Verifiable Security Templates for Hardware

William L. Harrison*, Gerard Allwein†
*Cyber Security Research Group, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
harrisonwl@ornl.gov
†US Naval Research Laboratory, Washington, DC, USA
gerard.allwein@nrl.navy.mil

***Abstract*—High-level synthesis (HLS) research generally focuses on transferring "software engineering virtues" (e.g., modularity, abstraction, extensibility, etc.) to hardware development with the ultimate goal of making hardware development as agile as software development. And recent HLS research has focused on transferring ideas and techniques from high assurance software formal methods to hardware development. Just as it has introduced software engineering virtues, we believe HLS can also become a vector for adapting software formal methods to the challenge of high assurance security in hardware. This paper introduces the Device Calculus and its mechanization in the Agda proof checking system. The Device Calculus is a starting point for exploring the formal methods and security of high-level synthesis flows. We illustrate the Device Calculus with a number of examples of formally verifiable security templates—i.e., functions in the Device Calculus that express common security structures at a high-level of abstraction.**

***Index Terms*—High Level Synthesis, High Assurance, Security, Type Systems, Proof Checking**

I. Introduction

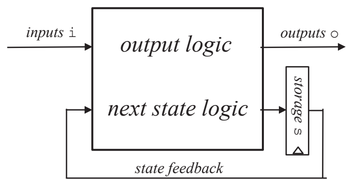
High-level synthesis (HLS) is usually motivated as a means for addressing the "programmability problem" in reconfigurable technology [1] by giving hardware designers software-like language abstractions and tools to achieve higher levels of productivity. More recently, HLS has been pursued as an avenue for producing high assurance hardware [2]–[4]. That is, by adopting ideas from software formal methods, the correctness, safety, and security of hardware designs can be formally analyzed and verified. But HLS abstractions come with a price. High-level abstractions in HLS flows must ultimately be translated to low-level HDLs, and this compilation process itself introduces a new source of assurance concerns. How do we know that an HLS hardware design is implemented faithfully by its compiler? If we prove a property of an HLS hardware design, how do we know that the circuit implementing it also possesses that property? Has the HLS compilation process itself introduced security flaws that may be exploited by an adversary?

Answering these kinds of questions requires formally verifying an HLS flow and there are prerequisites to doing so: both the HLS source and target languages must possess rigorous mathematical semantics; and these semantic specification(s) must be formalized in verification systems like Coq and Agda. Formal verification of software compilers is a well-established area within programming languages research that has, of late, enjoyed considerable success with realistic compilers [5]. Compiler verification involves proving that, for a source program p, the meaning of p according to the source semantics can be related to the target semantics of the compiled code for p. As with the case of software compiler verification, both the HLS source and target languages must be compared within a suitable semantic framework if the HLS flow is to be verified. The formal semantics of commodity HDLs like VHDL and Verilog is a known challenge [6] and so the choice of target language is also an important consideration. To achieve the highest levels of assurance, proofs of correctness, safety, and security properties should be developed and checked mechanically by an automated tools like Coq and Agda; mechanizing the HLS source and target semantics is, therefore, a prerequisite.

This article reports work-in-progress towards formal verification for a particular HLS flow called ReWire. Prior research has demonstrated ReWire's application to the development of high assurance hardware [4]This article focuses on one piece of this larger verification agenda: the development of a suitable mechanized semantic framework for the ReWire HLS flow that we call the *Device Calculus*. The Device Calculus is, in effect, a formalization of Mealy machines in the Agda proof assistant. The Device Calculus is a variety of λ-calculus with special operators for building Mealy machines and composing Mealy machines from existing ones. Rather than presenting a complete specification of the Device Calculus, we illustrate it with examples of *verifiable security templates*. These templates are functions that take Mealy machines as arguments and create composite devices with verifiable security properties. The technical details in this article have been kept to a minimum to enhance its readability for a larger audience. Follow-on publications will present the Device Calculus in precise detail. All Agda code presented here is available upon request.

The rest of this section presents related work. Section II introduces the Device Calculus and its mechanization in Agda. Section III presents a number of examples of security templates for hardware formulated in the Device Calculus. Section IV considers future work and concludes.

