over the years, HCSS research projects have delivered significant advances within both developmental and analytic areas, and yet substantial questions remain unanswered:

- How can software and systems be made secure in a more cost-effective manner, and how can one obtain high assurance that security has been achieved?
- How can one gain confidence in software and systems that are developed at arm’s length, or worse, are of unknown provenance?

This issue of TNW provides a glimpse into the multi-faceted research strategy gaining traction within and beyond the HCSS community—a strategy that attempts to tackle tough questions such as those identified above. Each facet of the strategy, whether **preventive** or **analytic**, will require better evidence—evidence capable of supporting an objective assessment that the system in question meets specified requirements. In short, the need for **evidence-based assurance** is the core tenet of each approach discussed here. Additionally, each article in this issue highlights the strong overlap between preventive and analytic methods, with an emphasis on the early application of analytic methods in the development process. When used at the earliest stages in the process, analytic methods guide development choices, thereby lessening engineering risks.

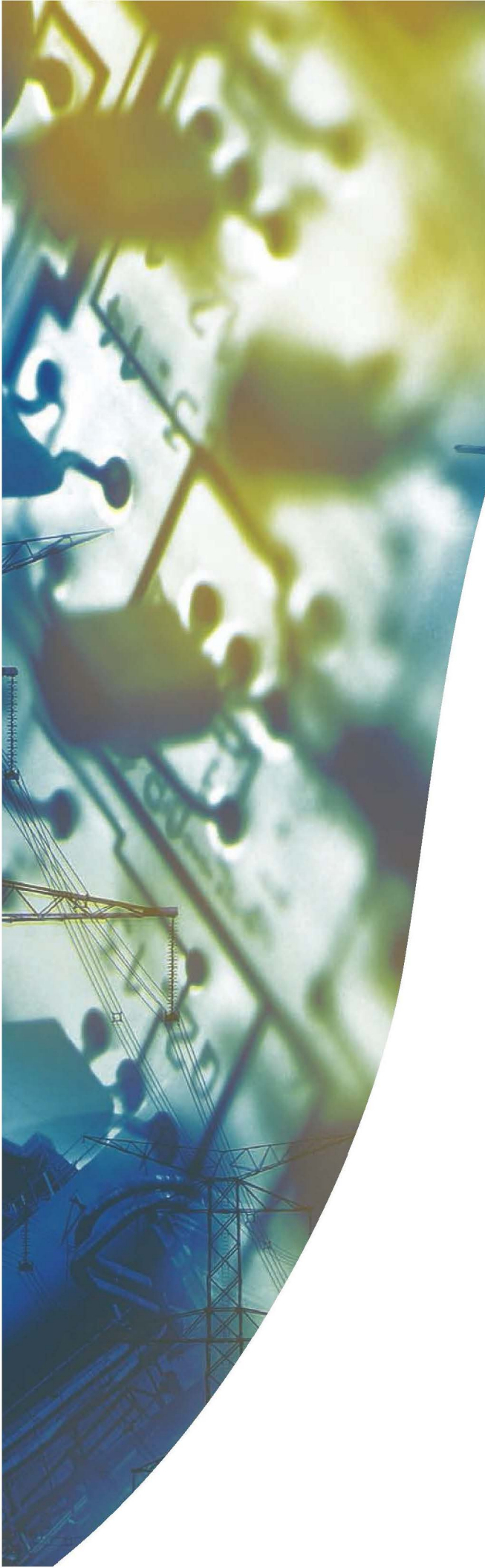
In closing, it would be irresponsible to publish this issue of TNW without explicitly acknowledging the one person I consider to be the heart of the HCSS community within the United States—Dr. Helen Gill from the National Science Foundation. Dr. Gill has worked tirelessly within this community, giving of her time, her talent, and her wisdom. Dr. Gill exemplifies the very best of those who have been charged with advancing science in the public’s interest.



William B. Martin,
Chief, High Confidence Software and Systems Division

The Next Wave is published to disseminate technical advancements and research activities in telecommunications and information technologies. Mentions of company names or commercial products do not imply endorsement by the US Government.



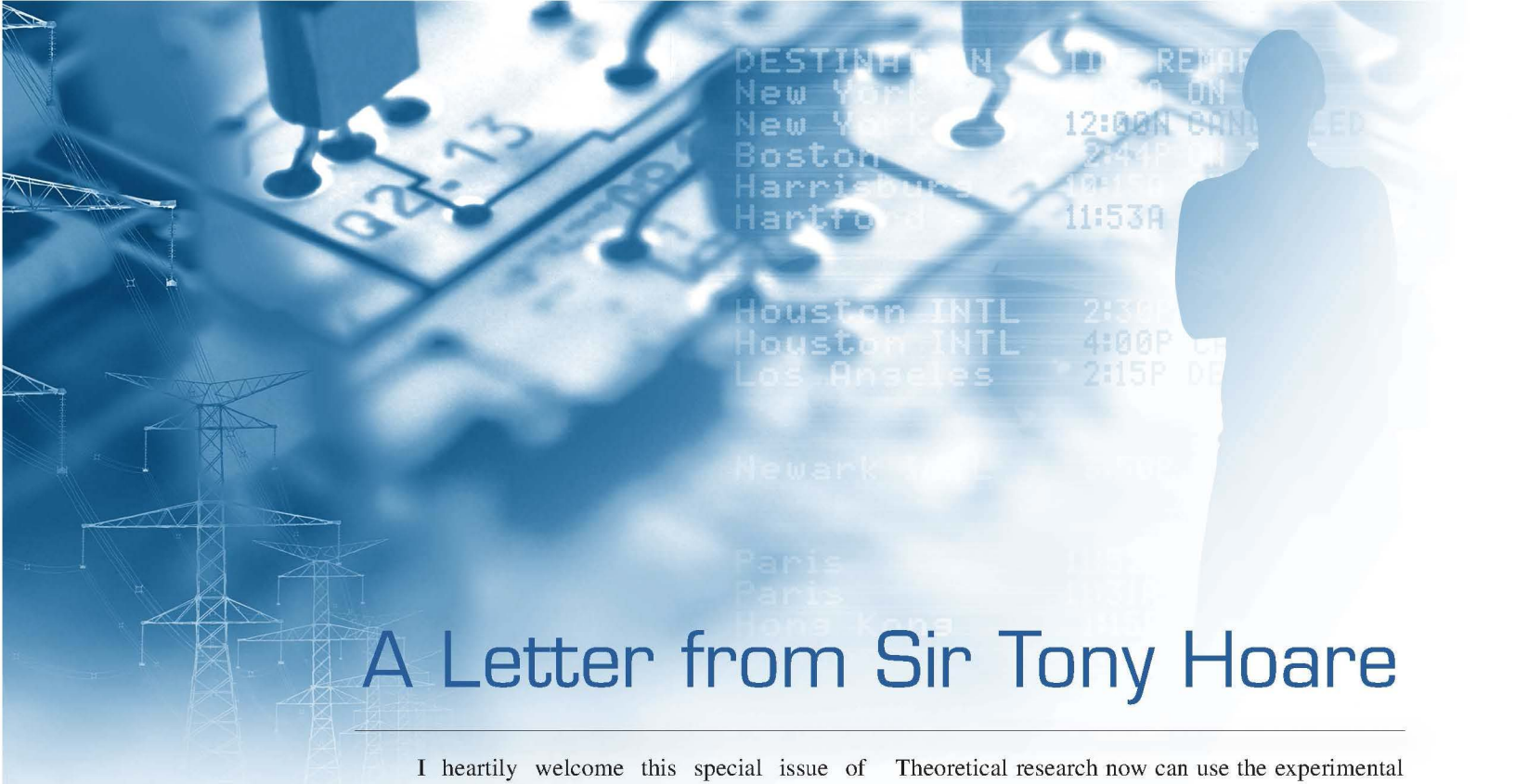


DESTINATION
New York
New York
Boston
Harri
Hart

CONTENTS

FEATURES

- 4 A Letter from Sir Tony Hoare
- 5 Empowering the Experts:
High-Assurance, High-Performance,
High-Level Design with Cryptol
- 14 A High-Assurance Methodology for
the Development of Security Software
- 22 Correct by Construction:
Advanced Software Engineering
- 32 Verified Software in the World
- 34 Software for Dependable Systems:
Sufficient Evidence?
- 44 Critical Code:
Software Producibility for Defense
- 50 Cyber-Physical Systems



DESTINATION	TIME	REMARKS
New York	12:00N	ON
New York	12:00N	CANCELLED
Boston	2:45P	
Harrisburg	10:15P	
Hartford	11:53A	
Houston INTL	2:30P	
Houston INTL	4:00P	
Los Angeles	2:15P	
Newark	3:00P	
Paris	1:00P	
Paris	1:00P	
Hong Kong	2:00P	

A Letter from Sir Tony Hoare

I heartily welcome this special issue of *The Next Wave*. It gives a realistic picture of the advancing state of the art in the specification, design, implementation, and certification of high confidence computer systems.

This topic has interested me since the 1960s, when I first encountered an article by Bob Floyd on Assigning Meanings to Programs. At that time, I judged this was a topic highly suited to pure academic research, a career on which I was just embarking. Like other scientific investigators, we hoped to enlarge scientific understanding of what computer programs do, and how and why they work. We hoped to test the range of applicability of scientific theory by experimental verification of real programs. We were driven by ideals of total program correctness, and total certainty achieved by mathematical proof.

As in other mature branches of science (e.g., physics, chemistry, and most recently biology), the fundamental research has now reached a point where it can be applied in engineering practice. As in other branches of engineering, the key to this technology transfer has been the availability of powerful programme analysis and theorem-proving tools. They are based soundly on scientific theory, but conceal this fact to an appropriate degree from their users. The tools are now subject to continuous improvement in the light of realistic academic and industrial experiments, and by exploiting the increasing performance of algorithms for logical and mathematical reasoning by computer.

Theoretical research now can use the experimental method as a means of differentiating, selecting, and improving the relevant theories for solution of existing and future problems.

The articles in this issue concentrate on advances in tools and experiments. They explicitly outline the remaining deficiencies and difficulties, but I hope that they give sufficient evidence to encourage a wider range of pioneering applications, leading at a sensible rate towards general adoption of computer-assisted programming methods, both by software engineers and by their customers. ■

Tony Hoare

About the author

In 1980, Sir Tony Hoare received the ACM Turing Award for his “fundamental contributions to the definition and design of programming languages,” and in 2000 he was awarded the Kyoto Prize for his “pioneering and fundamental contributions to software science.” These two awards represent the top international accolades available to a computer scientist. Also in 2000 he was knighted by Her Majesty the Queen for services to education and computer science. Sir Tony is now Emeritus Professor at Oxford University, and works as a Senior Researcher at Microsoft Research in Cambridge.

Empowering the Experts: High-Assurance, High-Performance, High-Level Design with Cryptol

A domain-specific language (DSL) is a programming language targeted at producing solutions in a given problem domain by enabling subject-matter experts to design solutions in terms they are familiar with and at a level of abstraction that makes most sense to them. In addition, a good DSL opens the way for powerful tool support: simulations for design exploration; automatic testing and generation of test harnesses; generation of highly specialized code for multiple targets; and generation of formal evidence for correctness, safety, and security properties.

Cryptographer as designer

You are a highly skilled cryptographer charged with designing a custom, state-of-the-art encryption solution for protecting mission-critical information. There are explicit and competing requirements for the implementation—throughput, size, power utilization, operation temperature, etc.—that may affect the implementation.

You produce a design and want to see how it matches up with the implementation requirements. How would you proceed?

Typically, you find an expert hardware designer who translates your algorithm into VHDL (a hardware description language), and then runs proprietary tools to characterize the implementation. If it uses too much power,

or has insufficient throughput, or..., the hardware designer iteratively tweaks the design until it is “good enough.”

But how do you know if it still works the way you intended?

Typically, the design is fabricated (if it is an ASIC—application-specific integrated circuit) or loaded into an FPGA (field-programmable gate array), placed

into a test harness, and blasted with test vectors. If it works, great. Otherwise, the search begins to find the error.

And what if a security hole; for example, a malicious counter or a back door; was introduced? Would you even know?

There must be a better way.

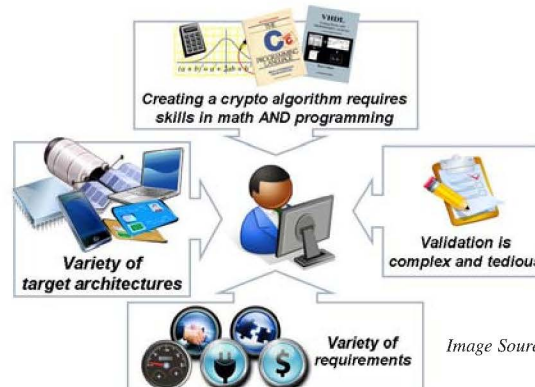


Figure 1: Traditionally, the crypto developer must be highly trained and expert at balancing a myriad of often conflicting requirements.

Image Source: Galois, Inc.

Cryptol: A better way

The Cryptol specification language was designed for the National Security Agency (NSA) as a public standard for specifying cryptographic algorithms [1]. The Cryptol tools provide a development path for cryptographic modules across the entire software process, from specification and implementation to verification and certification. Cryptol tools significantly reduce overall life-cycle costs by addressing the key cost drivers in the deployment of cryptography.

Rapid design cycle

Cryptol specifications are fully executable, allowing designers to experiment with their programs incrementally as their designs evolve. The Cryptol tools support a refinement methodology that bridges the conceptual gap between specification and low-level implementation, thereby reducing time to market. For example, Cryptol allows engineers and mathematicians to program cryptographic algorithms on FPGAs as if they were writing software.

Reusable specification

The Cryptol tools provide a platform-neutral specification language that generates implementations on multiple platforms. Cryptol tools can generate software implementations, hardware implementations, and formal models for verification, all from a single Cryptol program.

Accelerated certification

A Cryptol reference specification becomes the formal documentation for the cryptographic module, eliminating the need for separate and voluminous English descriptions. In addition, Cryptol verification tools show functional equivalence between the specification and the implementation at various stages of the toolchain.

From Section 3.1 of the AES definition [2]:

The **input and output** for the AES algorithm each consist of sequences of **128 bits**... The Cipher **Key** for the AES algorithm is a sequence of **128, 192 or 256** bits. Other input, output and Cipher Key lengths are not permitted by this standard.

In Cryptol:

```
{k}{k >= 2, 4 >= k}
=> ([128],[64*k]) -> [128]
```

Image Source: Galois, Inc.

Figure 2: The constraints and requirements from the Advanced Encryption Standard (AES) [2] can be translated directly into Cryptol types, as shown above. The colored text shows the linkage between English constraint and Cryptol type.

```
des : ([64],[56]) -> [64];
des (pt, key) = permute (FP, last)
  where {
    pt' = permute (IP, pt);
    iv = [ round (lr, key, rnd)
          || rnd <- [0 .. 15]
          || lr <- [(split pt')] # iv
          ];
    last = join (swap (iv @ 15));
    swap [a b] = [b a];
  };

round : ([2][32], [56], [4]) -> [2][32];
round ([l r], key, rnd) = [r (l^f(r, kx))]
  where {
    kx = expand(key, rnd);
    f(r,k) = permute(PP, SBox(k^permute(EP, r)));
  };
```

Image Source: Galois, Inc.

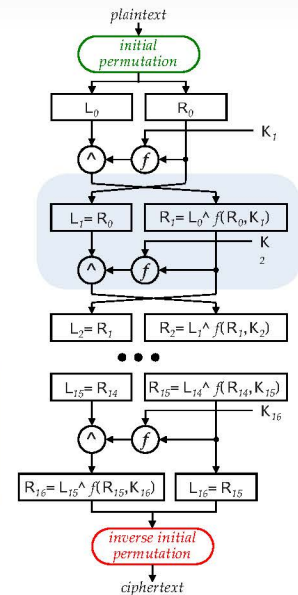


Figure 3: The Data Encryption Standard (DES) algorithm is a block cipher that uses a 56-bit symmetric key. The diagram above is taken from the Standard [3]. Cryptol uses parallel stream comprehensions to interleave data and lazy evaluation to encapsulate multiple computational stages in a single statement. Colors and shapes are used to help relate the program text to the diagram. Details of the language can be found in [4] and at www.cryptol.net.

Design: The Cryptol language

Cryptol [1] is a pure functional language built on top of a polymorphic type system that has been extended with size polymorphism and arithmetic type predicates designed to capture constraints that arise naturally in cryptographic specifications.

Figure 2 shows an excerpt from the AES specification [2] that describes the generator inputs and outputs, and the corresponding Cryptol definition. The text to the left of $\Rightarrow ([128],[64*k])$ in the Cryptol definition describes quantified type variables and predicates on them. In this case, the type is size polymorphic, relying on the size variable k . The

predicates constrain the range of values the quantified size variables can accept; here, k must be between 2 and 4. To the right of the $\Rightarrow$, we see the actual type. The function has two inputs: a 128-bit word containing the plaintext and a $64*k$ -bit wide key. The function outputs another 128-bit word, the ciphertext. Note the precise correspondence of the type to the English description in the standard.

Figure 3 shows a Cryptol code snippet—a specification for the core of the DES algorithm. Note the compact mathematical function notation and the definition of sequence structures and bit sizes. The *Cryptol Reference Manual* [4] has many more examples as well as a detailed description of the language.

Implement: The Cryptol FPGA

Type 1 cryptographic devices protect information of national importance. The information assurance standards for such products are correspondingly high. In addition, crypto modernization requirements mandate field programmability, and various operational requirements call for a reduced space, weight, and power footprint.

FPGAs offer a compelling platform to address these needs. They are field updatable by design, offer tremendous performance potential, and have fewer nonrecurring engineering costs than traditional ASIC designs.

However, FPGA development still requires the considerable time and talents of skilled hardware designers, which increases development time and costs. Mainstream design tools supplied by FPGA vendors have more in common with VLSI (very-large-scale integration) design tools than with modern programming environments. These design tools automatically limit the user population to designers trained in VLSI design.

The Cryptol FPGA generator introduces a new design flow that allows engineers and mathematicians to program cryptographic algorithms on FPGAs in a high-level language incorporating concepts and constructs familiar to cryptologists. The vision is that instead of demanding low-level hardware design knowledge, users are able to express their designs and programs at a much higher level of abstraction and take advantage of powerful automated mechanisms for generating, placing, and routing the circuits.

In some ways, the mathematics behind a cryptographic specification is like a hardware description. Both give unambiguous specification of how bits are to be handled and how bit-level

operations are to be applied. But there the resemblance ends. Sequences, which appear repeatedly in the mathematical descriptions of crypto algorithms, have many different instantiations as hardware. At one extreme, the sequence can be spread out in space as side-by-side parallelism. At the other extreme, the sequence can be laid out in time as consecutive values held in a register, or over many registers in a pipeline. Many combinations of these are also possible.

The Cryptol FPGA generator uses a wide variety of engineering heuristics to pick an appropriate translation of a Cryptol function to an FPGA configuration that will make effective and efficient use of the silicon. The user can also provide pragmas (compiler commands) about space/time mappings, thereby guiding the translation process without compromising the integrity of the original specification.

The declarative quality of Cryptol, which makes Cryptol a good specification language, also plays a key role in the effectiveness of automatic generation of FPGA cores. In contrast, the inherent sequentiality of mainstream programming languages makes them a poor match for the highly parallel nature of FPGAs.

Creating high-performance designs

The Cryptol FPGA generator produces cores whose throughput and area usage have been comparable to (and in some cases better than) hand-coded VHDL/Verilog. For example, an implementation of 128-bit AES for the Xilinx Virtex 4 FPGA has been generated with clock rates in excess of 200 MHz (which translates to throughput of better than 25 Gbps) using only 6912 slices (25 percent of the slices on the chip) and 100 Block RAMs (62 percent of the available Block RAMs). Theoretical results based on Xilinx tools indicate that 500 MHz (65 Gbps) is achievable by these cores.

High-level exploration of the design space

Good design is always at the root of great performance. One of the key factors in Cryptol's performance results is its ability to explore the implementation design space at a very high level. A Cryptol developer can experiment with many different microarchitectures in the course of a few days, covering ground that would otherwise take weeks or months using traditional methods. A variety of implementation approaches can be modeled and characterized quickly.

For example, at the Cryptol level, a straightforward idiom identifies pipelined functional units in hardware. Recall the specification for DES shown in Figure 3. The designer has created a pipelined version of the round function by hand by factoring the high-level Cryptol specification, as shown in Figure 4. The Cryptol FPGA generator produces an efficient pipelined circuit, also shown in Figure 4 on page 8.

High-level design exploration provides a profound advantage in the development of high-performance algorithms (or in algorithms meeting other design constraints). The key is the speed with which the developer is able to iterate the design, the bottleneck of hardware design. A crypto developer can produce rapid design iterations using the Cryptol Toolkit, effectively increasing productivity by up to an order of magnitude over traditional VHDL development.

Trust: The Cryptol verification framework

The FPGA generator uses semantic models to establish the correctness of the process. To gain final assurance, Cryptol developer Galois provides an automatic equivalence checker to prove that the actual code that will run on the FPGA is equivalent to the reference implementation.

```

round : [inf]([2][32],[56]) -> [inf]([2][32],[56]);
round data0 = data3
where {
  data1 = [zero] # [ (expand key ^ permute(EP, r), [l r], key)
                    || ([l r], key) <- data0
                    ];
  data2 = [zero] # [ (SBox(kx), [l r], key)
                    || (kx, [l r], key) <- data1
                    ];
  data3 = [zero] # [ ([r (l ^ permute(PP, sb))], key)
                    || (sb, [l r], key) <- data2
                    ];
};

```

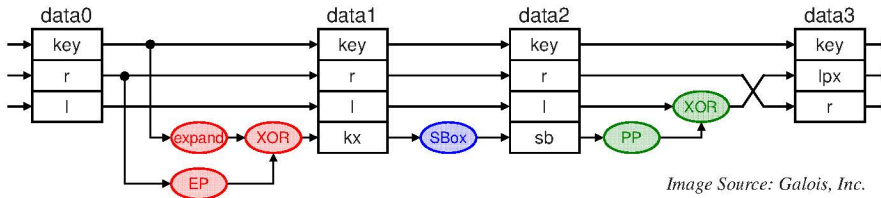


Figure 4: The code snippet above shows a new implementation of the DES round function, shown in Figure 3 on page 6. A flow diagram is included, with colors showing the correspondence between code and diagram element. This version uses sequence comprehensions that can be performed in parallel and introduces extra variables that translate into registers and pipelined operations in the VHDL implementation.

