Towards a Tool for Verified Hardware Construction

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

June 10, 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 model we formally discuss and the implementation that we eventually design. In the face of changing demands 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 tool chain for the formal verification of hardware synthesis.

1 Introduction

Hardware security is becoming and increasingly important topic both as a research problem and one that has an effect on 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 existence to manage everything from avionics, pace makers, to your thermostat. An issue then becomes, that as we give these devices more and more control, we are more and more interested in moderating their control. This is because control of these devices isn't just the device or its contents as it has historically been, but access to sensitive personal control, for example an exploited iPhone may have greater implications when it can open your doors, or command household electronics. Households could be made vulnerable to "malfunction" whether malicious or not. As well there are issues to be had with mobile security, as smart phones play a more and more important role in in daily life. It goes hand in hand with cloud computing, which is playing out as a nice dual to the rise of smart phones, more and more computation has been moved centrally to the "cloud" and it introduces central points of failure if they are compromised. No matter the platform it is necessary to begin ensuring the integrity of our software from the top down. Even if we introduce notions of verified software it is meaningless in the face of exploitable hardware.

Traditionally the specification and verification of security properties has only been done at an architectural level, by either design or by a baked in policy, and enforcement . A lot of work has been done about properties been about verifying correctness and security properties about the systems running on top. The ideal presented in older works was to be able to run an entire stack of critical software; a verified application, running on a verified operating system. This picture unfortunately doesn't factor in vulnerabilities at the hardware level. Often times the hardware's security has been an afterthought, but there has been many recent examples of attacks that leveraged hardware level exploits in order to compromise systems.

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 success in verifying simple hardware description, and properties of the designs. The goal is to take the kernel of an idea and flesh it out into a tool focused on hardware construction that exposes very powerful tools to the engineer who designs hardware.

Unfortunately the state-of-the-art [3] only presents a limited domain specific language in a theorem proving environment, providing very little support to to a hardware designer who wants to be able to write designs and then prove properties. We believe that it is possible to do better providing a pragmatic and useful tool that allows us to leverage the power of some properties for free, as well as the ability to add more via the proof system, enabling hardware design.

2 Background

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

2.1 Secure Architecture

In [7] 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. [**?**]. It seems that although Lee talks about performance vs. Security trade offs in her book, though their work is not clear about which trade offs they made. They also lack 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 verifying the properties hold in practice.

In [1] [14] they discuss mechanisms for architectural and language support for mitigating the leakage of information through side channels. They take the approach of inserting time stops in order to prevent implicit leaks from control flow structures. In [14] they described ways to add a type system to a small imperative language in order to mitigate the leakage of timing information. Ideally these concerns shouldn't leak beyond the hardware level, and they grapple with this in their analysis of the technique.

In [13] they also detail an information flow architecture base upon their GLIFT technique. The critical idea of the work is that we can provide secure leases that allow for a caller to bound the privileges both in terms of information and time for the lessee.

Having identified some of the weaknesses in the implementations of secure architectures we would ideally like to compliment them by providing a tool that enables the following criteria:

- state high and low level security properties
- reason about abstract specification
- correspondence between specification and hardware
- missing rigorous formal analysis

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 of a circuit. In reality they have various weaknesses and failings that mostly result from the weaknesses introduced by current tooling. We would ideally like to make it easier to state precise specifications of hardware, and verify that the implementations matches said specification.

2.3 Type Theory

Type theory is a foundational framework for mathematics. It was an approach that evolved out of Russel's work on setting mathematics on 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 widely used of which is Martin-Lof type theory. You can then view types in this typed lambda calculus as various features of constructive logic.

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 interpret 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 correspond 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 important additions to the Simply Typed Lambda Calculus that are needed to turn it into a dependently typed lambda calculus. The key idea is the erasure of phase distinction via the inclusion of a stratified type hierarchy, and Pi types.

