Hardware/Software Co-verification of Cryptographic Algorithms using Cryptol

Levent Erkök Magnus Carlsson Adam Wick

Galois, Inc.
421 SW 6th Ave. Suite 300
Portland, OR 97204

***Abstract*—Cryptol is a programming language designed for specifying cryptographic algorithms. Despite its high-level modeling nature, Cryptol programs are fully executable. Further, a large subset of Cryptol can be automatically synthesized to hardware. To meet the inherent high-assurance requirements of cryptographic systems, Cryptol comes with a suite of formal-methods based tools that enable users to perform various program verification tasks. In this paper, we provide an overview of Cryptol and its verification toolset, especially focusing on the co-verification of third-party VHDL implementations against high-level Cryptol specifications. As a case study, we demonstrate the technique on two hand-written VHDL implementations of the Skein hash algorithm.**

***Index Terms*—Specification and Verification, Equivalence checking, HW/SW Co-verification, Cryptography**

I. Introduction

Cryptol is a domain specific language tailored for the domain of cryptographic algorithms.[1] Cryptol specifications are fully executable on commodity hardware using an interpreter, and a large subset of Cryptol programs can be automatically synthesized to various hardware platforms via translation through VHDL. Explicit support for verification is an indispensable part of the Cryptol toolset, due to the inherent high-assurance requirements of the application domain. To this end, Cryptol comes with a suite of formal-methods based tools, allowing users to perform various program verification tasks. In this paper, we provide a short overview of the Cryptol language, especially focusing on verification.

Cryptol is a pure functional language, built on top of a Hindley-Milner style polymorphic type system, extended with size-polymorphism and arithmetic type predicates [1]. The size-polymorphic type system has been designed to capture constraints that naturally arise in cryptographic specifications. To illustrate, consider the following text from the AES definition [2, Section 3.1]:

> 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.

This description is captured precisely in Cryptol by the following type:

[1]Cryptol toolset licenses are freely available at www.cryptol.net.

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

Anything to the left of => are quantified type-variables and predicates on them. In this case, the type is size-polymorphic, relying on the size-variable `k`. The predicates constrain what values the quantified size-variables can accept: Here, `k` is restricted to be between 2 and 4. To the right of =>, we see the actual type. The input to the function is a pair of two words, the first of which is the 128-bit plain-text. The second argument is a `64*k`-bit wide word (i.e., 128, 192, or 256 bits, depending on `k`) representing the key. The output of the function, the ciphertext, is another 128-bit word. Note how this type precisely corresponds to the English description.

II. Verification of Cryptol Programs

Cryptol's program 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 one to formally prove that f and g are semantically equivalent, ensuring that the (often very complicated and extensive) optimizations performed during synthesis are semantics preserving. Note that the final implementation (i.e., g) need not necessarily be in Cryptol: an important use case of the verification framework is in verifying third-party algorithm implementations (typically in VHDL) are functionally equivalent to their high-level Cryptol versions. In this case, Cryptol acts as a HW/SW co-verification tool.

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 reliability and availability of Cryptol-based products, since they eliminate the need for sophisticated run-time exception handling mechanisms whenever applicable.

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 [3]. As the reader might suspect, the push-button system suffers