The Cryptol equivalence checker utilizes state-of-the-art SAT (Boolean satisfiability) and SMT (satisfiability modulo theories) solvers as proof engines, together with custom heuristics and techniques. For example, the equivalence checker can show the equivalence of an AES specification written in Cryptol with an unrolled, pipelined VHDL implementation of AES generated from Cryptol and passed through the Xilinx toolchain all the way to place and route.

Two classes of problems

Cryptol’s verification framework has been designed to address equivalence- and safety-checking problems.

The equivalence-checking problem asks whether two functions, f and g , agree on all inputs. Typically, f is a reference implementation of some algorithm, following a standard textbook-style description, and g is a version optimized for time and/or space for a particular target platform. The equivalence-checking framework allows a developer to formally prove that f and g are semantically equivalent, ensuring that the often very complicated and extensive optimizations performed during synthesis have not introduced bugs. Note that the final implementation g does not need to be in Cryptol—an important use case of the verification framework is to verify that third-party algorithm implementations (typically in VHDL) are functionally equivalent to their high-level Cryptol versions. In this case, Cryptol acts as a hardware/software verification tool [5].

The safety-checking problem is about run-time exceptions. Given a function f , we would like to know if f ’s execution can perform operations such as division by zero or index out of bounds. These checks are essential for increasing the reliability of Cryptol-generated implementations, since they eliminate the

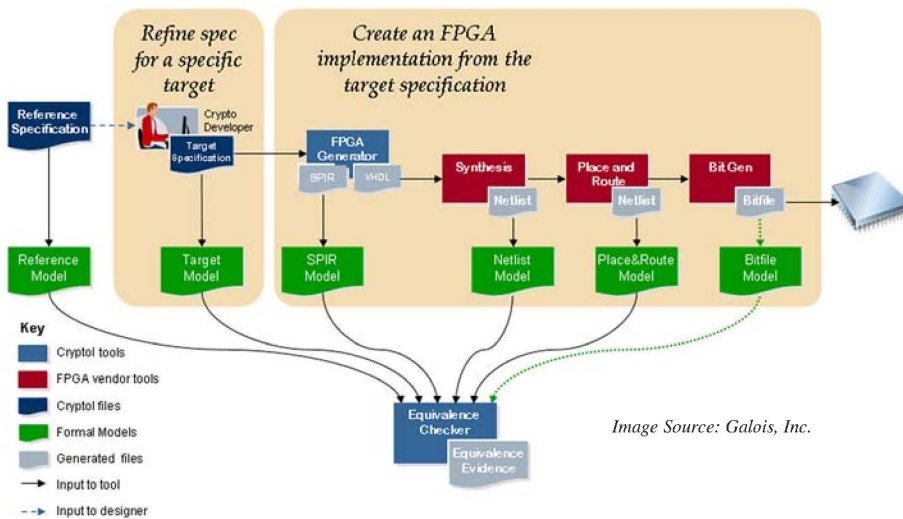


Figure 5: Verification can be performed at various points during the translation, which allows for high-assurance refinement during development. Note that the major compiler phases (the flow through the top line) remain outside the trusted-code base for verification. Trust in the down-arrows, representing translators from various intermediate forms to formal models, along with the off-the-shelf equivalence checkers themselves, is all that is needed.

need for sophisticated run-time exception handling mechanisms.

The Cryptol toolset comes with a push-button equivalence/safety checking framework to answer these questions automatically for a large subset of the Cryptol language [6]. Cryptol uses off-the-shelf SAT/SMT solvers such as ABC [7] or Yices [8] as the underlying equivalence-checking engine, translating Cryptol specifications to appropriate inputs for these tools automatically. However, the use of these external tools remains transparent to the users, who only interact with Cryptol as the main verification tool.

Of course, equivalence checking applies not only to handwritten programs but also to generated code. Cryptol's synthesis tools perform extensive and often very complicated transformations to turn Cryptol programs into hardware primitives available on target FPGA platforms. The formal verification framework of Cryptol allows equivalence checking between Cryptol and netlist representations that are generated by various parts of the compiler, as we will explain shortly. Therefore, any potential bugs in the compiler itself are also caught by the same verification framework. This is a crucial aspect of the system: proving the Cryptol compiler correct would be an extremely challenging if not impossible task. Instead, Cryptol provides a verifying compiler that generates code along with a formal proof that the output is functionally equivalent to the input.

Design and verification flow

Figure 5 provides a high-level overview of a typical Cryptol development and verification flow. Starting with a Cryptol reference specification, the designer iteratively refines the program and “runs” it at the Cryptol command line. These refinements typically include various pipelining and structural

transformations to increase speed and/or reduce space usage. Behind the scenes, the Cryptol toolchain translates Cryptol to a custom signal-processing intermediate representation (SPIR), which acts as a bridge between Cryptol and FPGA-based target platforms. The SPIR representation allows for easy experimentation with high-level design changes, because it remains fully executable while also providing essential timing/space usage statistics without going through the computationally expensive synthesis tasks.

Once the programmer is happy with the design, Cryptol translates the code to VHDL, which is further fed to third-party synthesis tools. Figure 5 shows the flow for the Xilinx toolchain, taking the VHDL through synthesis, place and route, and bit-file generation steps. In practice, these steps might need to be repeated, using feedback from the synthesis tools, until the implementation satisfies the requirements. The overall approach aims at greatly reducing the number of such repetitions by providing early feedback to the user, at the SPIR level. The final outcome is a binary file that can be downloaded onto a Xilinx FPGA board, completing the design process.

Cryptol's verification flow is interleaved with the design process. As depicted in Figure 5, Cryptol provides custom translators at various points in the translation process to generate formal models in terms of AIG (and inverter-graph) representations [9]. In particular, the user can generate AIG representations from the reference (unoptimized) Cryptol specification, from the target (optimized) Cryptol specification, from the SPIR representation, from the post synthesis circuit description, and from the final (post-place-and-route) circuit description. By successive equivalence checking of the formal models generated at these

check points, Cryptol provides the user with a high-assurance development environment, ensuring that the applied transformations preserve semantic equivalence. The final piece of the puzzle for end-to-end verification is generating an AIG for the bit file generated by the Xilinx tools, as represented by the dashed line in Figure 5. At this time, the format of this file remains proprietary.

Verification for the cryptography domain: Why this works

Cryptol's formal verification framework clearly benefits from recent advances in SAT/SMT solving. However, it is also important to recognize that the properties of cryptographic algorithms make applications of automated formal methods particularly successful. This is especially true for symmetric key encryption algorithms that rely heavily on low-level bit manipulations instead of the high-level mathematical functions employed by public-key cryptography.

In particular, symmetric-key cryptographic algorithms almost never perform control flow based on input data, in order to avoid attacks based on timing. The series of operations performed are typically “fixed,” without any dependence on the actual input values. Similarly, the loops used in these algorithms almost always have fixed bounds; typically these bounds arise from the number of rounds specified by the underlying algorithm. Techniques like SAT-sweeping [10] are especially effective on crypto algorithm verification, since simulation-based node-equivalence guesses are likely to be quite accurate for algorithms that rely heavily on shuffling input bits. Obviously, these properties do not make formal verification trivial for this class of crypto algorithms; rather, they make the use of such techniques highly feasible in practice [11].

Verify: Evaluating third-party VHDL implementations

The process of verification in Cryptol typically begins with understanding the high-level interface of the VHDL implementation under study. Through Cryptol's foreign-function interface, the base interface to the VHDL is simply imported using Cryptol's "extern" declaration capability. Then the required interface-matching code is written in Cryptol, mainly implementing the proper use of control signals. This process makes the external implementation available at the Cryptol command prompt, enabling the user to call it on specific values, pass it through previously generated test vectors, essentially making the external definition behave just like any other Cryptol function. This facility greatly increases productivity, since it unifies software and hardware under one common interface. Once the reference specification and the Cryptol/VHDL hybrid expose the same interface, the user generates formal models for both of them, and checks for equivalence.

Challenges ahead

Increasing the coverage of formal methods. Cryptol's formal verification framework works on a relatively large subset of Cryptol [6]. The main limitation is in verifying algorithms for all time, i.e., programs that receive and produce infinite streams of data. Currently, Cryptol can verify such algorithms only up to a fixed number of clock cycles, effectively introducing a time bound. While this restriction is irrelevant for most block-based crypto algorithms, it does not generalize to stream ciphers in general. The introduction of induction capabilities in the equivalence checker or the use of hybrid methods combining manual top-level proofs with fully automated SAT/SMT-based sub proofs might provide a feasible alternative for handling such problems.

Proving security properties. Not all properties of interest can be cast as functional equivalence problems. This is especially true for cryptography. For instance, if we are handed an alleged VHDL implementation of AES, in addition to knowing that it implements AES correctly, we would like to be sure that it does not contain any "extra circuitry" to leak the key. In general, we would like to show that an end user cannot gain any information from an implementation that cannot be obtained from a reference specification.

Reducing the size of the trusted code base. Cryptol's formal verification system relies on the correctness of the Cryptol compiler's front-end components (i.e., the parser, the type system, etc.), the symbolic simulator, and the translators to SAT/SMT solvers. Note that Cryptol's internal compiler passes, optimizations, and code generators (i.e., the typical compiler back-end components) are not in the trusted code base. While Cryptol's trusted code base is only a fraction of the entire Cryptol tool suite, it is nevertheless a large chunk of code from the open-source functional programming language, Haskell. Reducing the footprint of this trusted code base, and/or increasing assurance in these components of the system, is an ongoing challenge. 

Q: What can YOU do with Cryptol?

A: Create a crypto algorithm and generate test vectors.

"...an experienced Cryptol programmer given a new crypto program specification and a soft copy of test vectors can be expected to learn the algorithm and have a fully functional and verified Cryptol model in a few days to a week."

"The AIM crypto engine software engineers at General Dynamics C4 Systems use the Cryptol modeling language as part of their Software Engineering Institute CMM® Level 5 development process. Cryptol provides four basic benefits leading to the certification of crypto equipment. First, Cryptol allows the design engineer to rapidly express an algorithm in a common mathematical notation, which is fully executable on the Cryptol interpreter, providing verification that the algorithm is completely understood. Second, the Cryptol notation for the various components of the algorithm are used to annotate the AIM micro sequencer code which provides much greater readability of that extremely dense assembly language. Third, component testing of AIM code, from small snippets through major subroutines is greatly facilitated with Cryptol generated test vectors derived from end-to-end test vectors provided in algorithm source specifications. Finally, Cryptol models are evolving to directly support the certification effort..."

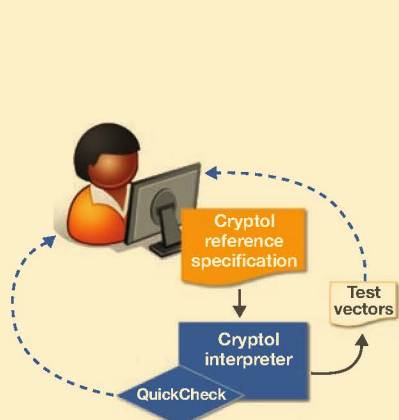


Image Source: Galois, Inc.

Q: What can YOU do with Cryptol?

A: Produce and refine a family of designs.

A team of developers from Rockwell Collins, Inc. and Galois, Inc. has successfully produced high-speed embedded Cryptographic Equipment Applications (CEAs), automatically generated from high-level specifications. An algorithm core generated from a Cryptol specification for AES-256 running in Electronic Codebook mode demonstrated throughput in excess of 16 Gbps. These high-speed CEA implementations comprise a mixture of software and VHDL, and target a compact new embedded platform designed by Rockwell Collins. Notably, almost no traditional low-level interface code was required in order to implement these high-performance CEAs. In addition, automated formal methods prove that algorithm implementations faithfully implement their high-level specifications. Significantly, the Rockwell Collins/Galois team was able to design, implement, simulate, integrate, analyze, and test a complex CEA on the new hardware in less than 3 months.

AES-256, ECB mode, Virtex-4 technology Implementation characteristics	Clockrate (MHz)	Resources (slices)	Throughput (Gbps/second)
Optimized for high throughput	127.5	2690	16.3
Optimized to minimize resource usage	135.1	849	1.2
Handwritten, minimal size	102.0	2535	0.9

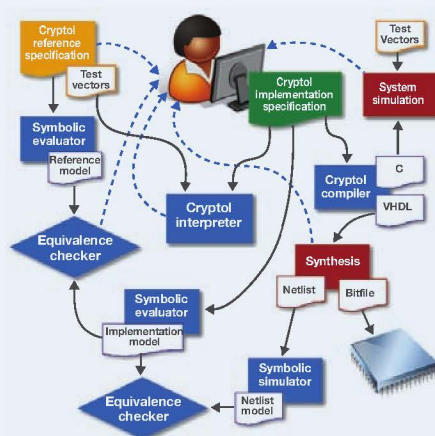


Image Source: Galois, Inc.

Q: What can YOU do with Cryptol?

A: Gain assurance about your design.

Van der Waerden's theorem states that for any positive integers r and k there exists a positive integer N such that if the integers $\{1 \ 2 \ \dots N\}$ are colored, each with one of r different colors, then there are at least k integers in arithmetic progression all of the same color. For any r and k , the smallest such N is the van der Waerden number $W(r,k)$.

Van der Waerden numbers are difficult to compute. In 2007, Dr. Michal Kouril of the University of Cincinnati established that $W(2,6)=1132$ (i.e., 1132 is the smallest integer N such that every 2-coloring of $\{1 \ 2 \ \dots N\}$ contains a monochromatic arithmetic progression of length 6) [19]. The most recent previous result, $W(2,5)=178$, was discovered some 30 years earlier. Kouril computed $W(2,6)$ using a special SAT-solver and clever techniques to bound the search and employed FPGAs to speed up the search.

Kouril wrote VHDL to program the FPGAs. In order to convince himself that the FPGA ensemble was doing what he expected, he also expressed his algorithm in Cryptol, generated formal models for both the Cryptol specification and the VHDL implementation, and verified that the two were equivalent!

Why not let Cryptol generate the solution? So far no one has found a way to prove unsatisfiability of $W(r,k)$ directly without an extensive search. The reliance on search makes the problem hard; and although people have found ways to generate long partitions without a monochromatic arithmetic progression [20], the true test that there are no longer partitions is currently only possible using a search.

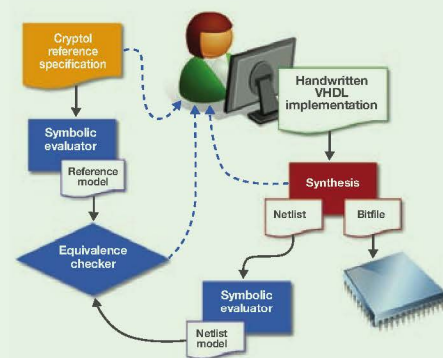


Image Source: Galois, Inc.

Q: What can YOU do with Cryptol?

A: Gain assurance about someone else's design.

Skein [12] is a suite of cryptographic hash algorithms targeted at the NIST SHA-3 competition [13]. At its core, Skein uses a tweakable block cipher named Threefish. The unique block iteration (UBI) chaining mode defines the mode of operation by the repeated application of the block cipher function.

Galois developed and published a Cryptol specification for Skein [14]. We have verified two independently developed VHDL implementations of Skein against our specification for one 256-bit input block, generating a 256-bit hash value.

The first verification was performed against Men Long's implementation [15]. Long implemented only the underlying Threefish encryption and the XOR of input data; we modified our reference specification to match. The AIG generated from the Cryptol specification had 118,156 AND-gates; the VHDL version was more than five times as large, with 653,963 AND-gates. Equivalence checking took about an hour to complete on commodity hardware using ABC [7].

In this work, we encountered a problem with Long's VHDL code that rotated a 64-bit signal a variable distance. The code was given different meanings by GHDL [16], simli [17], and the Xilinx synthesis tools. We removed the ambiguity by replacing it with the standard library function `rotate_left`. Thus, the Cryptol verification path identified an otherwise undetected ambiguity bug.

The second verification was performed against Stefan Tillich's full Skein implementation [18]. The AIG sizes in this case were 301,085 AND-gates for the reference Cryptol versus 900,239 AND-gates for the VHDL implementation: about three times larger. Equivalence checking was completed in about 18 hours, again using ABC.

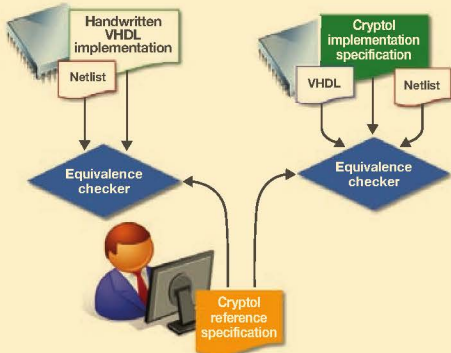


Image Source: Galois, Inc.

Q: What can YOU do with Cryptol?

A: Teach and learn about cryptography, satisfiability theory,....

"Cryptol was quite an experience. We began with simple sequences such as [1 2 3 4] and by applying '@' and '!' to our list of numbers, we learned the priority/position of each number: when using @, the order is zero based, [0th 1st 2nd 3rd], and when using !, the order is reversed, [3rd 2nd 1st 0th]. Each number or element contains a certain number of bits: 1 (0b1) contains one bit, 2 (0b10) is two bits, 3 (0b11) is also two bits and 4 (0b100) is three bits.

Once the group grasped the concept of bits, we moved on to shifting and permuting sequences using split, join, splitBy, groupBy, take, drop, reverse, and transpose. We then applied these fundamentals we had learned about Cryptol to interact with its interpreter and to explore some of the concepts we had learned earlier in the year, such as Pascal's Triangle, the Fibonacci sequence, the sum of a series of odds, even, etc. Once that was complete, and given that Cryptol's intended use is cryptography, we used Cryptol to encrypt plaintext and decrypt ciphertext for a range of classes of cryptographic algorithms, to include classic (substitution and transposition) and modern (symmetric and asymmetric) cryptographic systems.

We concluded our study of Cryptol by looking into propositional logic and satisfiability, and ultimately at a satisfiability solver that could be called from within the Cryptol interpreter. In our examination of propositional logic, we were initially forced to prove our satisfying assumptions by hand through the construction of small truth tables with assignments of values with the goal of having the formula evaluate to 'true', that is, they were satisfied. To extend these concepts we utilized the automated satisfiability solver that we could call from the Cryptol interpreter. One application where we were able to represent a problem within Cryptol and to utilize the satisfiability solver was in solving Sudoku puzzles. It was an amazing experience and I will continue to play around with Cryptol and the satisfiability solver because it was so very intriguing."

Q: What can YOU do with Cryptol?

A: Make a MILS FPGA.

The Cryptol Development Toolkit from Galois provides a tool flow that puts FPGA implementation into the hands of mainline developers, improving both productivity and assurance, without sacrificing performance.

The Xilinx Single Chip Cryptographic (SCC) technology enables Multiple Independent Levels of Security (MILS) in a single chip. These two technologies fit seamlessly into a single development flow.

The combined solution can address high-grade cryptographic application requirements (redundancy, performance, red/black data, and multiple levels of security on a single chip) as well as high assurance development needs (high-level designs, automatic generation of implementation from design, automatically-generated equivalence evidence), and has the potential to significantly reduce the time of costs of developing Type-1 cryptographic applications.

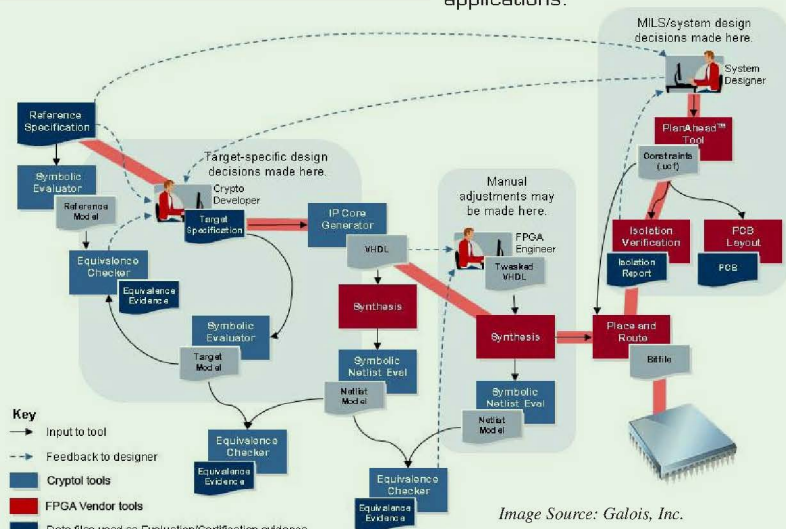


Image Source: Galois, Inc.