Related Work: The ReWire functional hardware description language is intended as a tool for producing high assurance hardware. ReWire is a subset of the Haskell functional programming language: every ReWire program is a Haskell program, but not necessarily vice versa. Previous work has described the design and implementation of ReWire [7] and its support for formal verification of reconfigurable hardware designs [4], [7]. Haskell was chosen as a host language for



```
record Mealy (i : Set) (s : Set) (o : Set) : Set
  where
    coinductive
    field mhead : i → s → (i × s × s × o)
          mtail : i → s → Mealy i s o
```

Figure 1: Mealy Machines are the usual design model for sequential circuits.

ReWire because Haskell is a pure functional language and, hence, amenable to formal methods itself.

One inherent challenge to mechanized reasoning (i.e., that performed with an automated proof tool like Agda) about hardware languages is that hardware devices are non-terminating by design and this non-termination must be represented one way or another. Reasoning (mechanical or otherwise) about infinite systems frequently involves a technique known as "coinduction." One principal advantage over prior research formalizing ReWire [4] (which uses Coq) is that, with the Device Calculus, security specifications such as the one from Procter et al. [7] can be readily transcribed into Agda. What makes this possible, we believe, is the deft handling of coinduction in Agda in comparison to Coq.

The types-as-properties view is the basis for verification systems like Coq and Agda that are based in dependent type theory. With the dependent type theory approach of Coq and Agda, program properties are formulated as types, and then verifying a program becomes a type-checking problem. Formal verification of hardware has been performed in this manner [2]–[4], [8]. Within language-based security [9], this view of security verification via type systems is also common. This research is part of an agenda seeking to adapt sofware language-based security ideas to hardware.

II. The Device Calculus in Agda

This section presents the Agda mechanization of the Device Calculus or, rather, the portion of it necessary to understand the security templates in Section III. This section is necessarily technical, although the authors endeavor to describe the material at a sufficiently high level so that readers without expertise in formal methods generally or Agda in particular can appreciate the basic approach.

Mealy machines (Figure 1, top), are a common model of sequential circuits used in hardware visualization and design. The sequential device in this takes two inputs on each clock cycle, external *inputs* i and internal *state feedback* from the register bank of *storage*, s. Based on these inputs, combinational logic computes the external *outputs* o and the *next state* s stored in the internal *storage* bank.

We assume that sequential circuits are intended to be non-terminating, and, consequently, the Agda representation of Mealy machines uses *coinductive* types to represent Mealy machines. Coinductive types and related reasoning techniques are usually used to represent and reason about (potentially) infinite structures like streams.

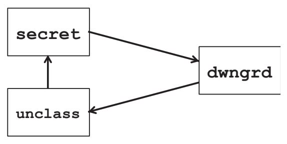
Figure 1 (bottom) presents the Agda formalization of the Mealy machine at the top of that figure using Agda's *coinductive record* syntax. Note that the `Mealy` type constructor parameterizes over the input, internal storage, and output types (`i`, `s`, and `o`, resp.) of a Mealy machine. There are two operations used to define `Mealy` machines, `mhead` and `mtail`. How these work is best explained through an example, which we provide below, but their basic intuition is simple. Given input, `i`, and the current internal storage, `s`, `mtail` produces the "next state" in the Mealy machine. Thus, for any `i` and `s`, there is always a next state (i.e., the machine never terminates). Given those same `i` and `s`, `mhead` produces a "snapshot" that records the current state of the circuit as a 4-tuple, $(\mathtt{i}, \mathtt{s}, \mathtt{s}', \mathtt{o})$. Here, `i` is the current input, `s` ($\mathtt{s}'$, resp.) represents internal storage at the beginning (end, resp.) of the clock cycle, and `o` is the output produced at the end of the clock cycle.

```
iter : ∀ { i s o : Set } → (i → s → (o × s)) → Mealy i s o
mhead (iter f) i s = (i , s , π₂ (f i s) , π₁ (f i s))
mtail (iter f) i s = iter f

sharedreg : ∀ { r : Set } → Mealy (Req r) r (Rsp r)
sharedreg = iter io
  where
    io : ∀ {r : Set} → Req r → r → (Rsp r × r)
    io nop r  = (nil , r)
    io (write r) _ = (ack , r)
    io read r = (val r , r)

 data Req (r : Set) : Set      data Rsp (r : Set) : Set
   where                         where
     nop  : Req r                  nil : Rsp r
     write : r → Req r             ack : Rsp r
     read : Req r                  val : r → Rsp r
```

Figure 2: A simple Device Calculus operator (`iter`) and an example of its use defining a register (`sharedreg`).

