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

Sally A. Browning, Magnus Carlsson, Levent Erkök, John Matthews, Brad Martin, Sean Weaver

domain-specific language (DSL) is a programming language targeted at producing solutions in a given problem domain by enabling subjectmatter 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.

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:

 ${k}{k \ge 2, 4 \ge k}$ => ([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.

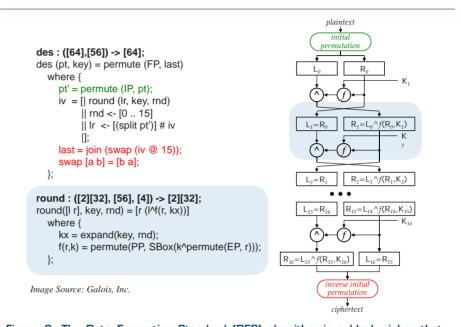


Figure 3: The Data Encryption Standard (DES) algorithm is a block cipher that uses a 56-bit symmetric key. The diagram above is taken from the Standard [3]. Cryptol uses parallel stream comprehensions to interleave data and lazy evaluation to encapsulate multiple computational stages in a single statement. 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 =>([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 =>, we see the actual type. The function has two inputs: a 128bit 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.

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.

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-largescale 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-byside 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) handcoded 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.

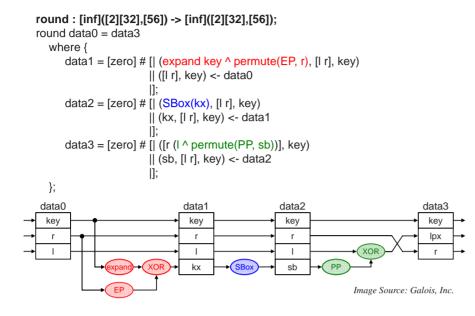


Figure 4: The code snippet above shows a new implementation of the DES round function, shown in Figure 3 on page 6. A flow diagram is included, with colors showing the correspondence between code and diagram element. This version uses sequence comprehensions that can be performed in parallel and introduces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

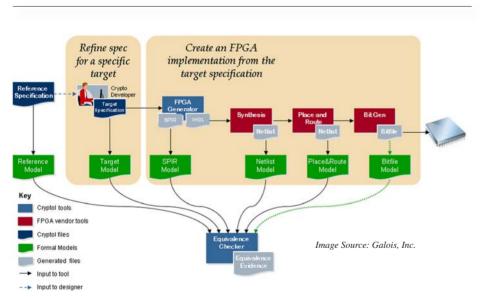


Figure 5: Verification can be performed at various points during the translation, which allows for high-assurance refinement during development. Note that the major compiler phases (the flow through the top line) remain outside the trusted-code base for verification. Trust in the down-arrows, representing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

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 equivalenceand 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 equivalencechecking 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 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 expensive computationally 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 invertergraph) 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. Cryptol's foreign-function Through 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 blockbased 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 toplevel 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 opensource 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.

Acknowledgements

Many people have worked on Cryptol and its formal verification toolset over the years, including Sigbjorn Finne, Andy Gill, Fergus Henderson, John Launchbury, Jeff Lewis, Thomas Nordin, Lee Pike, Mark Shields, Joel Stanley, Frank Seaton Taylor, Philip Weaver, and Adam Wick.

Men Long of Intel and Stefan Tillich of TU Graz kindly made their VHDL code available to us for verification and answered several questions about their implementations. General Dynamics C4 Systems, Rockwell Collins, and Michal Kouril have graciously allowed us to write about their experiences with the Cryptol tools.

About the authors

Galois is a research and development company with a strong drive to transition technology from research into practice in the commercial and government sphere. Since our founding in 1999, we have been funded for research and development by members of the Intelligence Community, the Departments of Defense, Homeland Security, and Energy, and the National Aeronautics and Space Administration.

Dr. Sally A. Browning (PhD, California Institute of Technology) leads the Cryptol family of projects at Galois. **Dr. Magnus Carlsson** (PhD, Chalmers University), **Dr. Levent Erkök** (PhD, Oregon Graduate Institute), and **Dr. John Matthews** (PhD, Oregon Graduate Institute) are key members of the team, with world-class expertise in functional programming, language design and formal methods.

The High Confidence Software and Systems (HCSS) Division supports development of scientific foundations and technologies for innovative systems design, systems and embedded application software, and assurance and verification to enable the routine production of reliable, robust, safe, secure, and certifiably dependable IT-centric physical and engineered systems. The HCSS Division resides within NSA's National Information Assurance Research Laboratory, a laboratory responsible for conducting and sponsoring research in the technologies and techniques needed to secure America's future information systems.

Brad Martin manages NSA's HCSS Division. In 1999 Brad Martin and John Launchbury, founder and chief scientist of Galois, conceived of Cryptol. **Sean Weaver** is a researcher within the HCSS Division and is the current technical lead for NSA's Cryptol Program.

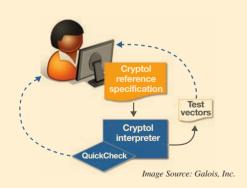
Q: What can YOU do with Cryptol?

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-toend test vectors provided in algorithm source specifications. Finally, Cryptol models are evolving to directly support the certification effort..."

Alan Newman General Dynamics C4 Systems



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.

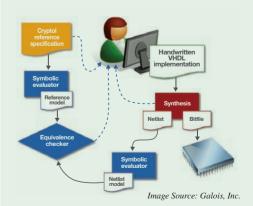
Q: What can YOU do with Cryptol? A: Gain assurance about your design.

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 $\{1 \ 2 \ ... N\}$ 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 de 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 ...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.



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], simili [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 ANDgates 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.



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, [Oth 1st 2nd 3rd], and when using !, the order is reversed, [3rd 2nd 1st Oth]. Each number or element contains a certain numbers of bits: 1 (Ob1) contains one bit, 2 (Ob1O) is two bits, 3 (Ob11) is also two bits and 4 (Ob1OO) 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."

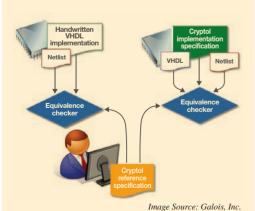
Excerpt from a report by Avery Tose, a senior attending Lighthouse Christian Academy in Stevensville, Maryland, who participated in "Exploring Science through Patterns in Nature," an enrichment activity led by Brad Martin, NSA, May, 2009.

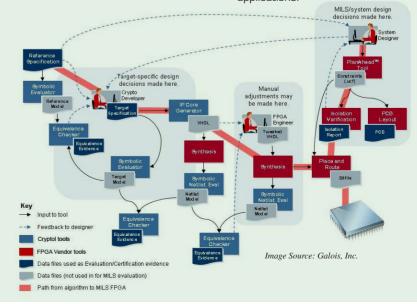
Q: What can YOU do with Cryptol? A: Make a MILS FPGA.

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 (highlevel designs, automatic generation of implementation from design, automaticallygenerated equivalence evidence), and has the potential to significantly reduce the time of costs of developing Type-1 cryptographic applications.





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/Pubs FIPS.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.skeinhash.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 highassurance 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