References

- [1] Lewis JR, Martin B. Cryptol: High-assurance, retargetable crypto development and validation. In: Proceedings of Military Communications Conference 2003 (MILCOM 2003); Oct 2003; Monterey (CA). p. 820–825. Available at: doi: 10.1109/MILCOM.2003.1290218
- [2] Announcing the AES. NIST; Nov 2001. FIPS Publication 197. Available at: <http://csrc.nist.gov/publications/PubsFIPS.html>
- [3] Data Encryption Standard (DES). NIST; Oct 1999. FIPS Publication 46–3. Available at: <http://csrc.nist.gov/publications/PubsFIPS.html>
- [4] Galois, Inc. The Cryptol reference manual. Available at: <http://www.cryptol.net>
- [5] L. Erkök L, Carlsson M, Wick A. Hardware/software co-verification of cryptographic algorithms using Cryptol. In: Proceedings of Formal Methods in Computer Aided Design (FMCAD '09); Nov 2009; Austin, (TX). p. 188–191. Available at: doi: 10.1109/FMCAD.2009.5351121
- [6] Erkök L, Matthews J. Pragmatic equivalence and safety checking in Cryptol. In: Proceedings of Programming Languages meets Program Verification (PLPV'09); Jan 2009; Savannah (GA). p. 73–81. Available at: <http://portal.acm.org/citation.cfm?id=1481860>
- [7] Mishchenko A. Berkeley Logic Synthesis and Verification Group. ABC: System for sequential synthesis and verification, release 70930. Available at: <http://www.eecs.berkeley.edu/~alanmi/abc>
- [8] Yices: An SMT Solver. Available at: <http://yices.csl.sri.com/>
- [9] Biere A. The AIGER And-Inverter Graph (AIG) format, version 20071012. Available at: <http://fmv.jku.at/aiger/>
- [10] Kuehlmann A, Paruthi V, Krohm F, Ganai MK. Robust Boolean reasoning for equivalence checking and functional property verification. IEEE Trans. on CAD of Integrated Circuits and Systems. 2002;21(12):1377–1394. Available at: doi: 10.1109/TCAD.2002.804386
- [11] Smith EW, Dill DL. Automatic formal verification of block cipher implementations. In: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design (FMCAD '08); Nov 2008; Portland (OR). p. 1–7 Available at: doi: 10.1109/FMCAD.2008.ECP.10
- [12] Ferguson N, Lucks S, Schneier B, Whiting D, Bellare M, Kohno T, Callas J, Walker J. The Skein hash function family. 2009. Available at: <http://www.skein-hash.info>
- [13] NIST's Cryptographic hash algorithm competition. 2008. Available at: <http://csrc.nist.gov/groups/ST/hash/sha-3>
- [14] Finne S. A Cryptol implementation of Skein. Galois, Inc.; 23 Jan 2009. Available at: <http://corp.galois.com/blog/month/january-2009>
- [15] Long M. Implementing Skein hash function on Xilinx Virtex-5 FPGA platform. 02 Feb 2009. Available at: <http://www.skein-hash.info/downloads>
- [16] GHDL simulator version 0.26. Available at: <http://ghdl.free.fr/>
- [17] Symphony EDA. VHDL Simili simulator version 3.1. Available at: <http://www.symphonyeda.com/products.htm>
- [18] Tillich S. Hardware implementation of the SHA-3 candidate Skein. Report 2009/159; Apr 2009. Cryptology ePrint Archive. Available at: <http://eprint.iacr.org/2009/159>
- [19] Kouril M, Paul JL. The van der Waerden number $W(2, 6)$ is 1132. Experimental Mathematics. 2008;17(1):53–61. Available at: <http://www.expmath.org/expmath/contents.html>
- [20] Herwig PR, Heule MJH, van Lambalgen PM, van Maaren H. A new method to construct lower bounds for van der Waerden numbers. The Electronic Journal of Combinatorics. 2007;14(1):R6. Available at: http://www.combinatorics.org/Volume_14/v14i1toc.html

Further reading

Hardin DS, Browning SA. HSE final report, available from the authors.

Hardin DS, editor. Design and verification of microprocessor systems for high-assurance applications. 1st ed. Springer; 15 Mar 2010. ISBN-10: 1441915382

Lewis JR, Hoffman C, Browning SA, Martin WB. A complete design flow for MILS in a single high-assurance FPGA. In: Proceedings of the Second Annual European Reconfigurable Radio Technologies (ERRT) Workshop; Jun 2010. Mainz, Germany. Available at: <http://groups.winnforum.org/p/cm/ld/fid=98>

McLean M, Moore J. FPGA-based single chip cryptographic solution. Military Embedded Systems. Mar 2007. Available at: <http://www.mil-embedded.com/articles/id/?2069>



A High-Assurance Methodology for the Development of Security Software

1. Introduction

Security systems require especially high levels of assurance of correctness, reliability, and security. Researchers in the National Information Assurance Research Laboratory (now Trusted Systems Research) at the National Security Agency (NSA) with the assistance of engineers at Rockwell Collins conducted a project to exercise, evaluate, and enhance a methodology for developing high-assurance software for an embedded system controller. In this approach, researchers captured system requirements precisely and unambiguously through functional specifications using the Z (pronounced “zed”) formal specification notation. Rockwell Collins then implemented these requirements using an integrated, model-based software development approach. The development effort was supported by a suite of tools that provides automated code generation and support for formal verification. The specific system is a prototype high-speed encryption system, although the controller could be adapted for use in a variety of critical systems in

which very high assurance of correctness, reliability, and security or safety is essential. In this article, we use the High Speed Crypto Controller (HSCC) project to illustrate a development methodology which we believe is useful in producing both high quality software and the assurance evidence to support evaluation.

In order to study advanced high-speed electronics technology, hardware research engineers in the NIARL started a project to build a prototype high-speed encryption system. The system architecture they arrived at is shown in Figure 1.

In this design, the data accelerators handle input/output functions, data formatting, and enforcement of some security policy rules. The encrypt core and decrypt core perform the actual encryption and decryption. These six subsystem blocks are in the high-speed data paths. The control block manages the subsystem blocks but lies outside the high-speed data path. An important consequence of this architecture is that the HSCC does not need to be implemented using any exotic high-speed electronics technology.

The critical HSCC design goals are high reliability and achieving very high assurance of functional correctness and essential security properties. As a result, project responsibility for implementing the data accelerators and the crypto cores remained with the hardware engineering organization while responsibility for the HSCC was passed to the High Confidence Software and Systems (HCSS) Division.

Because of the research mission of the HCSS division, the project had two main goals. The first goal was to deliver a working controller. The second goal was to exercise, evaluate, and try to enhance a strong software development methodology. Since HSCC is a security system, the methodology has to support a full range of development aspects from requirements through very rigorous evaluation by independent evaluators. In addition to being rigorous, it should also be cost-effective in time and money.

Given the project goals and the limited resources of our research organization, we in the HCSS division needed an industrial partner. We found the ideal partner in Rockwell Collins. One reason for teaming with Rockwell

Collins was their capability with the AAMP7G microprocessor and high-assurance FPGA development. The AAMP7G supports strict time and space partitioning in hardware, and has received an NSA MILS certification based in part on a formal proof of correctness of its separation kernel microcode, as specified by the EAL-7 level of the Common Criteria [1]. The formal verification of the AAMP7G partitioning system was conducted using the ACL2 theorem prover and culminated in the proof of a theorem that the AAMP7G partitioning microcode implements a high-level security policy [2].

Perhaps more important than their hardware capabilities, Rockwell Collins has a solid approach to software development. It features an integrated, model-based development suite of tools—a toolchain—with a focus on providing a domain-specific modeling environment that abstracts the implementation details, promotes architectural level design, and provides automated transformations between the problem domain formalisms and the target platform. The tools simplify code development and facilitate the application of automated formal analysis tools. In addition, the toolchain is capable of interfacing directly to a simulation environment, providing another level of assurance of design correctness.

For their part, HCSS researchers have experience in the Z specification language [3]. They have written Z functional specifications and design descriptions for several internal development projects [4]. In these projects, [5,6] HCSS researchers played the role of customers and read and commented on draft specifications and designs in Z written by Praxis High Integrity Systems. In addition to experience in the requirements stage of development, HCSS people are familiar

with the security evaluation work done by other NSA personnel.

The approach we chose for the HSCC project was for HCSS researchers to take the lead in writing control software requirements in the form of functional specifications in Z. Rockwell Collins would take these specifications as input into their established development process. They would look for opportunities to strengthen the process, including the support for evaluation, or save time and money by taking advantage of the formal specifications.

2. Z specification work

Over the last ten years, HCSS researchers have worked with other organizations using Z in support of a variety of development projects. We use the Z/EVES [7] support tool and have found it quite suitable for our needs. Based on our experience, we chose to use Z to write functional specifications on this high-assurance controller project.

On this project we tried to follow good habits acquired over the years. We think carefully about names and try to

use clear helpful names and well-chosen abbreviations. We have a house style for notational details such as capitalization. The important point is that both writers and readers of Z benefit greatly from a consistent style. The specific details of the style are not nearly as important as the fact that there is a set of standard conventions. In our finished documents, we adhered strictly to the principle that every Z paragraph was immediately preceded by an accurate natural language translation.

Since the HSCC project was to produce the controller for a crypto system, we had to describe, at a suitable level of abstraction, the main work of the system. On the outbound data path this includes accepting, filtering, and formatting unsecured data in the Red Ingress data accelerator; encrypting in the encrypt core; and formatting and sending secure data out in the Black Egress data accelerator. The inbound data path is a mirror image with a decrypt core.

From this basic system analysis we could see what control data structures had to be provided by the controller to

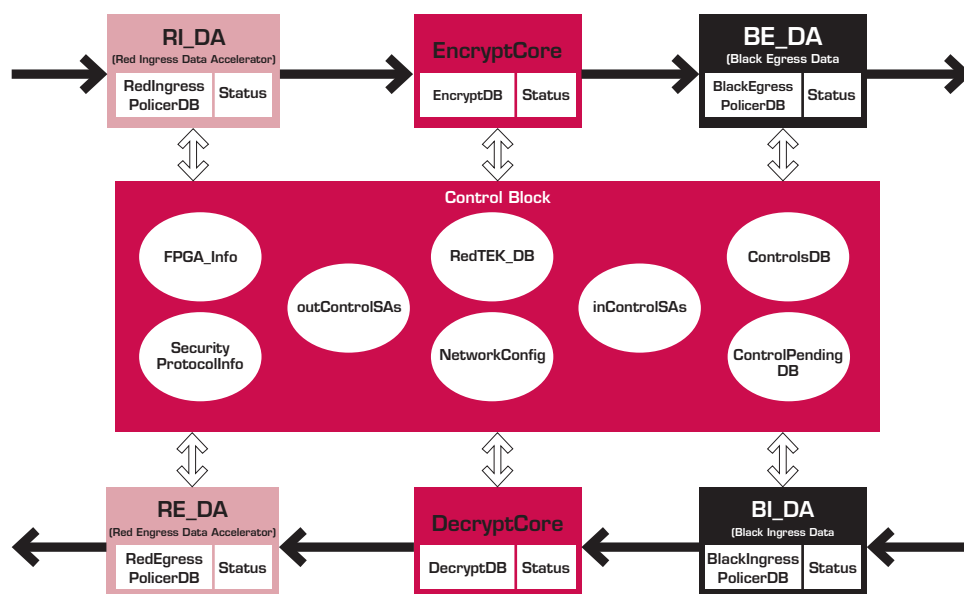


Figure 1: High-Speed Crypto System functional block diagram

Figure 2: Z specification of the routing table database

graphical modeling languages that can be executed and analyzed before the actual system is built. The use of such modeling languages allows the developers to create a model of the system, execute it on their desktops, analyze it with automated tools, and use it to automatically generate code and test cases.

Software for the HSCC system was developed in two parts. Some code was hand coded by a human guided by the Z spec and general engineering knowledge. Other code was generated using portions of the tool chain in Figure 3.

System software (drivers and interrupt/trap handling) and portions of the high-level application code (message

3. Model-based development

formatting and control processing) were implemented in hand-coded SPARK. This code includes information flow annotations to enable use of the Praxis toolchain and to provide assurance of correctness.

Database transactions were designed and developed using the Rockwell Collins MBD toolchain, Gryphon [8]. Simulink/Stateflow models were created for each database transaction. Each model was then tested via simulation in the Reactis tool to discover and correct obvious errors. When complete, the Gryphon framework is used to translate the model into the Prover tool. Gryphon supports several back-end formal analysis tools, including Prover, NuSMV and ACL2; for this project, Prover was deemed to have the best combination of performance and automation. Prover is used to exhaustively verify each transaction preserves properties (derived from Z specifications) about the database it is acting upon. The Simulink model proven to be correct was then used to generate SPARK-compliant Ada95 for use on the target. Figure 3 illustrates the process flow.



The HSCC software development process relies on a several tools:

Simulink®, Stateflow®, and MATLAB® are products of The MathWorks, Inc. [9] Simulink was chosen for development because it is the standard model-based development environment at Rockwell Collins and has extensive existing tool support, including support for formal analysis.

Reactis® [10], a product of Reactive Systems, Inc., is an automated test generation tool that uses a Simulink/Stateflow model as input and auto-generates test code for the verification of the model. The test suites may be used in testing of the implementation for behavioral conformance to the model, as well as for model testing and debugging.

Gryphon [8] refers to the Rockwell Collins tool suite that automatically translates from two popular commercial modeling languages, Simulink/Stateflow® and SCADE™ [11], into several back-end analysis tools, including model-checkers and theorem provers. Gryphon also supports code generation into SPARK/Ada and C. Gryphon uses the Lustre formal specification language as its internal representation and has been used at Rockwell Collins on several significant formal verification efforts involving Simulink models.

Prover [12] is a best-of-breed commercial model-checking tool for analysis of the behavior of software and hard-ware models. Prover can analyze both finite-state models and infinite-state models, that is, models with unbounded

integers and real numbers, through the use of integrated decision procedures for real and integer arithmetic.

By leveraging its existing Gryphon translator framework, Rockwell Collins designed and implemented a toolchain capable of automatically generating SPARK-compliant Ada95 source code from Simulink/Stateflow models.

3.2 Transaction development

Simulink/Stateflow models are used as the common starting point for both the implementation and analysis. Each model corresponds to a single database transaction. Model inputs correspond to SPARK procedure “in” parameters and outputs correspond to “out” parameters. Note the database object used by each transaction model may appear as both an input and an output if the database is modified by the transaction. In this case, the database object access appears as an “in-out” parameter in the generated code. For each database, one model must be created to initialize the data object, in addition to models to perform necessary transactions (add, delete, lookup) on the database. Additional models are required for the formal analysis to model invariants on the database object. This topic will be covered in more detail in subsequent sections.

The screenshot in Figure 4 shows a sample Simulink model that contains the Dest_Encr_Addr_Found lookup function performed on the routing table. This function performs a lookup in the routing table to determine if the specified destination encryptor address is found in the

table. The inputs (at left) are the routing table [Rt_Tbl] and the destination encryptor address [Dest_Encr_Addr] for which to search. The output (at right) is the Boolean value [Found] resulting from the search. The rectangular block in the center is a Simulink subsystem block that implements the database lookup.

Typically, a transaction model will contain a Stateflow chart inside the Simulink model. Stateflow is well-suited to the implementation of the database operations. The screenshot in Figure 5 shows the contents of the Simulink subsystem block depicted in Figure 4. The heavy vertical bar at the left is a Simulink bus selector. Simulink bus objects are roughly analogous to a record in Ada or SPARK. (The Reactis tool does not allow bus objects as inputs to Stateflow charts, so a bus selector is used to separate the component parts of the bus object into separate inputs to the Stateflow chart.) The large rounded rectangle block is a Stateflow chart.

As stated earlier, a model must be built for each transaction in each database. In the case of the routing table, these are:

- Init** – procedure to initialize the routing table data structure (called upon reset)
- Add** – database transaction to add a routing record to the routing table
- Delete** – database transaction to remove a routing record from the routing table
- Dest_Encr_Addr_Found** – database query to determine existence of destination encryptor address
- Get_Dest_Addr_List** – database lookup to return list of addresses mapped to an encryptor address

Figure 4: Destination Encryptor Address Found model

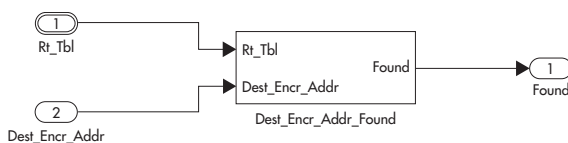
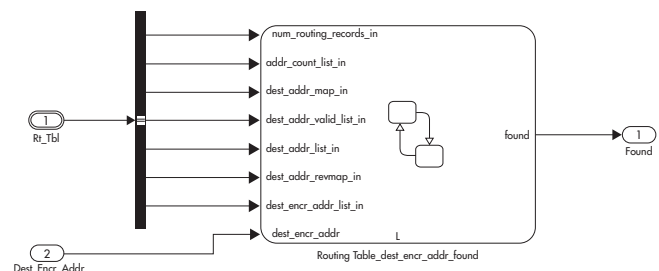


Figure 5: Stateflow chart inside the model



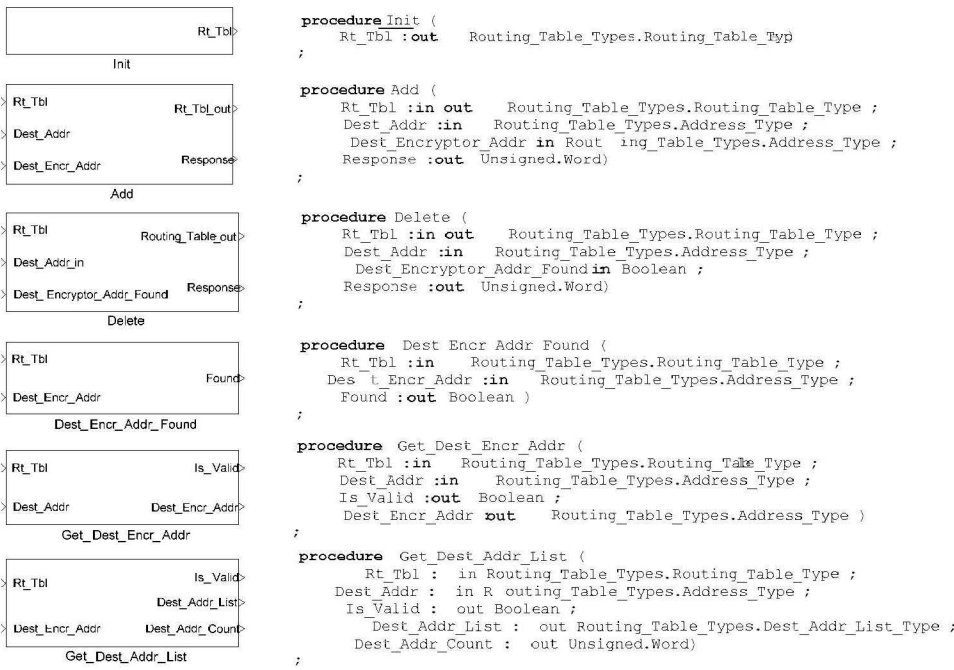


Figure 6: Transaction models and associated SPARK signatures

Get_Dest_Encr_Addr – database lookup to return encryptor address mapped to a destination address

Figure 6 shows the interfaces provided by each model, alongside the generated SPARK procedure signature.

3.3 Invariant modeling

To perform formal analysis on the transaction models, it is first necessary to model any invariants on the data structures. These invariants are taken directly from the Z specification. As an example, the invariants shown in Figure 7 appear in the Z specification for the routing table.

This specification indicates that no duplicate destination addresses or duplicate encryptor addresses may appear in the routing table. These invariants are checked by the `no_dups` model (shown in Figure 8). Given a routing table input [Rt_Tbl], the model checks that no dupli-

cate destination encryptor addresses exist in the data structure and sets the output Boolean values accordingly. Note that the number of Boolean outputs in the model is determined by the internal representation of the routing table data structure, and that the condition in which all four Boolean outputs are “false” indicates that both invariants hold.

3.4 Formal verification

In order to perform the formal verification of a database transaction, we need to establish two kinds of properties: 1) data invariants over the databases (as defined by the Z schemas defining each database) and 2) transaction requirements that ensure that the operation performed by a model matches the Z schema for that transaction. The necessary models include both the transaction model and any invariant models associated with the relevant database(s).

3.4.1 Proof strategy

The proof strategy employed for the data invariants is induction over the sequence of transactions that are performed. We first verify that the Simulink models responsible for initializing each database establish the data invariant for that database. This step provides the basis for our induction. We then prove every transaction that modifies a database maintains the invariant for that database. More concretely, on the “init” models we use the model checker to determine whether or not the data invariants hold on the model outputs. For the other transactions the proof strategy is to assume the invariants in the input “pre” database (prior to performing the transaction), and then use the model checker to determine whether the invariants hold in the output “post” database (resulting from performing the transaction).

We prove all the invariants required by the Z specification and also additional invariants involving implementation details related to realizing the Z databases in Simulink/Stateflow. For example, a linked-list representation is used for many of the finite sets described in the Z document. In this case, additional invariants establish that the linked list is a faithful representation of the finite set.

The transaction requirements for each operation are specified as additional properties that must hold on the “post” database. For example, when deleting an element, these properties ensure that the

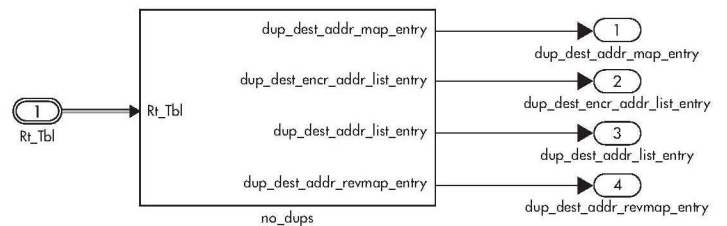
Figure 7: Z specification invariant sample

```

[] kda: knownDestAddresses; rr1, rr2: routingRecords
[] kda [] rr1.destinationAddresses [] kda [] rr2.destinationAddresses
[] rr1 = rr2
[] rr1, rr2: routingRecords
[] rr1.destinationEncryptorAddress = rr2.destinationEncryptorAddress
[] rr1 = rr2

```

Figure 8: Sample invariant model



element in question has been removed from the database.

3.4.2 Formal verification results summary

The formal verification effort for the project as a whole resulted in the proof of some 840 properties for the HSCC databases, of which 140 were written by the verification team and the remainder (mainly well-formedness checks) automatically generated by the Gryphon framework. Verification required less than five percent of total project effort over the course of seven calendar months.

3.5 Code generation

Code generation is performed after a transaction is proven to satisfy all of its invariant properties. Code generation for this project is accomplished through the use of a translation tool, developed during the program, that leverages the existing Gryphon framework to generate SPARK-compliant Ada95 source code for use on the AAMP7G, including the automatic generation of SPARK annotations.

All of the transactions are compiled into single Ada95 package for use by the system programmer. The procedures in the package declaration are shown in Figure 6.