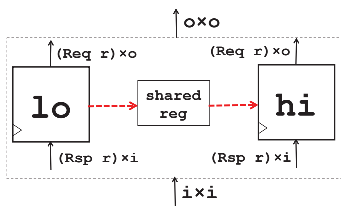
Assume one has a function, $\mathtt{f} : \mathtt{i} \to \mathtt{s} \to (\mathtt{o} \times \mathtt{s})$, that, for input values of type `i` and internal store `s`, returns a pair of type, $(\mathtt{o} \times \mathtt{s})$, consisting of the next output and updated internal store. One simple device of type `Mealy i s o` simply repeats function `f` *ad infinitum*, applying it to each new input occurring at each new clock cycle. This Device Calculus iteration operator is defined in Agda in Fig. 2 (top). In this definition, π_1 and π_2, are the left and right projections; e.g., $\pi_1(x, y) = x$. The type signature (first line) says that `iter` take a function of `f`'s type and returns a `Mealy i s o`. The second line defines the snapshot of `iter f` using `mhead` given the current input `i` and internal store `s`. The third line defines the next state transition using `mtail` given input `i` and current store `s`—there is only one state in the `iter f` device so the transition is particularly simple.

Using `iter`, Fig. 2 (middle) defines a simple register, `sharedreg` of type `Mealy (Req r) r (Rsp r)`, where the request and response types, `Req` and `Rsp`, are also defined in that figure. Note that the type of storage, `r`, is parameterized over within these definitions; `r` could be a single bit or a 64-bit word,



```
downgrade :
  ∀ { os dst od iu unclst u2s ou is secst : Set} →
    Mealy os dst od                          → -- dwngrd
    Mealy (iu × od) unclst (u2s × ou) → -- unclass
    Mealy (is × u2s) secst os               → -- secret
    (od × (u2s × ou) × os)                  →
    Mealy (iu × is) (dst × unclst × secst) ou
downgrade dwngrd unclass secret o₀ =
  feedback out₁ rte₁ o₀ (dwngrd <||> unclass <||> secret)
```

Figure 3: Downgrader from Rushby [10] (simplified to two security domains).



```
harness : ∀ { i s o r : Set }                 →
          ((Req r × o) × (Rsp r) × (Req r × o)) →
          Mealy (Rsp r × i) s (Req r × o) →
          Mealy (Rsp r × i) s (Req r × o) →
          Mealy (i × i) (s × r × s) (o × o)
harness o₀ lo hi
  = feedback out₂ rte₂ o₀ (lo <||> sharedreg <||> hi)
```

Figure 4: Dual Core Template from Procter et al. [7].

etc. The Device Calculus, in other words, inherits Agda's expressive polymorphism. The `io` function, when passed a `write r` request, replaces the current storage with the new value `r` and, when passed a `read` request, returns the current stored value.

While the Device Calculus has a number of operators, the only ones used in Section III are `iter`, `<||>`, and `feedback`; the types of the latter two are:

```
<||>     : ∀ {i₁ i₂ s₁ s₂ o₁ o₂ : Set} →
             Mealy i₁ s₁ o₁          →
             Mealy i₂ s₂ o₂          →
             Mealy (i₁×i₂) (s₁×s₂) (o₁×o₂)
feedback : ∀ {i₁ i₂ s o₁ o₂ : Set} → (o₁ → o₂) →
            (o₁ → i₂ → i₁) → o₁ → Mealy i₁ s o₁ → Mealy i₂ s o₂
```

An application of the parallelism operator, `m1 <||> m2`, places two devices, `m1` and `m2`, in parallel and in isolation from one another. Note that the input, internal storage, and output types of `m1 <||> m2` are just pairs of their respective component input, internal storage, and output types. If $\mathtt{m}_2$ is defined in terms of an existing device $\mathtt{m}_1$ (e.g., as in `feedback out rte` $\mathtt{o}_0$ $\mathtt{m}_1$) it is useful to think of the *output* and *routing* functions, `out` and `rte`, as combinational logic. (We refer to such functions as *output* and *routing* throughout the remainder.) The output function, $\mathtt{out} : \mathtt{o}_1 \to \mathtt{o}_2$, determines $\mathtt{m}_2$'s output directly from $\mathtt{m}_1$'s. The routing function, $\mathtt{rte} : \mathtt{o}_1 \to \mathtt{i}_2 \to \mathtt{i}_1$, takes $\mathtt{m}_1$'s output and $\mathtt{m}_2$'s input and feeds them back into $\mathtt{m}_1$. Device $\mathtt{m}_1$'s initial output is just $\mathtt{o}_0$.