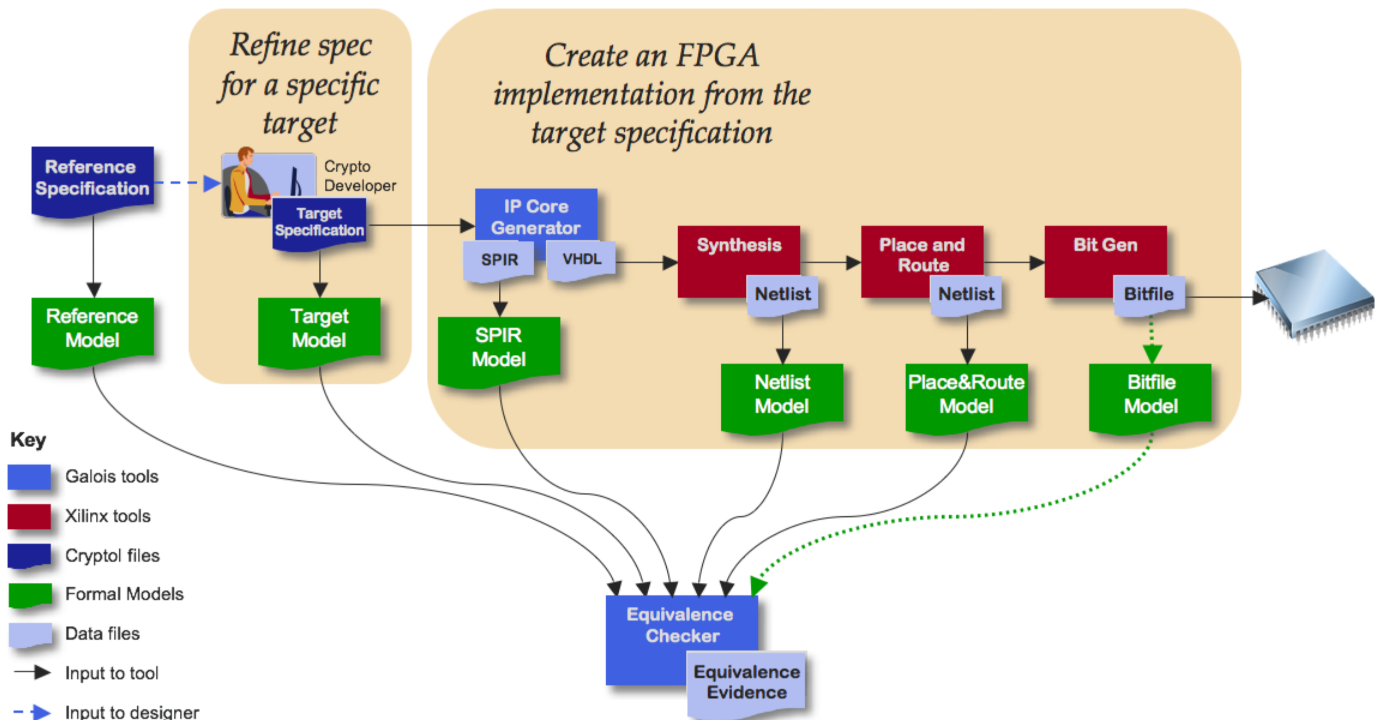

Fig. 1. Typical design and verification flow in Cryptol. Verification can be performed at various points during the translation, which allows for high-assurance refinement during development. Note that the major compiler phases (the flow through the top-line) remain out of the trusted-code base for verification: One only needs to trust the down-arrows representing translators from various intermediate forms to AIG-based formal models, along with the off-the-shelf equivalence checkers themselves.

from state-explosion, and thus might fail to provide an answer in a feasible amount of time for larger designs. Cryptol uses off-the-shelf SAT/SMT solvers (such as ABC or Yices) as the underlying equivalence checking engine, translating Cryptol specifications to appropriate inputs for these tools automatically [4], [5]. However, the use of these external tools remains transparent to the users, who only interact with Cryptol as the main verification tool.

It is essential to emphasize that equivalence checking applies not only to hand-written programs but also to generated code as well. 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 a prohibitively expensive task, if not impossible. Instead, Cryptol provides a *verifying* compiler, generating code together with a formal proof that the output is functionally equivalent to the input program [6], [7]. As opposed to using a deep-embedding in a theorem prover like ACL2, however, we utilize modern off-the-shelf SAT and SMT solvers to perform automated equivalence checking.

A. Design and Verification Flow

Figure 1 provides a high-level overview of a typical Cryptol development and co-verification flow. Starting with a reference specification in Cryptol, the designer successively refines his program and "runs" his design 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 tool-chain 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 since it remains fully executable, 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 1 shows the flow for the Xilinx tool-chain; taking the VHDL through synthesis, place and route, and bit-file generation steps. In practice, these steps might need to be repeated depending on feedback from the synthesis tools. The overall approach aims at reducing the number of such repetitions greatly, by providing early feedback to the user at the SPIR level. The final outcome is a custom binary file that can be downloaded onto an Xilinx FPGA board, completing the design process.

Our co-verification flow is interleaved with the design process. As depicted in Figure 1, Cryptol provides custom translators at various points in the translation process to generate formal models in terms of AIG (and-inverter graph) representations [8]. In particular, the user can generate AIG representations from the reference (unoptimized) Cryptol spec-

ification, 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 transformations applied 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 1. The format of this file remains proprietary, but we hope to provide this final link through future collaboration with Xilinx.