4. Conclusion

Our experiences developing the HSCC system have shown that the methodology described in this paper is a viable process for the development of high-assurance software for use in cryptographic systems.

NSA-provided specifications written in the Z formal notation proved to be superior to those written in English language in producing a complete and unambiguous set of software requirements. Using these specifications as the main development artifact, Rockwell Collins was able to quickly and

accurately determine the necessary “pre” and “post” conditions for each database transaction.

The use of a model-based approach to transaction development provides early simulation capabilities, leading to earlier discovery of errors in both the specification and in the implementation. The use of automated code generation removes the possibility of human coding errors. The application of automated model checkers provides a proof of correctness at a level unattainable

through traditional software testing methods. With all these components in our software development approach, we have exercised a viable methodology to deliver high-assurance software with a much greater level of confidence than software developed through traditional approaches.

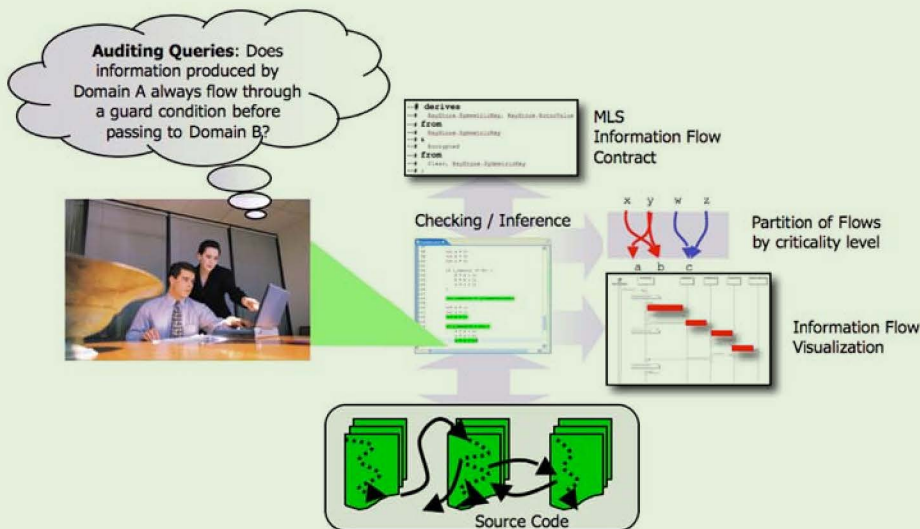
The use of SPARK information flow annotations for Ada95 code at the system level provides assurance the system code is properly routing information to each of the devices



Model-based development is used with increasing frequency in the development of aircraft avionics. By using a model-based development approach, developers can detect errors early, avoiding more expensive fixes later on.

Model-based development was used successfully to develop the ADGS-2100 Adaptive Display and Guidance System (ADGS) Window Manager. In modern aircraft critical status information is provided to pilots through computerized display panels like those shown. The ADGS-2100 is a Rockwell Collins product that provides the heads-up and heads-down displays and display management software for next-generation commercial aircraft. The system ensures that data from different applications is routed to the correct display panel, and in the case of a component failure decides which information is most important and routes that information to the correct display panel. The displays are essential to the safe flight of an aircraft since they provide critical flight information to the flight crew.

Rockwell Collins has developed tools that translate models used to develop systems like the ADGS-2100 to a suite of analysis tools. Verification throughout a design process—while a design is still changing—leads to earlier error detection. During the ADGS-2100 development project, 563 properties were developed and checked and 98 errors were found and corrected in early versions of the model where they are much easier to fix.



Developing and certifying systems with multiple levels of security (MLS) has proven to be extremely challenging. Despite the widespread use of sophisticated integrated development environments (IDEs) with analysis and verification tools for conventional software development, IDEs that provide dedicated support for specification and certification of MLS systems have yet to emerge.

Researchers at Kansas State University are moving to fill this void by developing an IDE called Chispa. Chispa is a visualization, analysis, and verification tool designed to evaluate MLS systems against associated information assurance requirements. For program development, Chispa uses SPARK, a safety-critical subset of Ada developed by Praxis High Integrity Systems and distributed by AdaCore. SPARK is used by various organizations, including Rockwell Collins and the National Security Agency (NSA), to engineer information assurance systems such as cryptographic controllers, network guards, and key management systems.

Chispa uses static analyses to automatically discover information flows in source code. A variety of visualizations are provided to help developers determine if these flows conform to desired MLS policies. System and procedure parameters can be tagged with security policy levels (Top Secret, Secret, Unclassified). Chispa uses its flow analysis to propagate this information to all program statements and to color each statement to indicate the security level of associated data. Chispa includes a software contract language that makes it easy for developers to specify formally the conditions under which information from one data component or security domain is allowed to flow to another. Chispa uses advanced automated deduction techniques to check that procedure and system implementations correctly follow their information flow contracts. Quality assurance teams as well as evaluators for certification authorities can use Chispa's analysis and visualization capabilities to improve the effectiveness of audits and code reviews and to pose automated "what if?" queries related to system assurance.

An early version of Chispa is being used to develop components of the high-speed cryptography engine project at Rockwell Collins.

in the HSCC architecture. Hardware enforced (AAMP7G partitioning) red/black separation serves as the final sentinel in preventing unintended red/black communication. In our judgment, the methodology described in this paper is sturdy enough to support full EAL-7 certification of a production encryptor based on this research prototype. 

References

- [1] Hardin D. Invited tutorial: Considerations in the design and verification of microprocessors for safety-critical and security-critical applications. In: Cimatti A, Jones R, editors. Proceedings of the Eighth International Conference on Formal Methods in Computer-Aided Design (FMCAD 2008); Nov 2008; Portland (OR). p. 1–7. Available at: doi: 10.1109/FMCAD.2008.ECP.5
- [2] Greve D, Richards R, Wilding M. A summary of intrinsic partitioning verification. In: Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004); Nov 2004; Austin (TX). Available at: <http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/>
- [3] Spivey JM. The Z Notation: A Reference Manual. 2nd ed. Prentice Hall International Series in Computer Science; 1992. Available at: <http://spivey.oriel.ox.ac.uk/~mike/zrm/>
- [4] Johnson R. Engineering protection software for the Tokeneer ID station (TIS). The Next Wave. 2006;15(2):21–25, 28–31.
- [5]. AdaCore. The Tokeneer Project. Available at: <http://www.adacore.com/home/products/sparkpro/tokeneer/>
- [6] Barnes J, Chapman R, Johnson R, Widmaier J, Cooper D, Everett W. Engineering the Tokeneer enclave protection software. In: Proceedings of the 1st International Symposium on Secure Software Engineering (ISSSE); Mar 2006; Arlington (VA). Available at: <http://www.altran-praxis.com/downloads/SPARK/technicalReferences/issse2006/tokeneer.pdf>
- [7] Saaltink M. The Z/EVES system. In Bowen JP, Hinchey MG, Till D, editors. Proceedings of the 10th International Conference of Z Users (ZUM'97); Apr 1997; Reading, UK. p. 72–86. Available at: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.4825>
- [8] Whalen M, Cofer D, Miller S, Krogh B, Storm W. Integration of formal analysis into a model-based software development process. In: Proceedings of the 12th International Workshop on Industrial Critical Systems (FMICS 2007), Jul 2007; Berlin, Germany. p. 68–84. Available at: <http://www.msse.umn.edu/publications/Integration-Formal-Analysis-Model-Based-Software-D>
- [9] The Mathworks, Inc. Simulink product description. Available at: <http://www.mathworks.com/products/simulink/description1.html>
- [10] Reactive Systems, Inc. Reactis product description. Available at: <http://www.reactive-systems.com>
- [11] Esterel Technologies, Inc. SCADE Suite product description. Available at: <http://www.esterel-technologies.com/products/scade-suite>
- [12] Prover Technologies, Inc. Prover SL/DE plug-in product description. Available at: http://www.prover.com/products/prover_plugin

Further reading

- Apt KR, Olderog E-R. Verification of sequential and concurrent programs. 3rd ed. Springer; 28 Oct 2010. ISBN-10: 1848827448
- Barnes J. High integrity software: The SPARK approach to safety and security. Addison-Wesley Professional; 25 Apr 2003. ISBN-10: 0321136160
- Brooks FP Jr. The mythical man-month: Essays on software engineering. Addison-Wesley Professional; 12 Aug 1995, anniversary edition. ISBN-10: 0201835959
- Clarke EM Jr, Grumberg O, Peled DA. Model checking. The MIT Press; 7 Jan 1999. ISBN-10: 0262032708
- Gries D. The science of programming. Springer; 1 Feb 1987. ISBN-10: 0387964805

- Hardin DS, editor. Design and verification of microprocessor systems for high-assurance applications. 1st ed. Springer; 15 Mar 2010. ISBN-10: 1441915382
- Harrison J. Handbook of practical logic and automated reasoning. 1st ed. Cambridge University Press; 13 Apr 2009. ISBN-10: 0521899574
- Jackson D. Software abstractions: Logic, language, and analysis. The MIT Press; 24 Mar 2006. ISBN-10: 0262101141
- Kaufmann M, Manolios P, Moore JS. Computer-aided reasoning: An approach. 1st ed. Springer; 31 Jul 2000. ISBN-10: 0792377443
- Kozen DC. Automata and computability. Springer; Aug 1997. ISBN-10: 0387949070
- Manna Z, Waldinger R. The deductive foundations of computer programming. Addison-Wesley Professional; 10 Mar 1993. ISBN-10: 0201548860
- Marek V. Introduction to mathematics of satisfiability. 1st ed. Chapman and Hall/CRC; 22 Sep 2009. ISBN-10: 1439801673
- Pierce BC. Basic category theory for computer scientists. 1st ed. The MIT Press; 7 Aug 1991. ISBN-10: 0262660717
- Pierce BC. Types and programming languages. 1st ed. The MIT Press; 1 Feb 2002. ISBN-10: 0262162098
- Thompson S. Haskell, the craft of functional programming. 2nd ed. Addison-Wesley Professional; 8 Apr 1999. ISBN-10: 0201342758
- Wiedijk F, editor. The seventeen provers of the world: Foreword by Dana S. Scott. 1st ed. Springer; 16 Mar 2006. ISBN-10: 3540307044

INFORMATION

Correct by Construction: Advanced Software Engineering

Over 60 years have passed since the introduction of computers and we still cannot get software right. Why does correct software elude us? First, software systems, maybe the most complex creation of mankind, exceed an individual's capacity to understand. Many different software engineering techniques have emerged over the years to address this complexity, for example structured and object-oriented programming, but failure-prone software persists. Second, subsequent changes to software obscure the author's original intent. In fact, no robust processes or techniques have emerged in practice to document design decisions so maintainers and developers can readily understand the implications of subsequent software changes. However, recent research in correct-by-construction techniques may help. By using formal specifications and automated synthesis we can make correctness claims about these systems and their evolution via an enhanced software engineering process that utilizes formally-described design knowledge. We will never obtain perfect assurance of correctness or security, but we can realize major improvements over current practice.

Formal methods are defined in this paper as traditionally applied in the information assurance domain and in correct-by-construction processes. A particular correct-by-construction (CxC) methodology, which uses the Specware tool, is then described. Specware supports the production of high-assurance¹ code. A programmer using Specware does not directly write or modify code. Instead, the technology creates code systematically and automatically from the programmer's input (the formal specification) and guidance (formally applied design decisions). In conclusion, new CxC techniques that have impacted real-world problems are noted as well as a description of how they could help resolve information assurance problems.

¹ "High-Assurance" in this paper means that the system meets its specification as expressed in the formal language. This includes functional correctness as well as other types of expressible properties.

1. CxC foundations

Many areas of computer science research provide the foundation for this work with CxC methods: artificial intelligence, programming languages, formal methods, and software engineering. The US Air Force Rome Air Development Center (now Rome Laboratory) provided the impetus for CxC research in 1983 by sponsoring the Knowledge-Based Software Assistant report [1], which became the basis for their Knowledge-Based Software Engineering (KBSE) program. KBSE is based on capturing all design decisions in a reusable and checkable form. However, the complexity of the captured information approaches the complexity of the software itself.

Relatively new areas of computer science are providing the structure and power to handle this complexity and achieve KBSE's goal by using CxC methodologies. Industry uses the term CxC to mean methods that range from good process with some formal support, to automated construction of the software from specifications [2,3]. Some examples of science supporting CxC include model-based software engineering and correctness-preserving transformations.

Specware, the CxC system used for the example in this paper, lies toward the automated end of the CxC spectrum, providing an emerging capability to generate correct implementations from software specifications. Although someone using Specware does not need to know category theory (CT)—a unifying concept in mathematics, CT provides the foundation for Specware's ability to structure the knowledge base in such a way to make compiling small, understandable software artifacts into complex ones practical.

A variety of tools have emerged that have prompted industry and academia to experiment with CxC methodologies: knowledge representation techniques and rewriting logic from the artificial intelligence community, compiler enhancements and semantically well-defined languages from the programming language community, reasoning techniques from the formal methods community, and software process improvements and support tools from the software engineering community. This paper describes one approach to CxC engineering.

2. Advanced software engineering

The Specware software development environment provides a good example of how CxC software development incorporates formal methods in ways that can benefit the information assurance (IA) community. In a variety of applications, Specware has already proven to be a powerful tool for specifying, designing, and developing code. Such CxC technologies have the potential to expand the trustworthiness of IA domain applications. What do we mean by "Formal Methods"?

2.1 Definition of formal methods

Formal methods (FM) are used to develop a solution to a problem through a prescriptive process. By applying mathematical rigor, a problem can be studied with precision. In industry the term *formal*

methods can mean simply a good software engineering process. Although FM require good processes, good processes by themselves do not satisfy the FM definition, nor do they guarantee good results. A FM-based software engineering process can achieve a qualitatively more robust solution.

The FM process can be depicted as a triad. The Formal Methods Triad (Figure 1) represents a process for moving from requirements to a solution. At the top of the triangle is a real world problem defined by requirements. To arrive at a workable solution to these requirements, the problem and solution must be described in detail. Therefore, the problem description and the solution description comprise the foundation of the triangle.

Arrows around the triangle represent the processes used to get from each point to the next. The arrows point forward to trace the main flow, but feedback and reiteration are central to the process. Returning to a previous step would occur, for example, when inconsistent or

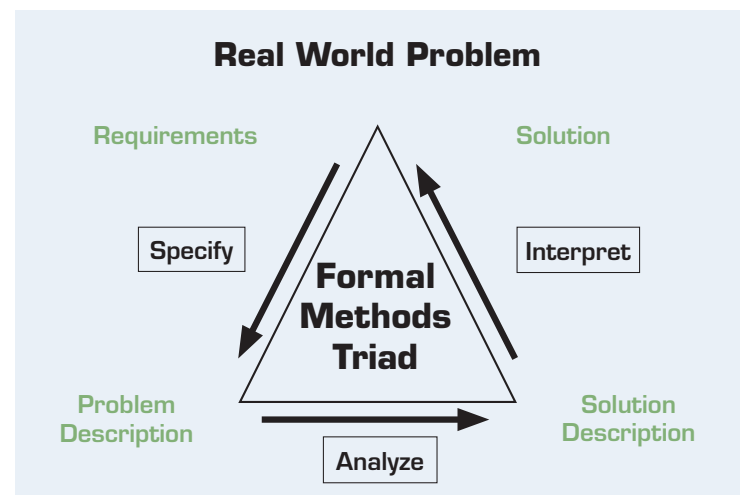


Figure 1: The Formal Methods Triad

Software lifecycle cost reductions due to a CxC development process

In a recent study, Kestrel researchers examined the suite of documentation required for certifying Type 1 devices, and the possibilities to extend Specware's correct-by-construction (CxC) development process to auto-generate certification documents. Our thesis is that by using automated tools to generate both the software and significant portions of its certification documentation, a CxC approach will dramatically lower lifecycle costs, including the cost of recertification. Furthermore, by speeding up the recertification process, a CxC approach facilitates the evolution process, resulting in higher quality products over the lifecycle.

To quantify these claims, we first estimated the cost reductions that arise from a CxC process independently from certification costs. The dominant factor seems to be the size reduction in formal specifications relative to executable code. This size reduction varies considerably over projects, but a ballpark figure of 4-5x is consistent with the JavaCard project and related efforts. A 4-5x reduction in size of the formal text usually correlates with a similar reduction in development and evolution costs. Consequently we estimate that, independent of certification costs, a CxC process should reduce lifecycle costs by roughly 75-80 percent. Second, we estimated the cost reduction due to extensions of the CxC process that allow auto-generation of certification documents as a by-product of the code generation process. For each of the thirteen documents required for certifying Type 1 devices, we estimated that the average cost savings vary from a high of 75 percent for Formal Security Policy Model (FSPM) documents to a low of 20 percent for a Security Verification Plan and Procedures (SVP) documents. Assuming roughly equal weight to each of the 13 documents, we estimated an average overall cost reduction of about 59 percent per certification application due to using CxC methods.

These two estimates can be combined in a variety of ways. For example, if we assume that certification costs are roughly the same as development costs, then CxC brings about a 70 percent reduction in lifecycle costs (evolution plus certification); that is, a CxC process will produce a certified product for 30 percent of the cost of a conventional process. If we assume, as is the case in aerospace applications, that the cost of certification is about 7x development costs, then we obtain an estimate of 63 percent cost reduction for a CxC process. This leads us to conclude that a CxC process will produce a certified product for roughly 30-40 percent of the cost of a conventional process. This estimate does not account for the possibility that some forms of certification become unnecessary because of the strong form of evidence provided by a CxC process.

incomplete requirements are discovered during the formal problem specification phase. Discovering errors early in the process can yield significant cost savings in the long run.

The arrow pointing from Real World Problem to Problem Description is labeled Specify because this step of the process uses formal language to describe the problem. The more expressive the language used, the more complete the analysis can be. The Analyze step moves the process from a problem description to a solution description. Methods that support this analysis with mathematical rigor (e.g., FM) are used. Finally, to arrive at a solution, we map the result of the analysis back to its meaning in the problem domain. This is the Interpret step in the process.

As a FM process example, consider the real world problem of determining how much wind a suspension bridge can withstand. The problem can be described by a set of integral equations representing a property of the bridge affected by wind. Solving these equations produces a solution description. The bridge engineer would select appropriate integral calculus equations—tools from his engineering domain knowledge—and insert the bridge's specific requirements into them and derive a solution. If the calculations from the formal analysis determined that winds sufficient to cause the bridge to collapse occur frequently, the interpretation would probably lead to condemning the bridge. Of course, no one would build a bridge before analyzing its design, but software is routinely built and then analyzed afterward!

2.2 The use of formal methods in the DoD

The US Department of Defense (DoD) has a long history of applying FM. In the 1980s the DoD Computer Security

Center developed what is commonly referred to as the Rainbow Series of standards, beginning with the Trusted Computer System Evaluation Criteria (TCSEC) volume, better known as the Orange Book [4]. In 1996, the National Security Agency's National Computer Security Center (NCSC) replaced the Orange Book with the Common Criteria [5]. Despite the computer security community's early excitement and subsequent disappointments regarding FM, formal methods may still have a future in building high-assurance systems.

The DoD has initiated significant efforts to incorporate FM in the design and evaluation of information security systems. Most notably, FM would apply at the Design Phase of the development and would focus on whether the design has the desired security properties, or at the Evaluation Phase where the implementing code would be analyzed for security vulnerabilities. These approaches have been implemented primarily in the research community and rarely in product development until recently.

2.2.1 Formal methods at design time

Formal methods are applied during the design phase by developing formal specifications of the system and the security policy, typically at a very high level of abstraction. The point of the abstract specification is to define the "what" the system should do and not the "how" it should do it, because the "how" normally biases the system toward particular implementations, thus potentially precluding the best implementations.

To begin this process, the system developer creates a system security policy (a set of requirements) and a system specification. These requirements are expressed most often in natural language. Early on, developers created formal specifications manually, with little

automated support. They would often reuse components of existing specifications, adapting them in much the same way as programmers reuse code. Later, theorem-proving frameworks evolved to help standardize and automate processes of writing system specifications and security requirements. The different theorem-proving tools employ unique variants of formal logic (i.e., a specification language), each having its own strengths and weaknesses. Early tools and languages used for specifying system requirements (some are still used) included EHD, Gypsy, InaJo, Larch, ACL2, Z (zed), NQTHM, GVE, PVS, SDV, Z-EVES, and the Larch Prover. Security policy would then be described using the same formal language.

Given the descriptions for the system requirements and security policy, the developer, when able to understand them, would “prove” that the system specification enforces the security policy. Usually these proofs required building an infrastructure of lemmas that rarely could be reused. The verified high-level design may or may not reflect the actual implementation, although implementations built from verified designs have a much greater chance of meeting their requirements.

An improvement to this method for achieving confidence that the system specification guaranteed the security policy would be to add detail to the specifications and reiterate the process. This, of course, required some kind of demonstration that the more concrete system and security policy specifications indeed represented their corresponding abstract specifications. At some point, usually leaving a significant gap between the most detailed specification and code, iteration would stop and the specification would become the basis for the code. Of course, we know that no code would ever be written before completing the specification!

Rarely, until recently, has anyone attempted to iterate refining a specification by adding detail down to the code level, and in most cases they did this on small slices of the system. The cost of producing code from this process was prohibitive.

This process of formally proven specifications, once too-costly and labor-intensive, has improved. Even so, the benefits of formal specification without complete proofs have provided sufficient value to be required for Common Criteria EAL7 rated systems.

2.2.2 Code-based analysis

The National Security Agency (NSA) has also applied FM through code-based analysis, mainly in support of evaluations of information assurance systems. Code-based analysis differs from the design phase FM process in that, instead of developing a formal description of what the software should do, code-based analysis attempts to discover and prove properties about the software code itself. For example, the user might identify points within the software where properties of interest must hold, and then annotate the code with stylized comments (formal language) about these properties. The developer then applies a tool that understands the semantics of the language and generates verification conditions based on the code and the annotations. This process outputs a set of logical statements that then can be used to validate these user-desired properties.