III. Security Templates in the Device Calculus

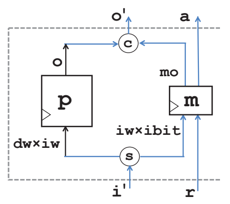
Template 1: Downgrader: The first template (Fig. 3) presents the Device Calculus formalization of the downgrader from Rushby [10]. A downgrader performs declassification—i.e., taking data from a higher security level and lowering it. Declassification breaks classic Goguen-Meseguer non-interference [11], because higher security level entities may communicate with lower security level entities, albeit only via a trusted intermediary (e.g., `dwngrd` in Fig. 3). Declassification is accomplished here simply with the routing function $\mathtt{rte}_1$ defined below in which the output of the `secret` device (`os` in the l.h.s. pattern) is passed to, and only to, the input of `dwngrd`. The output function, $\mathtt{out}_1$, ensures that only the output from the `unclass` device reaches the output of the composite `downgrade` device.

```
rte₁ : ∀ {od u2s ou os iu is : Set} →
        (od × (u2s × ou) × os) →
        (iu × is)               →
        (os × (iu × od) × (is × u2s))
rte₁ (od , (u2s , _) , os) (iu , is)
        = (os , (iu , od) , (is , u2s))
out₁ : ∀ {od u2s ou os : Set} →
        (od × (u2s × ou) × os) → ou
out₁ (_ , (_ , ou) , _) = ou
```

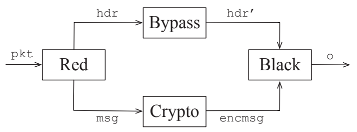
Template 2: Secure Dual Core: Fig. 4 presents the Device Calculus formulation that generalizes the secure dual-core template from Procter et al. [7]. Within this configuration, `sharedreg` is read-only for the `lo` processor and write-only for the `hi` processor, thereby enforcing a "no write down" security policy. Like the previous example, the work of restricting information flow takes place in the definition of the routing function $\mathtt{rte}_2$ in which a `write` request results in a `nop` signal to `sharedreg`: $\mathtt{rte}_2\ (\cdots, \cdots, (\mathtt{write}\ \mathtt{x}, \mathtt{o}_2))\ (\mathtt{i}_1, \mathtt{i}_2) = (\cdots, \mathtt{nop}, \cdots)$.

The security specification for the dual-core presented in Procter et al. [7], a form of non-interference [11], states, roughly speaking, that the `lo` processor's behavior is unchanged when the `hi` is replaced with a "no-op" processor that does nothing at all. This behavioral invariance of `lo` implies that, regardless of the behavior of the `hi` processor, no information can flow from `hi` to `lo`. The security specification in the aforementioned article is formulated as an equation and is proved "by hand" (i.e., not machine-checked). This same specification can be expressed directly in Agda as an equation in the Device Calculus; this is a significant advantage over prior work [4]. Follow-on publications will elaborate on this as space concerns do not permit so here.

Fig. 5 (top) presents the Device Calculus formalization of a hardware integrity monitor [12]. The main elements in the security template are a processor device `p` placed in parallel with monitor device `m`. On each clock cycle, `p` consumes an input of type `i` and produces an output of type `o`. In parallel on the same clock, monitor `m` consumes a pair of inputs (resp., of types `mi` and `r`) and produces a pair of outputs (resp., of types `mo` and `a`). Types `r` and `a` represent reset and alarm signals.



```
monitor : ∀ { dw iw sd o ibit r sm mo a o' i' : Set } →
            Mealy (dw × iw) sd o                 → -- processor
            Mealy (iw × ibit × r) sm (mo × a)    → -- monitor
            ((o × mo) → o')                      → -- combine
            (i' → ((dw × iw) × ibit))            → -- sample
            (o × (mo × a))                       → -- initial outputs
            Mealy (i' × r) (sd × sm) (o' × a)
monitor p m c s o0 = feedback (out3 c) (rte3 s) o0 (p <||> m)
```



```
cryptctrl :
  ∀ {pkt hdr msg s1 s2 s3 s4 hdr' encmsg o : Set} →
    Mealy pkt s1 (hdr × msg)      → -- Red
    Mealy hdr s2 hdr'             → -- Bypass
    Mealy msg s3 encmsg           → -- Crypto
    Mealy (hdr' × encmsg) s4 o → -- Black
    ((hdr × msg) × hdr' × encmsg × o) →
    Mealy pkt (s1 × s2 × s3 × s4) o
cryptctrl red byp cry blk o0
  = feedback out4 rte4 o0 (red <||> byp <||> cry <||> blk)
```

