Towards a Tool for Verified Hardware Construction

Jared Roesch,
Department of Computer Science,
University of California Santa Barbara.

May 31, 2014

Abstract

Designing secure hardware is an active area of research that has seen much growth in recent years. Hardware is now more complicated then ever but even if we state and prove high level security properties about our designs we have no way to demonstrate a correspondence between the thing we formally discuss and the thing that we eventually design. Unfortunately the tools used to construct hardware have not fundamentally changed in decades. We present a high level overview of the state of the art with regards to these issues, and sketch a plan of how to adapt these techniques in order to build a toolchain for the formal verification of hardware synthesis.

1 Introduction

TODO: need to finish putting all my citations in

Hardware security is becoming and incresingly important topic both as a research problem and a pragmatic issue effecting day to day life. Hardware is multiplying around us and proliferating through all parts of life. We are seeing more and more devices spring into existance to manage everything from avionics, pace makers, to your thermostat. One of the problems is that we are giving these devices more and more control over critical systems that govern the safety and lives of humans. At the same time we also have issues with mobile security, cloud computing, and more.

Traditionally many of these problems have only been thought about at the software level and much of the related work has been about verifing correctness and security properties about the systems running on top. Ideally we want to be able an run entire stack of critical software; a verified application, running on a verified operating system. This picture unforuntately doesn't factor in vulnerabilities at the hardware level. Often times the hardware's security has been an afterthought, but there has been multiple recent examples of attacks that leveraged hardware level exploits in order to compromise systems(cite here).

The missing part of this equation then becomes the verification of hardware. Hardware verification is not necessarily a trivial problem. Although hardware construction has similarities with design and verification of software, there are key issues that must be solved. There has been (insert citations) in verifiying simple hardware description, and properties ofhardware. (More limitations here) The goal then is to build a tool focused on hardware construction that exposes more powerful tools to the engineer who designs hardware.

(incorporate these ideas) Unforunately previous approaches present a DSL in a Theorem Proving environment. There is a endeavor to free the hardware designer to be able to both build hardware and prove properties.

2 Background

There has been distinct work in each area of secure computer arhcitectures, and hardware description languages, and theorem proving. We survey the important highlights from each that are necessary to understanding our approach. more text here probably

2.1 Secure Architecture

- Ruby's work, Tim's on execution leases, Appel
- All high level properties that are loosely reasoned about
- want flexibility in design, parametrizable modules (show example from GitHub here)
- Missing rigorous formal analysis
- When they have analysis it suffers from being Pen and Paper, and being abstracted from the actual work.

In [3] they describe secret-protected architecture that enables the protection of critical secrets for users in an online environment. They argue that their architecture requires only a few extra features to be built into a general purpose microprocessor to protect secrets and related computations. They do this with a small amount of extra hardware. They are able to decouple user secrets from the devices, avoid symmetric master keys, and avoid doing factory-installed device secrets. Their approach can be divided into a few core features the Key Chain, Trusted Software Module, Concealed Execution, Hardware Extensions, Secure I/O, and OS support.

work some critique in here: The evaluation could use some work. There is a written description of both their security and performance evalution, but neither has explicit numbers. It seems that although Ruby talks about performance vs. secu- rity tradeoffs in her book, she is not completely clear about the ones made here. As well they are lacking any sort of formal proof of their security properties. This may not be important to the architecture community, but it seems questionable at best to make claims about the security of a system at a high level without actually formally verifiying the properties hold in practice. The performance evaluation also seems to be very high level, and ignores important details, for example they dismiss the performance impact on the overall system without presenting precise empirical evaluation of it. They do evaluate the instruction load of their hardware crypto but do not compare it aganist anything else in the space.

In [5] they described ways to add a type system to a small imperative language in order to mitgate the leakage of timing information. Ideally these hardware concerns shouldn't leak beyong the hardware level, and they grappel with this in their analysis of the technique.

put this somewhere: It would be desireable to adapt their technique from [1] and use our technique to help prove that correctness of the approach.

2.2 Hardware Description Languages

Hardware description languages are software tools for synthesizing hardware from a high level programmatic description. A hardware description language is ideally a precise, formal description of an electronic circuit that makes it easy to do automated analysis, simulation, and testing for a circuit. In reality they have various weaknesses and failings

2.3 Type Theory

Type theory is a foundational framework for mathematics. It was an approach that evolved out of Russel's [?] as an approach to a consistent foundational system for mathematics. The early 20th century was spent developing type theory as a proof system. The key insight was the Curry-Howard Isomorphism, which roughly states that types can be associated with propositions and proofs with programs. This insight was adapted and it became apparent that typed variants of the lambda calculus can be used as proof systems, the most famous of which is Martin-Lof type theory. You can then view types in this typed lambda calculus as various features of constructive logic.