In the analysis process we attempt to find and prove the targeted properties of the software. A SAT solver (a Boolean “Satisfiability” or SAT solver uses specialized procedures to attempt to satisfy test conditions) or an ATP (an “automated theorem prover” derives the truth of the specified conditions from more basic facts) is applied to these verification conditions. We get any of three possible outputs from this analysis: the conditions

JAVA Card Runtime Environment

We used Specware to formally specify a real-world smart card operating system, the Java Card Runtime Environment (JCRE). The JCRE consists of a JAVA virtual machine (VM) and system libraries (e.g., for I/O and cryptography), along with card management capabilities according to the Global Platform Standard. The formal specification is about 30,000 lines long and over 6,000 consistency proofs of it have been mechanically verified so far. A desktop simulator (reference implementation) has been generated by refinement from the formal specification; the correctness of the refinements is currently being mechanically verified. A C implementation for a commercial chip has been manually derived from the formal specification; a new version of this implementation is currently being generated via automated refinements, with mechanical proofs. We anticipate that this will be the highest level of assurance yet achieved, and that it will reduce the cost and increase the confidence of a Type 1 certification.

Specware has also been used to study the extension of (standard) JCRE with MILS and MLS separation. The study has been carried out on a formal specification of an idealized subset of the JCRE. Separation policies have been formally specified, along with run-time monitors to enforce the policies. The monitors have been formally proved to guarantee the policies. The monitors and the formal proofs are currently being extended from the idealized to the complete JCRE.

See <http://www.kestrel.edu/java> for more information.

can hold; the conditions do not hold; or in some cases, the analysis cannot determine either way if the conditions hold or not with the computational resources given. The first two conditions tell us something useful about the code.

The source and/or binary code level, the detail required to perform the analysis, can be cost prohibitive. However, as better provers and solvers become available and with sheer computing power increases, these methods are becoming more practical. We already see industry using code analyzers based on FM such as CodeHawk, Java Pathfinder, and ESC/Java2.

There is, however, at least one CxC approach that combines the best of both design verification and code analysis.

2.3 A correct-by-construction approach

The formal methods described previously try to ascertain something after the fact. In the design phase, FM are used to verify the specifications after writing them, and in code analysis the analysis is applied to code already developed. The

alternative would be to simultaneously conduct the analysis while creating the specifications/artifacts. By maintaining or enforcing the properties of concern through the development process itself, design errors could be caught early, when repair is less costly. Let us look at the CxC approach for a software development.

The developer typically starts with some notion of what the system should do. Ideally, creative programming energy is directed at capturing system requirements in a specification. This is hard work! CxC with automation moves the work from low-level programming to high-level problem development.

A CxC approach solves the problem incrementally by developing requirements and deriving satisfactory implementations with ever increasing levels of refinement. This is done by using FM to automatically compute code from specifications. CxC industrial practices vary by the degree of automation of the compute process. An ideal CxC approach would completely automate this process. Although full automation is not the only way to go, it

greatly enhances the benefits of CxC for reuse and recertification—an advantage industry is starting to recognize [3].

Figure 2 illustrates the CxC model used for our software development project. At the highest level of abstraction, the requirements, expressed in formal language, plus any existing domain knowledge specifications are used to create a formal specification, or model. This step is represented in the diagram by the Specify arrow. Merely formalizing the requirements in this way can expose inconsistencies and ambiguities early in the process, saving money and time. At lower levels of abstraction, in the Compute step in the diagram, the implementation can be derived through a posit-and-prove approach or a transformative approach.

In the posit-and-prove approach, programmers create the low-level specification and then propose a mathematical argument, with varying degrees of formality, that will prove whether or not the solution specification obeys the problem specification. A code generator is used to generate the code from the low-level solution specification.

With a transformative approach, the user takes the high-level specification of the system and iteratively applies transformations that apply computer science and problem domain knowledge to get an efficient and correct low-level executable specification. This process terminates after transforming the entire problem specification into an executable specification. Remember that the initial specification defined “what” the system should do. Each transformation adds detail and makes decisions that bring it closer to a particular “how” the system should do it. In addition, each transformation preserves the properties and functionality of the source specification in the target specification (the result of the transformation). Once the “how” has been determined and expressed

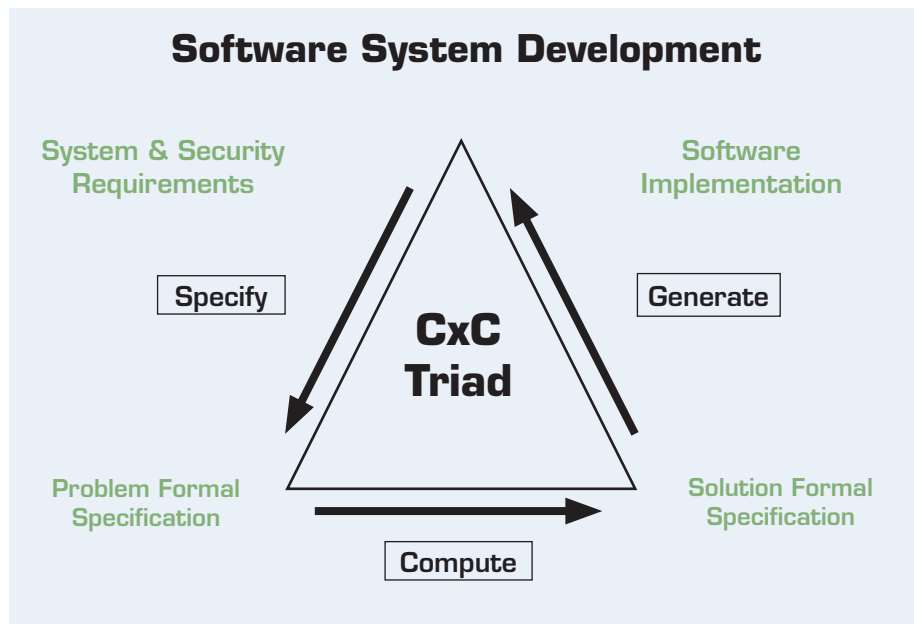


Figure 2: The CxC Triad

as a formal specification, the next step is to generate software in some target programming language (as in the posit-and-prove approach).

The initial high-level formal description created from the requirements in the Specify step must still be validated against the original requirements and specifications in the standard way. Another way to look at this, CxC simply tries to automate the FM process for INFOSEC systems by iterating the process all the way down to code. This begs the question, “Does such a tool exist?”

2.4 Specware, a correct-by-construction tool

Yes, it does: Specware, developed by Kestrel Institute [6]. With this tool we can build specifications for requirements, combine small specifications into larger ones, implement design decisions by refining specifications, and generate code from executable specifications, all the while providing for the proof that the derived specifications and code enforce the requirements.

Specware [7,8,9,10,11] uses a version of Church’s higher-order logic for its specification language. This language borrows features from functional programming and highly expressive automated theorem prover (ATP) languages, and is thus useable by many in the computer science and theorem proving disciplines. Researchers have used Specware for a wide range of specifications, from a full-blown operating system (Java Card Runtime Environment or JCRE) and a mathematically assured separation kernel (the Mathematically Analyzed Separation Kernel [MASK], part of the Advanced INFOSEC Module chip), to low-level algorithms and data structures.

2.4.1 An NSA security token

At NSA we used Specware to successfully build a robust security token.

We contracted for the development of a robust JCRE and robust applets running on a specific hardware platform.

To create the JCRE, a formal specification was developed and a posit-and-prove approach was used to refine the code. The complexity of the initial specification evolved as standards were added. Incremental changes were easier to implement using Specware than in standard software development processes. Even though the Specware tools used at the time were primitive compared with later versions, they were sufficiently robust to produce a working JCRE, complete with cryptography and some assurance of correctness.

Creating applets involved the development of a domain-specific language (SmartSlang) and a corresponding compiler (AutoSmart, produced with Specware). Using SmartSlang a developer can specify an applet more easily than by writing Java Card code directly. The compiler generates both the Java Card code for the applet and a proof that the code

AutoSmart

The AutoSmart (automatic generator of smart card applets) tool is an example of a domain-specific CxC generator. It features a specification language tailored to the smart card domain, with constructs to conveniently capture concepts like personal identification numbers, cryptography, ISO 7816 I/O exchanges, and so on. AutoSmart performs several consistency checks on the applet specifications, including a security analysis that flags potential leaks of confidential information like private and secret keys. AutoSmart compiles the applet specifications to Java Card code, which can be compiled and loaded into a Java Card. Along with the code, AutoSmart also generates documentation for FIPS 140-2 certification as well as informal documentation for the applets (e.g., tables of commands and internal data). AutoSmart is currently being extended with the capability to generate a machine-checkable formal proof of the correctness of the generated Java Card code with respect to the input specifications. This “credible compiler” capability enables trust in the correctness of the code to be shifted from the AutoSmart tool to a much smaller and simpler proof checker, in the spirit of proof-carrying code.

See <http://www.kestrel.edu/jcapplets> for more information.

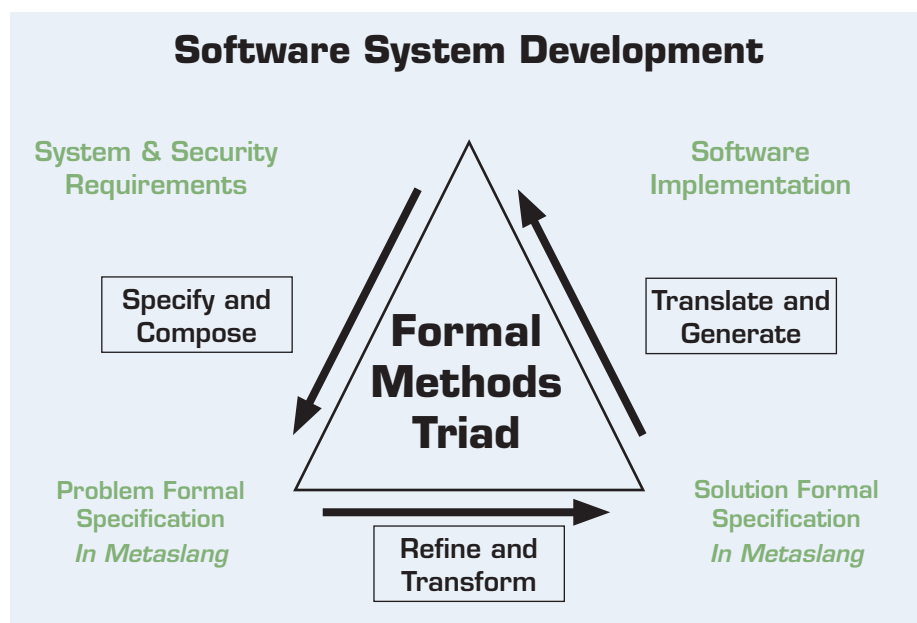


Figure 3: The Specware Triad

Dramatic improvements to propositional satisfiability (SAT) solvers were made during the last two decades [e.g., 1,2]. We used Specware to demonstrate the automated generation of fast SAT solvers. The main result was that we were able to recapitulate many of the key design features of a modern SAT solver using mechanized representations of abstract and reusable design knowledge. Starting with a formal specification of the SAT problem (find a satisfying assignment for a given set of propositional clauses, if any), the overall algorithmic structure of a Davis-Putnam-Logemann-Loveland (DPLL) SAT solver was calculated from the global search and constraint propagation algorithm paradigms [3,4]. Performance of the correct, but high-level algorithm was improved by applying problem-independent transformations for expression simplification, finite differencing [5], and data type refinement. Applying these algorithm design tactics and transformations in different ways resulted in a family tree of SAT algorithm variants, including some novel non-DPLL variants.

This project, together with previous work on scheduling applications [6], provided evidence that it is feasible to generate customized high-performance solvers for particular problems. Key features of state-of-the-art SAT solvers, such as conflict-resolution and learning, can be applied mechanically to other problems.

References

- [1] Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S. Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th Conference on Design Automation (DAC '01); Jun 2001; Las Vegas (NV). p. 530-535. Available at: <http://www.princeton.edu/~chaff/publication/DAC2001v56.pdf>
- [2] Marek V. Introduction to mathematics of satisfiability. 1st ed. Chapman and Hall/CRC; 22 Sep 2009. ISBN-10: 1439801673
- [3] Smith DR. Structure and design of global search algorithms. Kestrel Institute; Nov 1987. Technical report number: KES.U.87.12.
- [4] Westfold S, Smith D. Synthesis of efficient constraint satisfaction programs. Knowledge Engineering Review. 2001;16(1):69-84.
- [5] Paige R, Koenig S. Finite differencing of computable expressions. ACM Transactions on Programming Languages and Systems. Jul 1982;4(3):402-454. Available at: doi: 10.1145/357172.357177
- [6] Smith DR, Parra EA, Westfold SJ. Synthesis of planning and scheduling software. In: A. Tate, editor. Proceedings of Advanced Planning Technology (ARPI '06); May 2006; Edinburgh, Scotland. p. 226-234. Available at: <http://www.aaii.org/Library/ARPI/arp6contents.php>

implements the specification. In addition, the compiler enforces properties stated in the applet specification and produces reports to meet certification requirements.

Kestrel Institute succeeded in producing a working JCRE on the chip running robust applets. Smart card developers are now considering the formal JCRE specification and the Specware toolset. However, given that Specware is not a commercial tool, there is some resistance to its use. Commercial support is critical for wider adoption of this technology.

2.4.2 Using Specware

Building specifications with Specware is no harder than programming. And because high-level specifications should only stipulate “what” is to be done, not “how” to do it, we can understand these top-level specifications much more easily than code. Thus we can make our changes in the high-level specifications and avoid the more complex low-level specifications by redoing the refinements. Most likely, the original design choices will apply with minimal changes and the low-level specifications regenerated. To illustrate this process, we take an example from the Specware tutorial [4].

2.4.2.1 Specification

In the Specware tutorial, the problem for which we want to specify and generate a solution is determining the first match of a word within a message, where a word is list of symbols and a message consists of a list of symbols and wilds (a wild matches all characters). For example, the word “ABCD” would match at the first position of the message “AB*D***” and “BAD” would match at second position.

Here is a typical specification:

```
WordMatching = spec
import Words
import Messages
import SymbolMatching

op word_matches_at?(wrd: Word, msg: Message, pos: Nat): Boolean =
  pos + length wrd <= length msg &&
  (fa(i:Nat) i < length wrd => symb_matches?(wrd@i, msg@(pos+i)))
endspec
```

In this specification, two conditions are necessary for a word match: (1) there is enough room left to contain the word in the message, and (2) all the symbols in the word match their corresponding positions in the message. Notice that this is not saying how to check these conditions; just what it means to match a word with a segment of a message. Also notice that the specification for what symbol matching means (two identical characters or one is a wild) is not in this specification at all, but in an included specification named “SymbolMatching.” So the game in specification is to build up a collection of specifications and compose them to say what is desired.

2.4.2.2 Design

Design involves the composition of a series of refinements to get an executable specification. Each design decision is formally captured and is available for reuse for

both exploring possible implementations of the current specification and for reuse in other developments of similar systems. Consider how hard (and costly) it would be to explore the design space using standard practice. With CxC, design space exploration becomes doable.

The key steps in design are to specify/generate the “how” and define/generate mappings between the “what” and “how” specifications. The mappings constitute property-preserving refinement. If we get the “how” right with respect to the “what,” we will be able to prove all obligations that Specware generates from the mapping construct. Essentially with the properties proved, we ensure that the definitions in source specification are theorems in the target specification. In other words our “how” does correctly “what” we want.

Design is done iteratively until arriving at a specification for “how” to compute the “what” from the high-level specification. (Specware has as a main goal the capabilities to derive/generate low-level specifications, which is the intended mode of operation. But this capability is still in its infancy.) Here is a specification of “how” to see if a word matches a segment of a message:

```
op word_matches_at?(wrd: Word, msg: Message, pos: Nat): Boolean =
  if pos + length wrd > length msg
  then false
  else word_matches_aux?(wrd, removePrefix(msg, pos))
```

The first test checks if there is room for the word at this position. If there is room, an auxiliary function is called to check from the current position if the word matches symbol for symbol with the rest of the message. The “what” specification, the one above, needs to be related to this “how” specification. This is done by mapping between the two specifications. In this example, a transformation is used to automate the construction of the refinement relating the word matching specifications:

```
WordMatching_Ref =
  morphism MatchingSpecs#WordMatching ->
    MatchingRefinements#WordMatching {}
```

This mapping, called a *morphism*, maps the symbols in WordMatching to corresponding elements in the target specification. In this example, all the names in the high-level (source) specification (MatchingSpecs#WordMatching) are also in the low-level specification, so the task is done (otherwise the name to name mapping would be explicit within the {}).

2.4.2.3 Code synthesis

Specware can synthesize code in several different target programming languages. Currently, there are several collections of low-level executable specifications mapped directly to a program language. The most robust collection supports the Lisp programming language—a natural fit to higher-order logic specifications. Much less-mature collections exist for C and Java. The code synthesis is automatic, allowing

maintenance and enhancements to be done at the specification level rather than in code, thus precluding a number of errors produced by code changes that have unforeseen side effects. The user’s goal in the Design phase is to complete his “how” specification in terms of one of these collections. Once done, a program can be synthesized from the specification.

2.4.2.4 Proof processing

All specification, specification composition, and specification refinement constructs may require proofs to establish and maintain properties through the entire process. Proof obligations are generated when the user requests them and packaged for distribution to a proof tool. The main proof tool available in the Specware environment is Isabelle/HOL [12], which has powerful automated proof methods and integrated expert user guidance. Isabelle has a large user base, including industrial use.

Since the Isabelle theorem prover is sound, proofs completed with the tool provide assurance that the constructed code is correct with respect to the high-level specification. The Isabelle/HOL language is close to the Specware language, but with a few quirks. Isabelle is difficult to use to prove some obligations, as are all other proof tools now in use. But, even though it requires effort to use, the CxC process with Specware allows for the complete proof of the system refinement to code, yielding extremely high-assurance software.

2.4.3 Correct-by-construction successes

The CxC approach has demonstrated practical successes. Praxis used CxC to develop almost error-free code from a formal specification of an enclave access system called Tokeneer [13,14]. With Tokeneer, a user presents a token and biometric input, and then is either allowed

or denied access based on a database of metrics, token ids, and user information. The published results caused a stir in the formal methods research community, providing a much needed real world example. Kestrel developed an alternate specification of the Tokeneer system [2] using traces of events, more abstract than the state machine formalism used in Praxis's specifications, thus proving additional properties.

Another Specware success, this time from industry, is the Mathematically Analyzed Separation Kernel (MASK) [15]. This specification resulted in more easily evaluated kernel code for the Advanced INFOSEC Module chip, developed by GE (then Motorola). Also Kestrel demonstrated the benefits of formal specification reuse for IARPA. They generated an "idealized" JCRE for single threaded Multiple Independent Levels of Security (MILS) and Multiple Levels of Security (MLS), and for multi-threaded MILS and MLS separation for smart cards. Not only were they able to reuse a lot of the original specifications, but they generated run-time monitors from these specifications and proved that they enforce the separation properties.

SAT solvers have also been successfully generated using CxC tools. Formal descriptions of the SAT algorithms and SAT data structures exist in Specware, so that various versions of solvers can be generated automatically using reusable refinement scripts. With more research, even more successes could be added to the list.

3. An extreme CxC vision

Where might CxC take software development? Suppose you had a system development environment where you could take your system requirements and produce formal specifications from standard constructs like UML and state

machine diagrams. Suppose you had libraries of reusable specifications for your problem domain, for standard algorithms and procedures, and for platform-aware implementations. Consider that the mere process of exploring a design space and choosing an implementation would result in a mathematically precise implementation optimized for your platform. In addition, automate the refinement process while still allowing user control to take advantage of human expertise. We claim that such a system would revolutionize our software development practices.

- Using CxC would improve time to market: Variations of existing systems are easily obtained through minor changes to specifications and the replay of the refinement to code process.

- Using CxC would improve time and cost to certify: All design decisions are captured; hence some criteria requirements can be generated automatically rather than by manually searching the code.

- Using CxC would improve time and cost to maintain and recertify: Automation of testing and certification evidence generation made possible in this environment would solve the age-old problem of maintaining consistency between specification and code because changes are made at the specification level and the code is re-generated, keeping the two in sync.

- Using CxC would increase the degree of assurance to a qualitatively higher level.

Specware is a step in the right direction and provides evidence that such tools are possible. We need to push CxC processes that have mathematical precision if we ever hope to get a handle on the complex software and hardware systems of today. 

References and Further Reading:

- [1] Trusted computer system evaluation criteria. Department of Defense Standard 5200.28-STD; Dec 1985
Available at: <http://csrc.nist.gov/publications/history/dod85.pdf>
- [2] Common criteria for technology security evaluation, version 3.1.
 - Part 1: Introduction and general model, Revision 1, Sep 2006
 - Part 2: Security functional components, Revision 2, Sep 2007
 - Part 3: Security assurance components, Revision 2, Sep 2007
- Available at: http://www.niap-ccevs.org/cc-scheme/cc_docs/
- [3] Specware 4.2 User Manual. Kestrel Institute; 2009. Available at: <http://www.specware.org/doc.html>
- [4] Specware 4.2 Tutorial. Kestrel Institute; 2009. Available at: <http://www.specware.org/doc.html>
- [5] Specware 4.2 Language Manual. Kestrel Institute; 2009.
Available at: <http://www.specware.org/doc.html>
- [6] Specware to Isabelle Interface Manual. Kestrel Institute; 2009.
Available at: <http://www.specware.org/doc.html>
- [7] Specware 4.2 Quick Reference. Kestrel Institute; 2009.
Available at: <http://www.specware.org/doc.html>
- [8] Kestrel Institute (home page). Palo Alto (CA). Available at: <http://www.kestrel.edu>
- [9] Isabelle generic proof assistant (home page). Cambridge University, UK; 20 Sep 2010
Available at: <http://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html>
- [10] Martin W, White P, Taylor FS, Goldberg A. Formal construction of the Mathematically Analyzed Separation Kernel. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00); Sep 2000; Grenoble, France. p. 133–141.
Available at: doi: 10.1109/ASE.2000.873658
- [11] Anton J, Coglio A, McDonald J. Tokeneer. Kestrel Institute technical report; released at the 2006 High Confidence Software and Systems Conference.
- [12] Green C, Luckham D, Balzer R, Cheatham T, Rich C. Report on a knowledge-based software assistant. Kestrel Institute; 15 Jun 1983. Technical report number: KES.U.83.2.
Available at: <http://www.kestrel.edu/home/publications/>
- [13] AdaCore. The Tokeneer Project. Available at: <http://www.adacore.com/home/products/sparkpro/tokeneer/>
- [14] Barnes J, Chapman R, Johnson R, Widmaier J, Cooper D, Everett W. Engineering the Tokeneer enclave protection software. In: Proceedings of the 1st International Symposium on Secure Software Engineering (ISSSE); Mar 2006; Arlington (VA). Available at: <http://www.altran-praxis.com/downloads/SPARK/technicalReferences/issse2006tokeneer.pdf>
- [15] Escher Technologies Ltd. (home page). Aldershot, UK.
Available at: <http://www.eschertech.com/index.php>

Verified Software in the World

Software is a critical component of the technological infrastructure. Many physical and electronic devices are controlled by software, which offers unparalleled sophistication and flexibility over coding in hardware. However, software is also a source of vulnerability. Unreliable software can be a significant cost in the development of software-based systems. Software bugs can be exploited to breach security and propagate malware. Software unreliability has been estimated to cost nearly one percent of the GDP to the United States economy. The technical challenges of developing and maintaining software are only growing in complexity with the advent of cyber-physical systems, service-oriented architectures, and multicore processors.