Traditionally we maintain a phase distinction where types and values live in separate universes. Dependent types are about easing this distinction. Type systems can usually be made more expressive by a series of additions. The simplest of which is the introduction of quantifiers which allow us to abstract over types with type lambdas. We can then introduce kinds (the types of types), and kind polymorphism which allow us to specify type shapes or (type arities). We can continue to enrich our kind and type system, eventually reflecting types as kinds, and values as types, but these systems are actually more complicated then a dependently typed calculi. Each of these refines become more and more complex because it requires the language to maintain a discipline between values, kinds, sorts. Dependent 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 stratified hierarchy of types, and adding Pi and Inductive types. We then can collapse our language into a simple set of pseudo terms, and typing rules which are easy to verify for correctness.

We can then ignore all the complexity introduced by types systems such as System F and generalizations of it. Instead we can focus on a simple language. Usually a dependently typed core only requires a few forms abstraction, variables, function application, and Pi Types. This is simply the lambda calculus that everyone knows enriched with Pi types. It is also useful to extend the calculus with inductive types, or recursive data types that may be parametrized by the values they are constructed with.

Pi Types can be simply viewed as a generalization of the traditional function type. The generalization allows the type of codomain to depend on value of the domain of the function. Given $B : P- > A; \Pi(x : A).B(x)$. This simple extension actually gives rise to a lot of power and will allow us to encode much of mathematics in type theory.

This is of course a very brief overview and more information can be found in [9] [**?**] and [11].

2.4 Theorem Proving

Automated Theorem Proving has been an area of active research for many decades. It is apparent that the verification of general properties is desirable to both high assurance software engineers, researches, and mathematicians. 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). Automated Theorem Proving has been applied in many areas from the NTSB, NASA, JPL, and a variety of other places.

The current best way to leverage the proving power of Type Theory are Automated or Interactive Theorem Proving environments. These allow for the automatic satisfaction of some properties, and interactive proving of others. The state-of-the-art theorem proving environments in use for software verification are Coq, Agda, and Idris all of which provide the benefits of:

- dependent types (equivalent to a fragment of isolationistic logic)
- provides tools for automated proof search
- specification and implementation are one and the same
- executable ”proofs” (i.e program extraction)

These are all based on versions of intensional type theory. Coq is base on on Coquand and Huet's Theory of Constructions, and Agda is an evolution Martin-Lof Type Theory [**?**].

Automated Theorem Proving has been an area of active research for many decades. It is apparent that the verification of general properties is desirable to both high assurance software engineers, researches, and mathematicians. 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). Th current state-of-art is either based on Coquand and Huet's Theory of Constructions or the work done on Martin-Lof Type Theory [?]. Automated Theorem Proving has been applied in many areas from the NTSB, NASA, JPL, and a variety of other places.

It appears that ideas from ATP could be very useful in the domain of hardware specification. Most specification is performed separately 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 assistant and also write proof carrying code[12][3]. We want to leverage the ideas here, but ideally built a separate front-end that exposes the full power of proving, but also allows one to provide both an usable interface that also generates useful proof properties for free [12]. [6] [10]

There is a prototype implementation of ideas discussed in the final section available. It is currently written in Idris (which is one of the more practical dependently typed programming languages).

2.5 Verified Hardware Construction

There has been work on creating tools for aiding in the design of secure hardware such as Caisson which presents a HDL language that has a preselected static information flow policy. This provides a information flow policy by baking in a static analysis to the compiler. We would like to be able to easily experiment with different different security properties and be able generally state arbitrary security properties of the hardware. We can easily discharge these proof assumptions by a mixture of proof automation and interactive proving. SAFE takes a similar approach to the compiler assumes and enforces a standardized security policy. This is also the case with Caisson a gate level approach that is baked in to the HDL compiler [8].

There also has been work on specialized systems like Cryptol that make simplifying assumptions allowing certain guarantees and properties to hold [4]. Cryptol is able to use tools like SMT solvers and simple sized type system in order to prove properties about cryptography implementations. They are able to make these simplifications be identifying ways in which traditional cryptography hardware is constructed and designing their language accordingly.