TODO: clean this section up with typeset examples and such

If we begin with just an implicational fragment of logic we can use the simply typed lambda calculus which only has a single type constructor, the function type. This corresponds to implication. We can now view the typing judgment $a : A$ as statement that a is a proof of A, and we can interepret the typing judgment $f : A- > B$ as a computational implication. At the type level it allows us to produce a B given an A, and computationally it gives us a method for transforming a program of A into a program of B. If we remember that program's corresond to proofs we now have a computational method for proving.

We can of course extend our type system with richer features. We will quickly survey the important additions to the Simply Typed Lambda Calculus that are needed to turn it into a dependently typed lambda calculus. The most important feature is the idea of Pi Types. Simple type systems can be enriched with many features such as products, sums, so on. There have been innovations in type system construction, but depdently typed calculi in the style of the CoC or MLTT are the most powerful.

Traditonally we maintain a phase distinction where types and values leave in seperate universes. Depdent types are about erasing this distinction. Types can be made more expressive by a series of additions. First if we allow quantifers we can now abstract over types with type lambdas. We can then introduce kinds which allow us to specify type arities. We can then introduce kind polymorphism, and many other features. Each of these systems become more and more complex because they have to maintain constant distinction between each universe of values, kinds, sorts. Depdent types simply introduce the idea of a Pi Type of dependent function space. This type system generalization is the idea that we can make everything much simpler by creating a stratfied hierarchy of types, and combining this Pi Types. We then can collapse our language into a simple set of pseudo terms, and typing rules examples can be found.

The idea of Pi Types can be simply viewed as a generalization of the traditional function type. The generalization allows the codomain to depdend on the domain of the function. Given $B : P- > A; \Pi(x : A).B(x)$.

2.4 Automated Theorem Proving

High level talking points:

- dependent types
- equal to a fragment of inutionistic logic
- provides tools for automated proof search
- specification and implementation are one and the same
- can do program extraction
- executable

Automated Theorem Proving has been an area of active research for many decades. It is apparent that the verification of general properties is desireable to both high assurance software engineers, researches, and mathmaticians. There have been various approaches to Automated Theorem Proving in the past ACL2, NuPRL, LF, LCF, Coq, Agda. The ones based on dependent type theories seem the most promising for various reasons (computational, program extraction, proof carrying code, ect). The cutting edge of type theory has been an evolutions of the work done my Per Martin-Lof and others.

Automated Theorem Proving has been applied in many areas from the NTSB, NASA, JPL, and a vareity of other places.

The problem is that very often the specification is seperate from the implementation and suffers the common problems of pen and paper proof. In many cases Embedded Domain Specific Languages (EDSLs) have been used as a way to gain the power of a proof assitant and also write proof carrying code. We want to leverage the ideas here, but ideally built a seperate front-end that exposes the full power of proving, but also allows one to provide both an usuable interface that also generates useful proof properties for free (similar to the work done by UCSD PLIDI '14 paper, needs citation).
NOTE: Work in (For now I'm using a proof environment called Idris for prototyping verified Hardware construction.)

2.5 Verified Hardware Construction

- Caisson [4]
- Fe-Si [2]
- Cryptol
- Bluespec

3 Extensions

- consistent design
- demonstrate examples like verification of overflow
- demonstrate bounded input sizes

- consistent design
- syntactic/semantic seperation unlike verilog behavioral/structural
- allow for some properties to be synthesized and proved automaticallly
- verify synthesize and semantics of HDL so that any well formed HDL program has certain properties (ala Coq EDSLs)
- find citation for UCSD PLDI '14 paper

Key weaknesses of previous approaches to verification:

- toy examples
- no automatic property generation/proof satisfaction
-

still prototyping the things I want to discuss here The goal is to take everything that has been described and synthesize the ways in which we can build a tool.

4 Conclusion

Wrap up, discuss weakness, challenges, ect

References

[1] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 297–307, New York, NY, USA, 2010. ACM.

[2] Thomas Braibant and Adam Chlipala. Formal verification of hardware synthesis. *CoRR*, abs/1301.4779, 2013.

[3] Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 2–13, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 109–120, New York, NY, USA, 2011. ACM.

[5] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110, New York, NY, USA, 2012. ACM.