Software development can be made highly rigorous. The theoretical understanding of software and hardware models has existed for decades, but recent dramatic advances in the technologies of software specification, design, and analysis make it feasible to carefully and productively examine large code bases for errors. Interactions between the software and the physical and biological world, as well as with human operators can be analyzed in this manner. The technologies for software analysis can also be used to find security vulnerabilities and to identify strategies for safe parallelization.

The Verified Software Initiative (VSI) is an ambitious fifteen-year, cooperative, international project directed at the scientific challenges of large-scale software verification. VSI is aimed at bringing formal scientific methods for software design into wider use so that software is viewed as the most trusted component in a system. The research agenda is directed at developing a comprehensive theory of program correctness that is supported by a coherent suite of novel and powerful tools for designing, debugging, composing, and verifying software. The theory and tools must be validated on a wide range of examples and used to train a new generation of software engineers in the construction of trustworthy software.

The need for verification technology is most acute in systems that are required to be reliable, resilient, and secure in an uncertain and hostile environment. Such systems include those from avionics, automotive control, process control, power distribution, health care, and electronic voting. These systems exhibit complex interaction between the software components and the physical world. The slightest flaw in the software can expose security vulnerabilities or lead to catastrophic system failure.

Verification approaches the construction of software through the use of rigorous formal models. These models have a mathematical meaning that is captured using formal logic. Such models can be used to capture requirements, emulate the operating environment, formulate specifications, craft designs, decompose system functionality into modules, interpret and annotate programs, generate test cases, and verify component and system properties. The use of mathematical models also facilitates the use of highly automated tools. These tools can be used to identify the presence of flaws through the systematic generation of test cases, proof obligations, interface assumptions, and security vulnerabilities. They can also establish the absence of certain kinds of errors through analysis, exploration, and proof. Finally, such tools can be used as design aids to decompose problems, derive new solutions, and compose existing solutions.

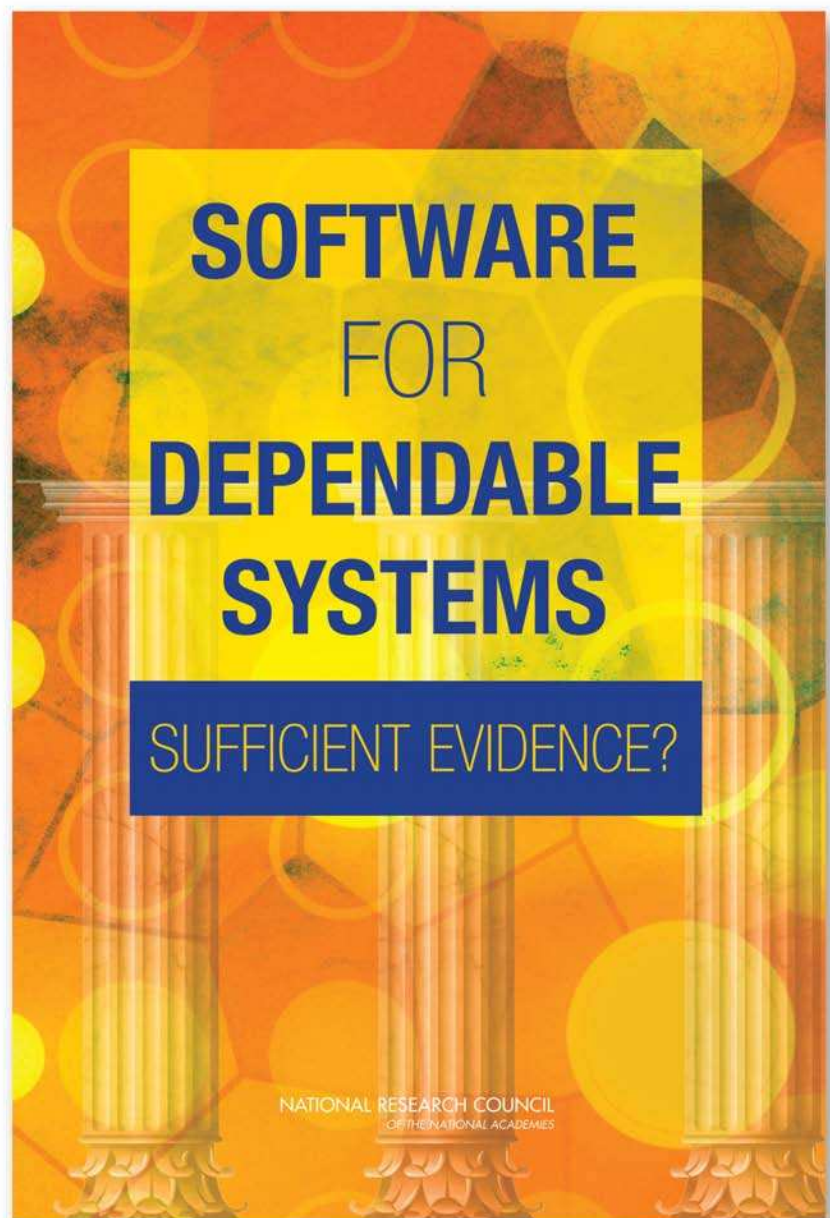
Verification technology has been improving rapidly in both scale and functionality. A range of robust and mature techniques for static and dynamic analysis, state space exploration, constraint solving, automated and interactive proof generation, and test case generation are now available and in use. These techniques can be applied to models and programs. Verification technologies need to be systematically woven into the software development process. The successful mainstreaming of verification technology requires a seamless integration of the individual techniques supported by an ambitious agenda of experimental work. Tool construction and experiments must be supported by novel theoretical insights leading to accurate and tractable models as well as scalable and efficient algorithms. Verification technology has a rapidly growing range of applications. Techniques like model checking and constraint solving are being used to model physical and biological systems and to generate plans, schedules, and optimizations. They are also used to fingerprint security threats such as worms and viruses and to check hardware and software equivalence to guard against the insertion of malicious code.

A comprehensive framework for verified software development can address a number of challenges in software engineering. At the requirements level, it provides a convenient modeling framework for describing discrete and continuous behavior, time and resource constraints, fault models, and security policies. These formal models can be analyzed for anomalies and putative properties, and also used for generating test cases. In the design phase, verification technology can be used to verify algorithms and architectures; decompose the system into modules; establish the absence of unintended information flows between software components; support semantic service discovery and composition; and facilitate resilient system operation in the face of device, platform, or operator failure. During the implementation phase, various integrated tools for synthesis and analysis can be used to generate and optimize code; establish the absence of run-time errors, race conditions, and information flows; identify interface properties; compose software modules; schedule tasks on multicore processors; and even

repair system state through constraint solving. Seamless integration between different tools is needed to generate run-time checks and monitors, test cases, counterexamples, conjectures, scenarios, abstractions, and proofs. A formal integrated development environment for verified software can be used to construct an assurance case for certification through a systematic argument for the safety and security of the system. Verification allows the assurance argument to be decomposed along the lines of components and service layers, each with its own reusable assurance case. Software is expected to operate in a safe, secure, and predictable manner in a world of physical uncertainty and virtual vulnerability. Powerful verification technology will be needed to economically develop, validate, and maintain software that is not only reliable, but manifestly trustworthy. 



Software for Dependable Systems: Sufficient Evidence?



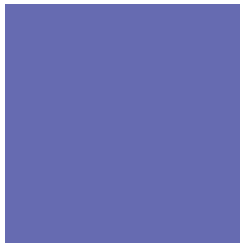
*Note: The following article is the introduction to the National Academy of Science (NAS) report, *Software for Dependable Systems: Sufficient Evidence?* Full copies of the report (free PDF download and book purchase) are available through the National Academy Press at http://www.nap.edu/catalog.php?record_id=11923*

How can software and the systems that rely on it be made dependable in a cost-effective manner, and how can one obtain assurance that dependability has been achieved? Rather than focusing narrowly on the question of software or system certification per se, this report adopts a broader perspective.

A system is dependable when it can be depended on to produce the consequences for which it was designed, and no adverse effects, in its intended environment. This means, first and foremost, that the term *dependability* has no useful meaning for a given system until these consequences and the intended environment are made explicit by a clear prioritization of the requirements of the system and an articulation of environmental assumptions. The effects of software are felt in the physical, human, and organizational environment in which it operates, so dependability should be understood in that context and cannot be reduced easily to local properties, such as resilience to crashing or conformance to a protocol. Humans who interact with the software should be viewed not as external and beyond the boundary of the software engineer's concerns but as an integral part of the system. Failures involving human operators should not automatically be

assumed to be the result of errors of usage; rather, the role of design flaws should be considered as well as the role of the human operator. As a consequence, a systems engineering approach—which views the software as one engineered artifact in a larger system of many components, some engineered and some given, and the pursuit of dependability as a balancing of costs and benefits and a prioritization of risks—is vital.

Unfortunately, it is difficult to assess the dependability of software. The field of software engineering suffers from a pervasive lack of evidence about the incidence and severity of software failures; about the dependability of existing software systems; about the efficacy of existing and proposed development methods; about the benefits of certification schemes; and so on. There are many anecdotal reports, which—although often useful for indicating areas of concern or highlighting promising



avenues of research—do little to establish a sound and complete basis for making policy decisions regarding dependability. Moreover, there is sometimes an implicit assumption that adhering to particular process strictures guarantees certain levels of dependability. The committee [NAS Committee on Certifiably Dependable Software Systems] regards claims of extraordinary dependability that are sometimes made on this basis for the most critical of systems as unsubstantiated, and perhaps irresponsible. This difficulty regarding the lack of evidence for system dependability leads to two conclusions, reflected in the committee’s findings and recommendations below: (1) that better evidence is needed, so that approaches aimed at improving the dependability of software can be objectively assessed, and (2) that, for now, the pursuit of dependability in software systems should focus on the construction and evaluation of evidence.

The committee thus subscribes to the view that software is “guilty until proven innocent,” and that the burden of proof falls on the developer to convince the certifier or regulator that the software is dependable. This approach is not novel and is becoming standard in the world of systems safety, in which an explicit safety case (and not merely adherence to good practice) is usually required. Similarly, a software system should be regarded as dependable only if it has a credible dependability case, the elements of which are described below. Meeting the burden of proof for dependability will be challenging. The demand for credible evidence will, in practice, make it infeasible to develop highly dependable systems in a cost-effective way without some radical changes in priorities. If very high dependability is to be achieved at reasonable cost, the needs of the



dependability case will influence many aspects of the development, including the choice of programming language and the software architecture, and simplicity will be key. For high levels of dependability, the evidence provided by testing alone will rarely suffice and will have to be augmented by analysis. The ability to make independence arguments that allow global properties to be inferred from an analysis of a relatively small part of the system will be essential. Rigorous processes will be needed to ensure that the chain of evidence for dependability claims is preserved.

The committee also recognized the importance of adopting the practices that are already known and used by the best developers; this summary gives a sample of such practices in more detail below. Some of these (such as systematic configuration management and automated regression testing) are relatively easy to adopt; others (such as constructing hazard analyses and threat models, exploiting formal notations when appropriate, and applying static analysis to code) will require new training for many developers. However valuable, though, these practices are in themselves no silver bullet, and new techniques and methods will be required in order to build future software systems to the level of dependability that will be required.

Assessment

Society is increasingly dependent on software. Software failures can cause or contribute to serious accidents that result in death, injury, significant environmental damage, or major financial loss. Such accidents have already occurred, and, without intervention, the increasingly pervasive use of software—especially in arenas such as transportation, health care, and the broader infrastructure—may make them more frequent and more serious. In the future, more pervasive deployment of software in the civic infrastructure could lead to more catastrophic failures unless improvements are made.

Software, according to a popular view, fails because of bugs: errors in the code that cause a program to fail to meet its specification. In fact, only a tiny proportion of failures can be attributed to bugs. As is well known to software engineers, by far the largest class of problems arises from errors made in the eliciting, recording, and analysis of requirements. A second major class of problems arises from poor human factors design. The two classes are related; bad user interfaces usually reflect an inadequate understanding of the user’s domain and the absence of a coherent and well-articulated conceptual model. Security vulnerabilities are to some extent an exception to this observation: The overwhelming majority of security vulnerabilities reported in

software products—and exploited to attack the users of such products—are at the implementation level. The prevalence of code-related problems, however, is a direct consequence of higher-level decisions to use programming languages, design methods, and libraries that admit these problems.

In systems where software failure could have significant human or financial costs, it is crucial that software be dependable—that it can be depended upon to function as expected and to not cause or contribute to adverse events in the environment in which it operates. Improvements in dependability would allow such systems to be used more widely and with greater confidence for the benefit of society. Moreover, software itself has great potential to bring improvements in safety in many areas.

Complete and reliable data about software-related system failures or the efficacy of particular software development approaches are hard to come by, making objective scientific evaluation difficult. Moreover, the lack of systematic reporting of software-related system failures is a serious problem that makes it more difficult to evaluate the risks and costs of such failures and to measure the effectiveness of proposed policies or interventions.

This lack of evidence has two direct consequences for this report. First, it has informed the key recommendations in this report regarding the need for evidence to be at the core of dependable software system development; for data collection efforts to be established; and for transparency and openness to be encouraged. Second, it has tempered the committee's desire to provide prescriptive guidance: The approach recommended is therefore largely free of endorsements or criticisms

of particular development approaches, tools, or techniques. Moreover, the report leaves to the developers and procurers of individual systems the question of what level of dependability is appropriate, and what costs are worth incurring to achieve it.

Nonetheless, the evidence available to the committee did support several qualitative conclusions. First, developing software to meet even existing dependability criteria is difficult and costly. Large software projects fail at a high rate, and the cost of projects that do succeed in delivering highly dependable software is often exorbitant. Second, the quality of software produced by the industry is extremely variable, and there is inadequate oversight in some critical areas. Today's certification regimes and consensus standards have a mixed record. Some are largely ineffective, and some are counterproductive. They share a heavy reliance on testing, which cannot provide sufficient evidence for the high levels of dependability required in many critical applications.

A final observation is that the culture of an organization in which software is produced can have a dramatic effect on its quality and dependability. It seems likely that the excellent record of avionics software is due in large part to a safety culture in that industry that encourages meticulous attention to detail, high aversion to risk, and realistic assessment of software, staff, and process. Indeed, much of the benefit of standards such as DO-178B, Software Considerations in Airborne Systems and Equipment Certification, may be due to the safety culture that their strictures induce.

Toward certifiably dependable software

The focus of this report is a set of fundamental principles that underlie

software system dependability and that suggest a different approach to the development and assessment of dependable software. Due to a lack of sufficient data to support or contradict any particular approach, a software system may not be declared “dependable” based on the method by which it was constructed. Rather, it should be regarded as dependable—certifiably dependable—only when adequate evidence has been marshaled in support of an argument for dependability that can be independently assessed. The goal of certifiably dependable software cannot therefore be achieved by mandating particular processes and approaches, regardless of their effectiveness in certain situations. Instead, software developers should marshal evidence to justify an explicit dependability claim that makes clear which properties in the real world the system is intended to establish. Such evidence forms a dependability case, and creating a dependability case is the cornerstone of the committee's approach to developing certifiably dependable software systems.

Explicit claims, evidence, and expertise

The committee's proposed approach can be summarized in “the three Es”—explicit claims, evidence, and expertise:

- *Explicit claims.* No system can be “dependable” in all respects and under all conditions. So to be useful, a claim of dependability must be explicit. It must articulate precisely the properties the system is expected to exhibit and the assumptions about the system's environment upon which the claim is contingent. The claim should also indicate explicitly the level of dependability claimed, preferably in quantitative terms.

Different properties may be assured to different levels of dependability.

- *Evidence.* For a system to be regarded as dependable, concrete evidence must be present that substantiates the dependability claim. This evidence will take the form of a dependability case arguing that the required properties follow from the combination of the properties of the system itself (that is, the implementation) and the environmental assumptions. Because testing alone is usually insufficient to establish properties, the case will typically combine evidence from testing with evidence from analysis. In addition, the case will inevitably involve appeals to the process by which the software was developed—for example, to argue that the software deployed in the field is the same software that was subjected to analysis or testing.

- *Expertise.* Expertise—in software

development, in the domain under consideration, and in the broader systems context, among other things—is necessary to achieve dependable systems. Flexibility is an important advantage of the proposed approach; in particular the developer is not required to follow any particular process or use any particular method or technology. This flexibility allows experts freedom to employ new techniques and to tailor the approach to the system’s application and domain. But the requirement to produce evidence is highly demanding and likely to stretch today’s best practices to their limit. It will therefore be essential that developers are familiar with best practices and deviate from them only for good reason.

These prescriptions shape any particular development approach only in outline and give considerable freedom to developers in their choice of methods, languages, tools, and processes. This approach is not, of course, a silver bullet. There are no easy solutions to the problem of developing dependable software, and there will always be systems that cannot be built to the required level of dependability even using the latest methods. But, the approach recommended is aimed at producing certifiably dependable systems today, and the committee believes it holds promise for developing the systems that will be needed in the future.

In the overall context of engineering, the basic tenets of the proposed approach are not controversial, so it may be a surprise to some that the approach is not already commonplace. Nor are the elements of the approach novel; they have been applied successfully for more than a decade. Nevertheless, this approach would require radical changes for most software development organizations and is likely to demand expertise that is currently in short supply.

Systems engineering approach

Complementing “the three Es” are several systems engineering ideas that provide an essential foundation for the building of dependable software systems:

- *Systems thinking.* Engineering fields with long experience in building complex systems (for example, aerospace, chemical, and nuclear engineering) have developed approaches based on “systems thinking.” These approaches focus on properties of the system as a whole and on the interactions among its components, especially those interactions (often neglected) between a component being constructed and the components of its environment. As software has come to be deployed in—indeed has enabled—increasingly complex systems, the system aspect has come to dominate in questions of software dependability.

- *Software as a system component.* Dependability is not an intrinsic property of software. The committee strongly endorses the perspective of systems engineering, which views the software as one engineered artifact in a larger system of many components, some engineered and some given, and views the pursuit of dependability as a balancing of costs and benefits and a prioritization of risks. A software component that may be dependable in the context of one system might not be dependable in the context of another.

- *Humans as components.* People—the operators and users (and even the developers and maintainers) of a system—may also be viewed as system components. If a system meets its dependability criteria only if people act in certain ways, then those people should be regarded as part of the system, and an estimate of the probability that they will



behave as required should be part of the evidence for dependability.

- **Real-world properties.** The properties of interest to the user of a system are typically located in the physical world: that a radiotherapy machine deliver a certain dose, that a telephone transmit a sound wave faithfully, that a printer make appropriate ink marks on paper, and so on. The software, on the other hand, is typically specified in terms of properties at its interfaces, which usually involve phenomena that are not of direct interest to the user: that the radiotherapy machine, telephone, or printer send or receive certain signals at certain ports, with the inputs related to the outputs according to some rules. It is important, therefore, to distinguish the requirements of a software system, which represent these properties in the physical world, from the specification of a software system, which characterizes the behavior of the software system at its interface with the environment. When the software system is itself only one component of a larger system, the other components in the system (including perhaps, as explained above, the people who work with the system) will be viewed as part of the environment. The dependability properties of a software system, therefore, should be expressed as requirements, and the dependability case should demonstrate how these properties follow from the combination of the specification and the environmental assumptions.

Coping with complexity

The need for evidence of dependability and the difficulty of producing such evidence for complex systems have a straightforward but

profound implication. Any component for which compelling evidence of dependability has been amassed at reasonable cost will likely be small by the standards of most modern software systems. Every critical specification property, therefore, will have to be assured by one, or at most a few, small components. Sometimes it will not be possible to separate concerns so cleanly, and in that case, the dependability case may be less credible or more expensive to produce.

As a result, one key to achieving dependability at reasonable cost is a serious and sustained commitment to simplicity, including simplicity of critical functions and simplicity in system interactions. This commitment is often the mark of true expertise. An awareness of the need for simplicity usually comes only with bitter experience and the

*“Testing is indispensable,
and no software system can be regarded as dependable
if it has not been extensively tested...”*

humility gained from years of practice. There is no alternative to simplicity. Advances in technology or development methods will not make simplicity redundant; on the contrary, they will give it greater leverage. To achieve high levels of dependability in the foreseeable future, striving for simplicity is likely to be by far the most cost-effective of all interventions. Simplicity is not easy or cheap, but its rewards far outweigh its costs.

The most important form of simplicity is that produced by independence, in which particular system-level properties are guaranteed by

individual components much smaller than the system as a whole, which can preserve these properties despite failures in the rest of the system. Independence can be established in the overall design of the system, with the support of architectural mechanisms. Its effect is to dramatically reduce the cost of constructing a dependability case for a property, since only a relatively small part of the system needs to be considered.