There has been some related work in building higher level design languages such as Lava, and Bluespec that allow the hardware designer to abstract over properties such as explicit timing information. The most closely related work has been Fe-Si [3] which can viewed as a deterministic version of BlueSpec. The key insights to take away from the Fe-Si work is the description of how to use the meta languages type system for great effect, as well as inight into the verificatin of HDL compiler passes. Adam Chilpala has done a lot work on describing novel ways in which to verify different aspects of programming languages [5][6]

There also has been some work from the dependently typed programming language community sketching ways in which dependent types can be made useful in hardware design[2]. The authors only scratch the surface of the issues that occur in real hardware design and take a tutorial like approach to describing how to leverage dependent types to assist in design correctness, demonstrating very simple combinational designs like adders.

3 Extensions

Our general goals are to leverage ideas from program analysis, type systems and apply them to a hardware construction language. We motivate our desire to replace existing HDLs for the reasons we listed in the previous section. Primarily it is about pursuing a more consistent design, that allows us to have both concise hardware descriptions, and write proofs about said specifications. We also want to equip the surface language with the ability to demand certain properties and proofs for free. For example we want to provide guarantees of properties such as overflow, prop wiring by construction, and so on, but leave the ability to establish higher level proof properties like the information flow expressed by default in Caisson's compiler [8].

- allow for some properties to be synthesized and proved automatically
- verify synthesize and semantics of HDL so that any well formed HDL program has certain properties similar to [12]

Most previous approaches have not been practical enough to use for building real designs. Fe-Si is as close to complete but is still unfortunately just a toy. The DSL approach has important limitations and requires programming in an awkward monadic style inside of the theorem proof system Coq. Ideally we would like to take the simple core described in the Fe-Si work, but extend it in a few key ways. First we would like to add a surface language that will give us the ability to add more complicated features than the limited Fe-Si. We will also like to make modules first class entities that express information about the functionality of the circuit as well so that we can use such information when proving properties about designs.

We have experimented with a few different designs, and am still considering which is the best way forward. So far we have not made a decision on which is best, but have noticed some important trade offs in the process of implementing prototypes. Many of the naive designs don't work because they lose precious typing information that is essential for proving properties about the design. It has become apparent that many of the decisions made by Adam Chilpalia's work are important. We will use the same approach of using a denontational approach, and relying on properties of the meta-language such as strong normalization, and its type system to assure that our designs are correct by construction. We could of course elaborate our language to a core calculus and implement our own checker but it is important to not duplicate effort early in the research process, and should piggy back on existing work. For now we are translating directly to Idris source language, but it exposes its core calculus and checker as a library we can eventually use if we want to piggyback on their implementation effort. We also want to model his approach to slowly adding and proving the correctness of optimizations beyond the simple ones performed in [3].

One of the long term goals is a verified framework for abstract interpretation a powerful analysis technique, but for now we will continue on in the style of Chilpalia.One of the big departures from FeSi is that we want to provide a back door into the full theorem proving environment which will allow for designers or verifiers to add more complex theorems, and maintain the power provided by a full proof assitant such as Coq.

4 Conclusion

There has been a lot of work poured into constructing architectures that exhibit desired security properties. Unfortunately there hasn't been much work on the implementation and verification of these architectures so although on paper we are able to reason about the behavior of the system we don't have many guarantees about the behavior of our actual implementation. This is a failing of the tools used by hardware designers. We present a blueprint of how to leverage work done in the programming languages community and leverage automated theorem provers and dependent types as tools for specifying and verifying properties of hardware designs. We leave the more difficult problem of designing and verifying hardware designs to future work, the first goal is to build a tool that is powerful and robust enough to actually port real designs to it.

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] Edwin Brady, James Mckinna, and Kevin Hammond. Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types.

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

[4] Sally Browning and Philip Weaver. Designing tunable, verifiable cryptographic hardware using cryptol. In David S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 89–143. Springer US, 2010.

[5] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 54–65, New York, NY, USA, 2007. ACM.

[6] Adam Chlipala. *Certified programming with dependent types*, volume 20. MIT Press, 2011.

[7] 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.

[8] 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.

[9] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[10] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2014. http://www.cis.upenn.edu/ bcpierce/sf.

[11] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[12] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Automating formal proofs for reactive systems. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 452–462, New York, NY, USA, 2014. ACM.

[13] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 493–504, New York, NY, USA, 2009. ACM.

[14] 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.