B. Verification for the Cryptography domain

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 crypto-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 [9] are especially effective on crypto-algorithm verification, since simulation-based node-equivalence guesses are likely to be quite accurate for algorithms that heavily rely on shuffling input bits. Obviously, these properties do not make formal verification trivial for this class of crypto-algorithms; but rather they make the use of such techniques quite feasible in practice [10].

III. Case study: Skein

Skein [11] is a suite of cryptographic hash algorithms targeted at the NIST competition for choosing the next-generation hash function SHA-3 [12]. 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. A detailed write-up on the Cryptol implementation of Skein is publicly available [13].

The process of proving that a given VHDL implementation is functionally equivalent to a Cryptol reference specification begins with understanding the high-level interfaces of both. Once the high-level input/output correspondences are determined, the VHDL implementation is imported into the Cryptol program using Cryptol's `extern` declaration capability. Then, the required interface-matching code is written in Cryptol, mainly taking care of the proper use of control-signals. This allows the external implementation to be 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 AIGs from both, and checks for equivalence.

We verified our implementation of Skein against two separate VHDL implementations using this methodology. In each case, we have used Alan Mishchenko's ABC tool as the underlying equivalence checker [4]. The verification was performed for one 256-bit input block, generating a 256-bit hash value.

The first verification was performed against Men Long's implementation [14]. Since Long did not implement the full Skein algorithm, but instead implemented only the underlying Threefish encryption and the XOR of input data, we tweaked our reference Cryptol implementation to match this partial result. The AIG generated for the reference implementation in Cryptol had 118156 and-gates, while the VHDL version gave rise to 653963 and-gates; about 5.5 times larger. The equivalence checking process took about an hour to complete on commodity hardware using ABC. During the verification effort, we encountered a problem with a piece of code that rotates a 64-bit signal a variable number of steps. It was given three different meanings by GHDL [15], simili [16] and the Xilinx synthesis tools. We were able to remove the ambiguity of the code by replacing it with the standard library function `rotate_left`. In essence, the Cryptol co-verification path found an ambiguity bug that remained undetected before.

We performed our second verification against Stefan Tillich's Skein implementation [17], which is a full implementation of the Skein algorithm. The AIG sizes in this case were 301085 and-gates for the reference Cryptol versus 900239 and-gates for the VHDL implementation; about 3 times larger. In this case, equivalence checking was completed in about 18 hours using ABC.

Commonly, the VHDL implementations include a global reset signal that brings back the circuit's internal registers to their initial state. In some cases, the reset signal is acted upon immediately regardless of the clock signal, that is, in an asynchronous manner. Our tools are currently not able to generate formal models from VHDL code with asynchronous resets—instead we have an additional assumption that the reset signal is only asserted initially and in a synchronous manner across a rising clock edge. Under this assumption, we manually perform an equivalence-preserving rewrite of the VHDL code so that it uses the reset signal synchronously. In the case of Tillich's code, this was a local rewrite consisting of a couple of lines of VHDL code change. While this is a problem for any asynchronous signal, we plan to enhance our tools so that they can at least handle asynchronous resets without the need for manual VHDL code modifications.

IV. Challenges

Increasing the coverage of formal methods: Cryptol's formal verification framework works on a (relatively large) subset of Cryptol [3]. The main limitation is in the verification of algorithms for all-time, i.e., those programs that receive

and produce infinite streams of data. While infinite streams pose no challenge for synthesis, Cryptol can only equivalence check such algorithms up to a fixed number of clock-cycles. Although 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: Unsurprisingly, not all properties of interest can be cast as functional equivalence problems. This is especially true in the domain of cryptography. For instance, if we are handed an alleged VHDL implementation of AES, we would like to ensure that it not only implements AES correctly but also that it does not contain any extra circuitry to leak the key.

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 Haskell code. Reducing the footprint of this trusted code base, and/or increasing assurance in these components of the system is an on-going challenge.