Appropriate simplicity and independence cannot be accomplished without addressing the challenges of “interactive complexity” and “tight coupling.” Both interactive complexity, where components may interact in unanticipated ways, and tight coupling, wherein a single fault cannot be isolated but brings about other faults that cascade through the system, are correlated with the likelihood of system failure.

Software-intensive systems tend to have both attributes. Careful attention should therefore be paid to the risks of interactive complexity and tight coupling and the advantages of modularity, isolation, and redundancy. The interdependences among components of critical software systems should be analyzed to ensure that there is no fault propagation path from less critical components to more critical components, that modes of failure are well understood, and that failures are localized to the greatest extent possible. The reduction of interactive complexity and tight coupling can contribute not

only to the improvement of system dependability but also to the development of evidence and analysis in the service of a dependability case.

Rigorous process and preserving the chain of evidence

Generating a dependability case after the fact, when a development is largely complete, might be possible in theory. But in practice, at least with today's technology, the costs of doing so would be high, and it will be practical to develop a dependability case only if the system is built with its construction in mind. Each step in developing the software needs to preserve the chain of evidence on which will be based the argument that the resulting system is dependable.

At the start, the domain and environmental assumptions and the required properties of the system should be made explicit; they should be expressed unambiguously and in a form that permits systematic analysis to ensure that there are no unresolvable conflicts between the required properties. Each subsequent stage of development should preserve the evidence chain—that these properties have been carried forward without being corrupted—so each form in which the requirements, design, or implementation is expressed should support analysis to permit checking that the required properties have been preserved. What is sufficient will vary with the required dependability, but preserving the evidence chain necessitates that the checks are carried out in a disciplined way, following a documented procedure, and leaving auditable records.

The roles of testing, analysis, and formal methods

Testing is indispensable, and no software system can be regarded as dependable if it has not been extensively

tested, even if its correctness has been proven mathematically. Testing may find flaws that elude analysis because it exercises the system in its entirety, whereas analysis must typically make assumptions about the execution platform, the external environment, and operator responses, any of which may turn out to be unwarranted. At the same time, it is important to realize that testing alone is rarely sufficient to establish high levels of dependability. It is erroneous to believe that a rigorous development process, in which testing and code review are the only verification techniques used, justifies claims of extraordinarily high levels of dependability. Some certification schemes, for example, associate higher safety integrity levels with more burdensome process prescriptions and imply that following the processes recommended for the highest integrity levels will ensure that the failure rate is minuscule. In the absence of a carefully constructed dependability case, such confidence is misplaced.

Because testing alone will not be sufficient for the foreseeable future, the dependability claim will also require evidence produced by analysis. Moreover, because analysis links the software artifacts directly to the claimed properties, the analysis component of the dependability case will usually contribute confidence at a lower cost than testing for the highest levels of dependability. A dependability case will generally require many forms of analysis, including (1) the validation of environmental assumptions, use models, and fault models; (2) the analysis of fault tolerance measures against fault models; (3) schedulability analysis for temporal behaviors; (4) security analysis against attack models; (5) verification of code against module specifications; and (6) checking that

modules in aggregate achieve appropriate system-level effects. These analyses will sometimes involve informal argument that is carefully reviewed; sometimes mechanical inference (as performed, for example, by “type checkers” that confirm that memory is used in a consistent way and that boundaries between modules are respected); and, sometimes, formal proof. Indeed, the dependability case for even a relatively simple system will usually require all of these kinds of analysis, and they will need to be fitted together into a coherent whole.

Traditional software development methods rely on human inspection and testing for validation and verification. Formal methods also use testing, but they employ notations and languages that are amenable to rigorous analysis, and they exploit mechanical tools for reasoning about the properties of requirements, specifications, designs, and code. Practitioners have been skeptical about the practicality of formal methods. Increasingly, however, there is evidence that formal methods can yield systems of very high dependability in a cost-effective manner, at least for small- to medium-sized critical systems. Although formal methods are typically more expensive to apply when only low levels of dependability are required, the cost of traditional methods rises rapidly with the level of dependability and often becomes prohibitive. When a highly dependable system is required, therefore, a formal approach may be the most cost effective.

Certification, transparency, and accountability

A variety of certification regimes exist for software in particular application domains. For example, the Federal Aviation Authority (FAA) itself certifies

new aircraft (and air-traffic management) systems that include software, and this certification is then relied on by the customers who buy and use the aircraft; the National Information Assurance Partnership (NIAP) licenses third-party laboratories to assess security software products for conformance to the Common Criteria. Some large organizations have their own regimes for certifying that the software products they buy meet the organization's quality criteria, and many software product manufacturers have their own criteria that each version of their product must pass before release.

Few, if any, existing certification regimes encompass the combination of characteristics recommended in this report—namely, explicit dependability claims, evidence for those claims, and a rigorous argument that demonstrates that the evidence is sufficient to establish the validity of the claims. To establish that a system is dependable will involve inspection and analysis of the dependability claim and the evidence offered in its support. Where the customer for the system is not able to carry out that work itself (for lack of time or lack of expertise) it may need to involve a third party whose judgment it can rely on to be independent of commercial pressures from the vendor. Certification can take many forms, from self-certification by the supplier at one extreme, to independent third-party certification by a licensed certification authority at the other. No single certification regime is suitable for all circumstances, so a suitable scheme should be chosen for each circumstance. Industry groups and professional societies should consider developing model certification schemes appropriate to their domains, taking account of the detailed recommendations in this report.

When choosing suppliers and products, customers and users can make informed judgments only if the claims are credible. Such claims are unlikely to be credible if the evidence underlying them is not transparent. Economists have established that if consumers cannot reliably observe quality before they buy, sellers may get little economic benefit from providing higher quality than their competitors, and overall quality can decline. Sellers are concerned about future sales, and “reputation effects” compel them to strive to maintain a minimum level of quality. If consumers rely heavily on branding, though, it becomes more difficult for new firms to enter the market, and quality innovations spread more slowly.

Those claiming dependability for their software should therefore make available the details of their claims, criteria, and evidence. To assess the credibility of such details effectively, an evaluator should be able to calibrate not only the technical claims and evidence but also the organization that produced them, because the integrity of the evidence chain is vital and cannot easily be assessed without supporting data. This suggests that in some cases data of a more general nature should be made available, including the qualifications of the personnel involved in the development; the track record of the organization in providing dependable software; and the process by which the software was developed. The willingness of a supplier to provide such data, and the clarity and integrity of the data that the supplier provides, will be a strong indication of its attitude to dependability.

Where there is a need to deploy software that satisfies a particular dependability claim, it should always be

explicit who is accountable for any failure to achieve it. Such accountability can be made explicit in the purchase contract, or as part of certification of the software, or as part of a professional licensing scheme, or in other ways. Since no single solution will suit all the circumstances in which certifiably dependable software systems are deployed, accountability regimes should be tailored to particular circumstances. At present, it is common for software developers to disclaim, so far as possible, all liability for defects in their products, to a greater extent than customers and society expect from manufacturers in other industries. Clearly, no software should be considered dependable if it is supplied with a disclaimer that withholds the manufacturer's commitment to provide a warranty or other remedies for software that fails to meet its dependability claims. Determining the appropriate scale of remedies, however, was beyond the scope of this study and would require a careful analysis of benefits and costs, taking into account not only the legal issues but also the state of software engineering, the various submarkets for software, the economic impact, and the effect on innovation.

Key findings and recommendations

Presented below are the committee's findings and recommendations, each of which is discussed in more detail in Chapter 4. (The full report is available at: http://www.nap.edu/catalog.php?record_id=11923)

Findings

Improvements in software development are needed to keep pace with societal demands for software. Avoidable software failures have already

been responsible for loss of life and for major economic losses. The quality of software produced by the industry is extremely variable, and there is inadequate oversight in several critical areas. More pervasive deployment of software in the civic infrastructure may lead to catastrophic failures unless improvements are made. Software has the potential to bring dramatic benefits to society, but it will not be possible to realize these benefits—especially in critical applications—unless software becomes more dependable.

More data is needed about software failures and the efficacy of development approaches. Assessment of the state of the software industry, the risks posed by software, and progress made is currently hampered by the lack of a coherent source of information about software failures.

Recommendations to builders and users of software

Make the most of effective software development technologies and formal methods. A variety of modern technologies—in particular, safe programming languages, static analysis (analysis of software and source code done without actually executing the program), and formal methods—are likely to reduce the cost and difficulty of producing dependable software.

Follow proven principles for software development. The committee's proposed approach also includes adherence to the following principles:

- *Take a systems perspective.* Here the dependability of software is viewed not in terms of intrinsic properties (such as the incidence of bugs in the code) but rather in terms of the system as a whole, including interactions among people, process, and technology.

- *Exploit simplicity.* If dependability is to be achieved at reasonable cost, simplicity should become a key goal, and developers and customers must be willing to accept the compromises it entails.

Make a dependability case for a given system and context: evidence, explicitness, and expertise. A software system should be regarded as dependable only if sufficient evidence of its explicitly articulated properties is presented to substantiate the dependability claim. This approach gives considerable leeway to developers to use whatever practices are best suited to the problem at hand. In practice the challenges of developing dependable software are sufficiently great that developers will need considerable expertise, and they will have to justify any deviations from best practices.

Demand more transparency, so that customers and users can make more informed judgments about dependability. Customers and users can make informed judgments when choosing suppliers and products only if the claims, criteria, and evidence for dependability are transparent.

Make use of but do not rely solely on process and testing. Testing will be an essential component of a dependability case, but will not in general suffice, because even the largest test suites typically used will not exercise enough paths to provide evidence that the software is correct nor will it have sufficient statistical significance for the levels of confidence usually desired. Rigorous process is essential for preserving the chain of dependability evidence but is not per se evidence of dependability.

Base certification on inspection and analysis of the dependability claim and the evidence offered in its support. Because testing and process alone are

insufficient, the dependability claim will require, in addition, evidence produced by other modes of analysis. Security certification in particular should go beyond functional testing of the security components of a system and assess the effectiveness of measures the developer took to prevent the introduction of security vulnerabilities.

Include security considerations in the dependability case. Security vulnerabilities can undermine the case made for dependability properties by violating assumptions about how components behave, about their interactions, or about the expected behavior of users. The dependability case must therefore account explicitly for security risks that might compromise its other aspects. It is also important to ensure that security certifications give meaningful assurance of resistance to attack. New security certification regimes are needed that can provide confidence that most attacks against certified products or systems will fail. Such regimes can be built by applying the other findings and recommendations of this report, with an emphasis on the role of the environment—in particular, the assumptions made about the potential actions of a hostile attacker and the likelihood that new classes of vulnerabilities will be discovered and new attacks developed to exploit them.

Demand accountability and make it explicit. Where there is a need to deploy certifiably dependable software, it should always be made explicit who or what is accountable, professionally and legally, for any failure to achieve the declared dependability.

Recommendations to agencies and organizations that support software education and research

The committee was not constituted or charged to recommend budget levels

or to assess trade-offs between software dependability and other priorities. However, it believes that the increasing importance of software to society and the extraordinary challenge currently faced in producing software of adequate dependability provide a strong rationale for investment in education and research initiatives.

Place greater emphasis on dependability—and its fundamental underpinnings—in the high school, undergraduate, and graduate education of software developers. Many practitioners do not have an adequate appreciation of the software dependability issues discussed in this report, are not aware of the most effective development practices available today, or are not capable of applying them appropriately. Wider implementation of the committee's recommended approach, which goes beyond today's state of the practice, implies a need for further education and training activities.

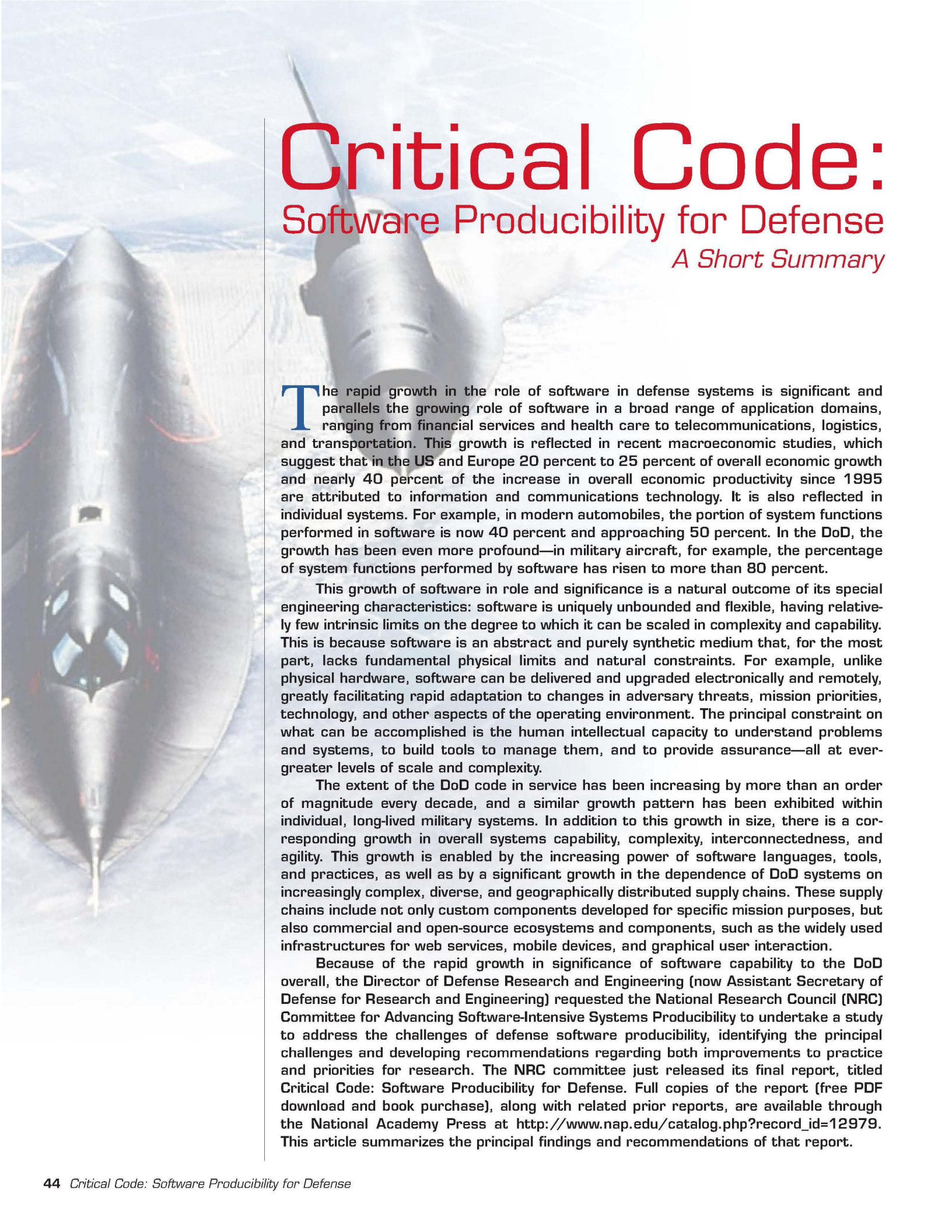
Federal agencies that support information technology research and development should give priority to basic research to further software-enabled system dependability, emphasizing a systems perspective and evidence. In keeping with this report's approach, such research should emphasize a systems perspective and “the three Es” (explicit claims, evidence, and expertise) and should be informed by a systems view that attaches more importance to those advances that are likely to have an impact in a world of large systems interacting with other systems and operators in a complex physical environment and organizational context. 📌

About the report

This report was authored by the National Research Council's (NRC) **Committee on Certifiably Dependable**

Software Systems, convened under the auspices of the NRC's Computer Science and Telecommunications Board. The committee consisted of 13 experts from industry and academia specializing in diverse aspects of systems dependability including software engineering, software testing and evaluation, software dependability, embedded systems, human-computer interaction, systems engineering, systems architecture, accident theory, standards setting, avionics, medicine, economics, security, and regulatory policy. Committee chair **Daniel Jackson**, a professor of Computer Science at MIT; committee member Martyn Thomas, visiting professor of software engineering at Oxford University; and Lynette Millett, senior staff officer at the NRC, edited the report.

Discussions initiated by the **High Confidence Software and Systems Coordinating Group (HCSS CG)** of the National Science and Technology Council's Networking and Information Technology Research and **Development (NITRD) Subcommittee** with the NRC's Computer Science and Telecommunications Board resulted in this study on the current state of certification in dependable systems. Funding for the study was obtained from HCSS CG member agencies.



Critical Code:

Software Producibility for Defense

A Short Summary

The rapid growth in the role of software in defense systems is significant and parallels the growing role of software in a broad range of application domains, ranging from financial services and health care to telecommunications, logistics, and transportation. This growth is reflected in recent macroeconomic studies, which suggest that in the US and Europe 20 percent to 25 percent of overall economic growth and nearly 40 percent of the increase in overall economic productivity since 1995 are attributed to information and communications technology. It is also reflected in individual systems. For example, in modern automobiles, the portion of system functions performed in software is now 40 percent and approaching 50 percent. In the DoD, the growth has been even more profound—in military aircraft, for example, the percentage of system functions performed by software has risen to more than 80 percent.

This growth of software in role and significance is a natural outcome of its special engineering characteristics: software is uniquely unbounded and flexible, having relatively few intrinsic limits on the degree to which it can be scaled in complexity and capability. This is because software is an abstract and purely synthetic medium that, for the most part, lacks fundamental physical limits and natural constraints. For example, unlike physical hardware, software can be delivered and upgraded electronically and remotely, greatly facilitating rapid adaptation to changes in adversary threats, mission priorities, technology, and other aspects of the operating environment. The principal constraint on what can be accomplished is the human intellectual capacity to understand problems and systems, to build tools to manage them, and to provide assurance—all at ever-greater levels of scale and complexity.

The extent of the DoD code in service has been increasing by more than an order of magnitude every decade, and a similar growth pattern has been exhibited within individual, long-lived military systems. In addition to this growth in size, there is a corresponding growth in overall systems capability, complexity, interconnectedness, and agility. This growth is enabled by the increasing power of software languages, tools, and practices, as well as by a significant growth in the dependence of DoD systems on increasingly complex, diverse, and geographically distributed supply chains. These supply chains include not only custom components developed for specific mission purposes, but also commercial and open-source ecosystems and components, such as the widely used infrastructures for web services, mobile devices, and graphical user interaction.

Because of the rapid growth in significance of software capability to the DoD overall, the Director of Defense Research and Engineering (now Assistant Secretary of Defense for Research and Engineering) requested the National Research Council (NRC) Committee for Advancing Software-Intensive Systems Producibility to undertake a study to address the challenges of defense software producibility, identifying the principal challenges and developing recommendations regarding both improvements to practice and priorities for research. The NRC committee just released its final report, titled *Critical Code: Software Producibility for Defense*. Full copies of the report (free PDF download and book purchase), along with related prior reports, are available through the National Academy Press at http://www.nap.edu/catalog.php?record_id=12979. This article summarizes the principal findings and recommendations of that report.

The necessity of sustaining software innovation

An initial question is whether software is indeed a strategic building material, worthy of special attention. This question has been addressed periodically by the Defense Science Board (DSB) since 1985—a 2007 DSB report, for example, stated that “in the Department of Defense, the transformational effects of information technology (IT—defined here broadly to include all forms of computing and communications), joined with a culture of information sharing, called Net-Centricity, constitute a powerful force multiplier. The DoD has become increasingly dependent for mission-critical functionality upon highly interconnected, globally sourced IT of dramatically varying quality, reliability, and trustworthiness.”

Despite the strength of this statement, every few years speculation surfaces that perhaps software and information technology may be approaching a plateau of capability and performance and that strategic attention to these technologies is consequently not merited. The committee emphasizes that this continues to be a false and dangerous speculation—the capability and the complexity of hardware and software systems are both rising at an accelerating rate, with no end in sight.

It is instructive, in this regard, to consider the publication in 1958—more than a half century ago—of the landmark

paper by John Backus describing the first Fortran compiler. The title included the words “automatic programming.” The point of this phrase, with respect to Backus’s great accomplishment, is that there was a much more direct correspondence between his high-level programming notation—the earliest Fortran code—and pure mathematical thinking than had been the case with the early machine-level code. One can construe that it was imagined that Fortran enabled mathematicians to express their thoughts directly to computers, seemingly without the intervention of programmers. The early Fortran was indeed an extraordinary and historical breakthrough. But we know that, in the end, those mathematicians of 50 years ago soon evolved into programmers—as a direct consequence of their growing ambitions for computing applications.

Just a few years after the Backus paper, Fortran was used to support list-processing applications, typesetting applications, compilers for other languages, and other applications whose abstractions required some considerable programming sophistication (and representational gerrymandering) to be represented effectively as early Fortran data structures—arrays and numeric values. Any program that manipulated textual data, for example, needed to encode the text characters, textual strings, and any overarching paragraph and

document structure very explicitly into numbers and arrays. A person reading program text would see only numerical and array operations because that was the limit of what could be explicitly expressed in the notation. This meant that programmers needed to keep track, in their heads or in documentation, of the nature of this representational encoding. It also meant that testers and evaluators needed to assess programs through this (hopefully) same layer of interpretation.

As languages have evolved (including more modern Fortran versions), these additional structures can be much more directly expressed—characters and strings, most obviously, are intrinsic in nearly all modern languages. It is interesting, however, that the claim of “automatic programming” continues to reappear from time to time as major steps are made in improved abstractions, for example related to data manipulation (the so-called 4GLs). These developments move us forward, but ironically they do not actually get us closer to “eliminating programmers” or otherwise emerging at some plateau of capability and near-commodity status. Instead, new software-manifest capabilities are constantly emerging—for example, techniques for machine-learning algorithms and highly parallel data-intensive analytics—that continue to demand considerable intellectual effort on the part of programmers.

The profound fact is that software capability is bounded primarily by our intellectual abilities—our human capability both to create new abstractions appropriate for application domains and to manifest those abstractions in languages, models, tools, and practices. As our understanding advances, so can our software capability advance with us.

As a consequence of this seeming unboundedness, the committee finds that *technological leadership in software is a key driver of overall capability leadership in systems*—and that at the core of the ability to achieve integration and maintain mission agility is the ability of the DoD to produce and evolve software. The committee recommends that, to avoid loss of leadership, the DoD take active

on mainstream commercial and open source components, supply chains, and software ecosystems, it nonetheless has special needs in its mission systems that are driven by the growing role of software in systems overall. The committee recommends that the DoD regularly **undertake an identification of areas of technological need** where the DoD has “leading demand” and where accelerated progress is needed.

Three goals for software-intensive development

The committee identified three areas where improvements in practice would materially benefit the ability of the DoD to develop, sustain, and assure software-intensive systems of all kinds. Each of these areas is the subject of a chapter in

made within an engineering process—the risks are high when the outcomes of immediate project commitments are both consequential and difficult to predict. Engineering risks can relate to many different kinds of engineering decisions—most significantly architecture, quality attributes, functional characteristics, and infrastructure choices.

When well managed, incremental practices can enable innovative engineering to be accomplished without a necessarily consequent increase of overall programmatic risk. (*Programmatic risk* relates to the successful completion of engineering projects with respect to expectations and priorities for cost, schedule, capability, quality, and other attributes.) This is because incremental practices enable engineering risks to be identified early and mitigated promptly. Incremental practices are enabled through the use of diverse techniques such as modeling, simulation, prototyping, and other means for early validation—coupled with extensions to earned-value models that measure and give credit for the accumulating body of evidence in support of feasibility. Incremental approaches include iterative approaches, staged acquisition, evidence-based systems engineering, and other methods that explicitly acknowledge engineering risk and its mitigation.

The committee finds that incremental and iterative methods are of fundamental significance to DoD for innovative, software-intensive engineering in the DoD, and they can be managed more effectively through improvements in practices and supporting tools. The committee recommends a diverse set of improvements related to advanced incremental development practice, supporting tools, and earned-value models.

“...to avoid loss of leadership, the DoD [should] take active steps to become more fully engaged in the innovative processes related to software producibility.”

steps to become more fully engaged in the innovative processes related to software producibility. In particular, the committee finds that industry, despite the extraordinary pace of innovation we are now witnessing, will not produce software innovations in areas of defense significance at a rate fast enough to allow the DoD to fully meet its software-related requirements and remain ahead of potential adversaries.

A loss of leadership could threaten the ability of the DoD to manifest world-leading capability, and also to achieve adequate levels of assurance for the diversely sourced software it intends to deploy. This is an important part of the rationale for the committee recommendation that the DoD reengage directly in the innovation processes.

The committee also finds that although the DoD relies fundamentally

the Critical Code report. (These three areas of practice correspond to Chapters 2, 3, and 4. Chapter 1 of the report focuses on the necessary role of DoD in software innovation. Chapter 5 summarizes the research agenda related to software producibility.) The three areas of practice are summarized below:

Practice improvement 1: Process and measurement

Advances related to process and measurement would facilitate broader and more effective use of incremental iterative development, particularly in the arms-length contracting situations common in DoD.

Incremental development practices enable continuous identification and mitigation of *engineering risks* during a systems development process. Engineering risks pertain to the consequences of particular choices to be

Practice improvement 2: Architecture

Advances related to architecture practice would facilitate the early focus on systems architecture that is essential particularly for systems with demanding requirements related to quality attributes, interlinking, and planned flexibility.

Software architecture models the structures of a system that comprises software components, the externally visible properties of those components, and the relationships among the components. Good architecture entails a minimum of engineering commitment that yields a maximum value. In particular, architecture design is an engineering activity that is separate, for example, from ecosystems certification and other standards-related policy setting.

For complex innovative systems, architecture definition embodies planning for flexibility—defining and encapsulating areas where innovation and change are anticipated. Architecture definition also most strongly influences diverse quality attributes, ranging from availability and performance to security and isolation. Additionally, architecture embodies planning for the interlinking of systems and for product line development, enabling encapsulation of individual innovative elements of a system.

For many innovative systems, therefore, it may be more effective to consider architecture and quality attributes before making specific commitments to functionality. Because architecture includes the earliest and, often, the most important design decisions—those engineering costs that are most difficult to change later—early architectural commitment (and validation) can yield better project outcomes with less programmatic risk.

The committee finds that in highly complex systems with emphasis on

quality attributes, architecture decisions may dominate functional capability choices in overall significance. The committee also notes that architecture practice in many areas of industry is sufficiently mature for DoD to adopt. The committee recommends that DoD more aggressively assert architectural leadership, with an early focus on architecture being essential for systems with innovative functionality or demanding quality requirements.

Practice improvement 3: Assurance and security

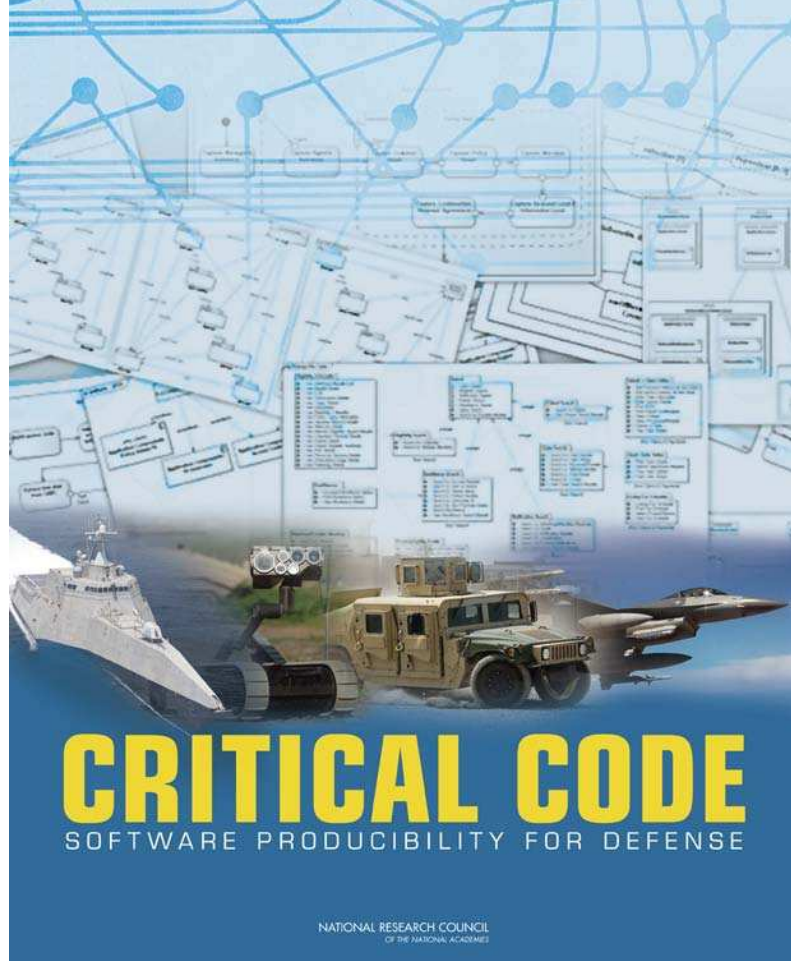
Advances related to assurance and security would facilitate achievement of mission assurance for systems at greater degrees of scale and complexity, and in the presence of rich supply chains and architectural ecosystems that are increasingly commonplace in modern software engineering.

Assurance is a human judgment regarding not just functionality, but also diverse quality attributes related to reliability, security, safety, and other

system characteristics. The weights given the various attributes are typically determined on the basis of models of hazards associated with the operational context, including potential threats. The process of achieving software assurance, regardless of sector, is generally recognized to account for approximately half the total development cost for major projects.

In addition to overall cost, DoD faces several particular challenges for assurance. First, there is often an arms-length relationship between a contractor development team and government stakeholders, making it difficult to develop and share the information necessary to making assurance judgments. This can lead to approaches that overly focus on *post hoc* acceptance evaluation, rather than on the emerging practice of “building in” evidence in support of an overall assurance case. Second, modern systems draw on components from diverse sources. This implies that supply-chain and configuration-related attacks must be contemplated, with “attack surfaces” existing within an overall application,





and not just at its perimeter. This has the consequence that evaluative and preventive approaches ideally must be integrated throughout a complex supply chain. A particular challenge is managing opaque, or “black box,” components in a system—this issue is addressed in the full report. Third, the growing role of DoD software in warfighting, in protection of national assets, and in the safeguarding of human lives creates a diminishing tolerance for faulty assurance judgments. Indeed, the Defense Science Board notes that there are profound risks associated with the increasing reliance on modern software-intensive systems: “this growing dependency is a source of weakness exacerbated by the mounting size, complexity, and interconnectedness of its software programs.” Fourth, losing the lead in the ability to evaluate software and to prevent attacks can confer advantage to adversaries with respect to both offense and defense. It can also force us to overly “dumb down” systems, restricting functionality or performance to a level such that assurance judgments can be more readily achieved.

The Defense Science Board found in 2007 that “it is an essential requirement that the United States maintain advanced capability for ‘test and evaluation’ of IT products. Reputation-based or trust-based credentialing of software (‘provenance’) needs to be augmented by direct, artifact-focused means to support acceptance evaluation.” This is a significant challenge, due to the rapid advance of software technology generally and also the increasing pace by which potential adversaries are advancing their capability. This, coupled with the observations above regarding software innovation, is an important part of the rationale for the committee recommendation that the DoD actively and directly address its software producibility needs.

In the full report, the committee addressed a broad range of issues related to software assurance, including evidence-based approaches, evaluation practices, and security-motivated challenges related to configuration integrity (particularly in the presence of dynamism) and separation (including isolation and sandboxing).

The committee notes that traditional approaches based purely on testing and inspection, no matter how extensive, are often insufficiently effective for modern software systems. It emphasizes that evaluation practices that focus primarily on *post hoc* acceptance evaluation are not only very costly but are often insufficient to justify useful assurance judgments. That is, quality and security must be built in, and not “tested in”—with the consequence that evidence production in support of assurance must be integrated into software development.

The committee finds that assurance is facilitated by advances in diverse aspects of software engineering practice and technology, including modeling, analysis, tools and environments, traceability and configuration management, programming languages, and process support. The committee also finds that, after many years of slow progress, recent advances have enabled more rapid improvement in assurance-related techniques and tools. This is already evident in the most advanced commercial development practice. The committee also finds that simultaneous creation of assurance-related evidence with ongoing development has high potential to improve the overall assurance of systems. The committee recommends enhancing incentives for preventive software assurance practices and production of assurance-related evidence throughout the software lifecycle and through the software supply chain. This includes both contractor and in-house development efforts.

The challenge of DoD software expertise

The committee also took up the issue of software expertise that is specifically aligned with DoD interests. The committee found that DoD has a growing need for software expertise,

but that it is not able to meet this need through intrinsic resources. This need is essential for the DoD to be a smart software customer and program manager, particularly for larger-scale innovative software-intensive projects. In particular, access to DoD-aligned expertise is important for the DoD to be able to take effective action in the three areas of practice that are identified above. Access to DoD-aligned expertise has been an area of ongoing challenge to the DoD, with recommendations made by various panels and committees since the 1980s.

The need to reinvigorate DoD software engineering research

In addition to recommending improvements to the three areas of practice, as outlined above, the committee identified seven areas of supporting research for consideration by science and technology program managers (managing 6.1, 6.2, and 6.3a funds and equivalent). These areas are identified on the basis of four criteria: (1) Advances would yield significant potential value for DoD software producibility. (2) A well-managed research program would result in feasible progress. (3) The goals are not addressed sufficiently by other federal agencies. (4) The pace of development in industry or research labs would be otherwise insufficient.

In each of the seven areas, the committee identified specific goals for research and technology development that, in its judgment, could feasibly meet the four criteria. The areas and, for each, the identified goals are summarized below. (Details are in the full report.)

1. **Architecture modeling and architectural analysis.** Goals include: (1) Early validation for architecture decisions; (2) Architecture-aware systems management, including: Rich supply chains, ecosystems, and infrastructure;

(3) Component-based development, including architectural designs for particular domains

2. Validation, verification, and analysis of design and code. Goals include: (1) Effective evaluation for critical quality attributes; (2) Components in large heterogeneous systems; (3) Preventive methods to achieve assurance, including process improvement, architectural building blocks, programming languages, coding practice, etc.

3. **Process support and economic models for assurance.** Goals include: (1) Enhanced process support for assured software development, (2) Models for evidence production in software supply chains, (3) Application of economic principles to process decision-making

4. **Requirements.** Goals include: (1) Expressive models, supporting tools for functional and quality attributes; (2) Improved support for traceability and early validation

5. **Language, modeling, coding, and tools.** Goals include: (1) Expressive programming languages for emerging challenges, (2) Exploit modern concurrency: shared-memory and scalable distributed, (3) Developer productivity for new development and evolution

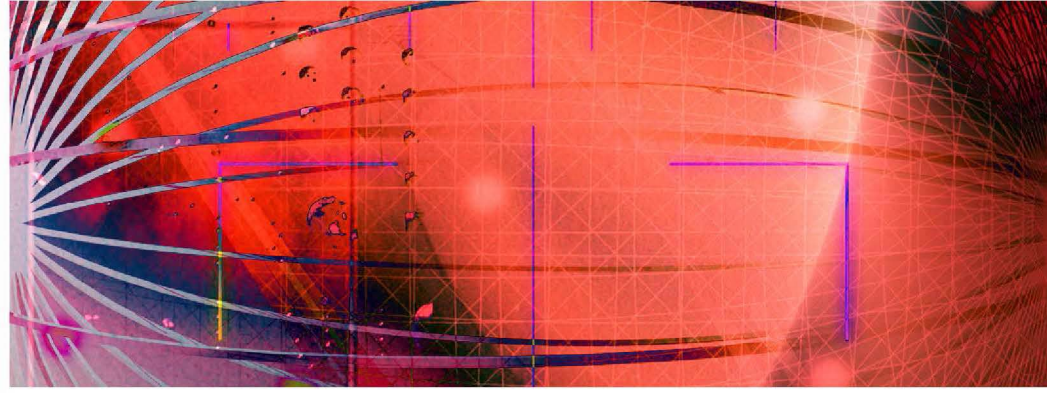
6. **Cyber-physical systems.** Goals include: (1) New conventional architectures for control systems, (2) Improved architectures for embedded applications

7. **Human-system interaction.** Goals include: (1) Engineering practices for systems in which humans play critical roles. *(This area is elaborated in a separate NRC report.)*

Under the auspices of the Office of Science and Technology Policy (OSTP) and the National Science and Technology Council (NSTC), there is a National Coordination Office for the Networking and Information Technology Research

and Development (NITRD) program. The NITRD program provides a framework for diverse federal agencies to coordinate R&D in areas related to networking and information technology. The framework includes two areas that primarily relate to software producibility, which are Software Design and Productivity (SDP) and High Confidence Software and Systems (HCSS). There is also a third area, Cyber Security and Information Assurance (CSIA) that encompasses some activities related to software producibility.

The committee undertook a longitudinal study of sponsored R&D budgets as identified in NITRD reports, with specific focus on SDP and HCSS. It found that while NITRD overall has grown over the past decade, there has been a significant reduction in both overall and DoD-sponsored R&D in SDP and HCSS. The committee recommends that DoD take immediate action to reinvigorate its investment in software producibility research, with focus in **the seven identified areas.** 



Cyber-Physical Systems (CPS)

Over its brief history, most of the computer science and engineering field has focused on systems (e.g., the Internet and Web) that enable humans through information, communication, and knowledge. Just as the first wave of desktop and high-performance computing technology revolutionized the way people interact with information and with each other, the second wave will revolutionize the way humans interact with their physical environment.

Our vision is one of fundamentally cyber-physical systems that exhibit deeply integrated computational and physical capability, interacting with humans through many new modalities. In this future, the ability to interact with, and expand capabilities of, the physical world through computational means will be the key technological multiplier. Individual precursors are seen in the control of inherently unstable systems such as *flying wings* and other extreme-performance aircraft, automobiles with hybrid gas-electric or hydrogen-electric car engines and enhanced vehicle stability systems, fully autonomous urban driving, medical devices for deep brain stimulation, and prostheses that allow brain activity to control physical objects. A rich field of innovative research is envisioned that can advance human progress through the *tensor product* of cyber (computing, communication, and control) technology and the dynamics of natural and

engineered physical systems—as well as their interactions with human participants.

What will such future systems be like? Every system action will be engineered to exploit both cyber and physical capability, deeply integrated throughout the system. Systems will interact with humans in entirely new ways, sharing authority. They may be highly tailored to the requirements and needs of individual users and uses, hence highly heterogeneous. These systems will be extensively, even ubiquitously, networked. The majority of the systems will be configured from cooperating components that interoperate through a complex mechanical, electrical, biological, and/or chemical system, coupled with a physical environment such as a human. Many (perhaps most) systems will be safety-, life-, or mission-critical and must be highly dependable, available, and secure. They will exhibit complex dynamics at many spatial and temporal

scales. They will need to be predictive, reactive to conditions and external events with predictable and accurate timing, and receptive to coordination and (private) negotiation. Control loops may need to be closed at various levels and scales. Topologies may adapt and reconfigure. Cyber-physical systems (CPS) will have to be fault tolerant and recoverable, satisfying potentially very high availability and timeliness requirements.

CPS is a vision then for developing a scientific and engineering foundation for routinely building cyber-enabled engineered systems in which cyber capability is deeply embedded at all scales, yet which remain safe, secure, and dependable—“systems you can bet your life on.” The CPS challenge spans essentially every engineering domain. It will require the integration of knowledge and engineering principles across many computational and engineering research disciplines (computing,

networking, control, human interaction, learning theory, as well as electrical, mechanical, chemical, biomedical, nanobioengineering, and other engineering disciplines) to develop a “new CPS science.”

Impact/need for the CPS initiative

A new foundation is required for future CPS. The existing science and engineering base does not support the routine, efficient, and robust design and development of these inherently complex systems. Such complex systems must possess trustworthy qualities that are lacking in much of today’s cyber infrastructures. Today we can produce (at great cost and effort) exceptionally complicated systems. We lack, however, the scientific and engineering foundations to securely, safely, and systematically understand, build, manage, and adapt CPS that remain reliable as they interact across internal subsystems, with each other, with human users, and with highly complex and uncertain physical environments.

The design complexity of *x-by-wire* for complex systems already is outstripping safe engineering design and implementation. Also, the opportunities for mischief in this generation of technology will make today’s Internet security problems pale by comparison. The consequence is inefficient, unsound, and potentially dangerous design outcomes, as well as tedious, costly, and failure-prone design cycles. Certification is estimated to consume 50 percent of the resources required to develop new, safety-critical systems in the aviation industry. Similar estimates are predicted for the medical and automotive domains. Over-design currently is the only path to safety and successful system certification, leading to a mindset of optimizing for a narrow task instead of encouraging adaptability and

evolvability. Yet, wide design margins both limit performance and may vanish in the face of changing usage patterns. This lack of design discipline induces extreme risk in technology-impovertised sectors such as the electric power industry.

The objective of an initiative would be to establish unified foundations and technologies, and exemplars for rigorous joint engineering of the cyber, physical, and human aspects of systems. This objective includes science and technology for the engineering of cyber and physical components that must be integrated to constitute such systems. Additionally, this objective includes the cyber-physical characterization of complex environments and human action, within which such systems must operate and to which they contribute. In contrast with today’s artisanal approach, our objective is to build foundations, tools, and highly capable infrastructure for rigorous design and engineering of 21st century systems that are truly *cyber-physical*.

Today, CPS grand challenges are being articulated in many sectors (for example, net-zero energy buildings, a smart grid, energy management systems for petroleum-free energy, zero-fatality and zero-crash highway and vehicle systems, zero-prototype manufacturing, and the wireless and highly automated *operating room of the future*). These heavily computation-, control-, and communication-centric systems call for a new, unified systems science and new engineering technologies imagined by the CPS initiative. In a keynote address on the challenges of design automation for emerging vehicle technologies, Scott Staley, Chief Engineer, Hybrid & Fuel Cell Technology Development for Ford Motor Company argued the need to abandon ad hoc experimental design approaches and find more rigorous methods, saying, “...incremental

modifications on the status quo will not work!” Don Winter, Vice President for Engineering and Information Technology, Boeing Phantom Works, in a hearing before the House Science Committee, called for “*a national strategy in which long-term CPS technology needs are addressed by combined government and corporate investment.*”

A focused initiative in CPS is needed that would seek to maximize human capability and well-being through computationally enabled engineered and physical systems. The goal would be to usher in a new era of CPS for which we have end-to-end science and engineering principles. The extent to which such advances are achieved will determine (and can transform) the course of US innovation; advancement of consumer health, safety, and security; and government agency mission effectiveness. 🇺🇸

Make a critical difference with what you know.

You already know that intelligence is vital to national security. But here's something you may not know.

The National Security Agency is the only Intelligence Community agency that generates intelligence from foreign signals and protects U.S. systems from prying eyes.

If you have the professional skills or technical expertise to support this important mission, then explore NSA. At NSA you can experience a variety of opportunities throughout your career as you work on real-world challenges with the latest technology. You'll also be able to maintain a good balance between work and family life, as well as enjoy a collaborative work environment with flexible hours.

You won't find this kind of experience anywhere else.

KNOWING MATTERS



NSA

www.NSA.gov/Careers

APPLY TODAY

WATCH THE VIDEO 

Excellent Career Opportunities in the Following Fields:

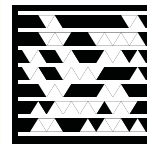
- Computer/Electrical Engineering
 - Computer Science
 - Information Assurance
 - Mathematics
 - Foreign Language
 - Intelligence Analysis
 - Cryptanalysis
 - Signals Analysis
 - Business Management
 - Finance & Accounting
 - Paid Internships, Scholarships, and Co-op
- >> **Plus** other opportunities

WHERE INTELLIGENCE GOES TO WORK®

 Find us on
Facebook

FOLLOW US ON 

Search: NSACareers



Get the free App for your camera phone at **gettag.mobi** and then launch the App and aim it at this tag.