V. Related Work

Software/Hardware co-verification problem is an open research area, especially focusing on equivalence checking C-style programs against RTL implementations [18], [19]. Similar to earlier pioneering work in this area, our approach does not freeze the—usually very large and complicated—code generator portion of the system, since the code generator is not in the trusted path. The reduction of the trusted code base is a significant gain for assurance purposes. Unlike earlier work, however, we do not assume that the implementations in these languages are "similar," or done in certain styles to enable effective verification. In fact, Cryptol specifications remain purely functional and hence combinatorial, while VHDL implementations are typically highly state-based and thus sequential. Note that, our approach remains bit-precise, i.e., no simplifying assumptions are made on the semantics of the underlying languages.

VI. Conclusions

In this paper, we have provided a brief description of the Cryptol language, an overview of its verification framework, and a case study of verifying the hash algorithm Skein. The novel part of our approach is the bridging of hardware and software artifacts, allowing designers to treat them uniformly during verification. The system has already proved itself useful in practice, especially in establishing the equivalence of reference and extensively optimized implementations of crypto-algorithms such as AES. Since such transformations are done both manually by programmers and automatically by the compiler and external synthesis tools, it is essential that automated formal-verification capabilities are seamlessly integrated into the process, ensuring that the final hardware implementations are semantically equivalent to their high-level reference specifications.

Acknowledgments

Many people have worked on Cryptol and its formal verification toolset over the years, including Jeff Lewis, Thomas Nordin, John Matthews, Sigbjorn Finne, Phil Weaver, and Sally Browning. Men Long of Intel and Stefan Tillich of TU Graz kindly made their VHDL code available to us for verification and answered several questions on their implementations.

References

[1] J. R. Lewis and B. Martin, "Cryptol: high assurance, retargetable crypto development and validation," in *Military Communications Conference 2003*, vol. 2. IEEE, Oct. 2003, pp. 820–825.

[2] NIST, "Announcing the AES," November 2001, FIPS Publication 197. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[3] L. Erkök and J. Matthews, "Pragmatic equivalence and safety checking in Cryptol," in *Programming Languages meets Program Verification, PLPV'09, Savannah, Georgia, USA*. ACM Press, Jan. 2009, pp. 73–81.

[4] A. Mishchenko, "ABC: System for sequential synthesis and verification," 2007, release 70930, Available at: http://www.eecs.berkeley.edu/~alanmi/abc.

[5] "Yices web site," http://yices.csl.sri.com/.

[6] L. Pike, M. Shields, and J. Matthews, "A verifying core for a cryptographic language compiler," in *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*. New York, NY, USA: ACM, 2006, pp. 1–10.

[7] W. A. Hunt and E. Reeber, "Formalization of the DE2 language," in *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3725. Springer, 2005, pp. 20–34.

[8] A. Biere, "The AIGER And-Inverter Graph (AIG) format," 2007. [Online]. Available: http://fmv.jku.at/aiger/FORMAT.aiger

[9] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[10] E. W. Smith and D. L. Dill, "Automatic formal verification of block cipher implementations," in *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–7.

[11] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The Skein Hash Function Family," 2009, http://www.skein-hash.info.

[12] "NIST Cryptographic Hash Algorithm Competition," 2008, http://csrc.nist.gov/groups/ST/hash/sha-3.

[13] S. Finne, "A Cryptol implementation of Skein," 2009, http://www.galois.com/blog/2009/01/23/a-cryptol-implementation-of-skein.

[14] M. Long, "Implementing Skein hash function on Xilinx Virtex-5 FPGA platform," 2009, http://www.skein-hash.info/sites/default/files/skein_fpga.pdf.

[15] "GHDL simulator version 0.26," http://ghdl.free.fr/.

[16] "Simili VHDL simulator version 3.1," http://www.symphonyeda.com/products.htm.

[17] S. Tillich, 2009, The Institute for Applied Information Processing and Communications (IAIK). http://www.iaik.tugraz.at/content/research.

[18] A. Pnueli, O. Shtrichman, and M. Siegel, "The code validation tool (cvt) - automatic verification of code generated from synchronous languages," *Software Tools for Technology Transfer*, vol. 2, 1998.

[19] L. Séméria, R. Mehra, B. M. Pangrle, A. Ekanayake, A. Seawright, and D. Ng, "Rtl c-based methodology for designing and verifying a multi-threaded processor," in *Proceedings of the 39th Design Automation Conference, DAC 2002, New Orleans, LA, USA, June 10-14, 2002*, 2002, pp. 123–128.