Figure 5: Hardware Integrity Monitor [12] (top) and Crypto-controller [10] (bottom). For reasons of space, the output and routing functions are not defined for either.

Monitor `m` connects to the processor `p`'s inputs and outputs via combinational logic `s` and `c` (resp., for sampling and combine functions). Appropriately typed `p`, `m`, `s`, and `c` can be composed into an `m`-monitored version of `p`, and this entire composition itself forms a device (indicated by the gray dotted line) with inputs, `i'` and `r`, and outputs, `o'` and `a`. This composed device produces an alarm `a` when the monitor `m` detects anomalous behavior based on `p`'s inputs.

Fig. 5 (bottom) presents a channel control scenario from Rushby [10]. This system takes a packet `pkt` as external input which is, in turn, split into a header `hdr` and message `msg` by Red. Bypass simply passes the header on, producing the presumably identical header `hdr'`. Crypto encrypts its input `msg`, producing `encmsg` as output. Black reassembles and outputs the header and encrypted payload, `o`.

IV. Future Work and Conclusions

This paper has introduced the Device Calculus, which is envisioned as a suitable semantic framework for formal verification of the ReWire HLS flow. The Device Calculus is mechanized within the Agda proof assistant, thereby supporting automated proof construction and checking. This short paper has not exhibited the full specification of the Device Calculus; we leave that and much else for follow-on publications. Instead, a number of security-related hardware constructions from the literature illustrated the Device Calculus.

Part of the burgeoning Chisel hardware ecosystem is the FIRRTL language ("Flexible Internal Representation for RTL"), which is an open-source hardware intermediate representation targeted by the Chisel compiler. The current ReWire compiler targets VHDL directly, but the lack of a formal semantics for VHDL renders formal verification of the current ReWire compiler essentially intractable. The authors became interested in retargeting the ReWire compiler to produce FIRRTL as an alternative to VHDL because FIRRTL is small, both well-designed and documented, and strongly typed—and, hence, an amenable target for formalization in the Device Calculus. To specify FIRRTL, the Device Calculus will be extended to multiple clock domains. Semantic models of multiple clock domains are rare (to the authors' best knowledge, only Czeck et al. [13] have published on this subject).

There are many formalisms for specifying hardware—how do you judge a new formalism like the Device Calculus? The evaluation of a formalism is, in part, a fundamentally qualitative or even aesthetic judgment: how easily and how aptly are useful designs expressed? The Device Calculus allowed expression of a number of verifiable security patterns from the literature that were succinct and, we would argue, straightforward as well. Agda's facilities for coinductive reasoning and structures provided a suitable formal foundation for developing the Device Calculus and this was, for the authors, a happy and somewhat unexpected discovery.

References

[1] D. Andrews, "Will the future success of reconfigurable computing require a paradigm shift in our research community's thinking?" Apr. 2015, keynote address, Applied Reconfigurable Computing.

[2] J. P. P. Flor, W. Swierstra, and Y. Sijsling, "Π-Ware: Hardware Description and Verification in Agda," in *Proc. TYPES*, 2015.

[3] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: a platform for high-level parametric hardware specification and its modular verification," *PACMPL*, vol. 1, pp. 24:1–24:30, 2017.

[4] T. N. Reynolds, A. Procter, W. Harrison, and G. Allwein, "The mechanized marriage of effects and monads with applications to high-assurance hardware," *ACM TECS*, vol. 18, no. 1, pp. 6:1–6:26, Jan. 2019.

[5] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009.

[6] M. J. C. Gordon, "The semantic challenge of Verilog HDL," *Proc. of 10th Annual IEEE LICS*, pp. 136–145, 1995.

[7] A. Procter, W. Harrison, I. Graves, M. Becchi, and G. Allwein, "A principled approach to secure multi-core processor design with ReWire," *ACM TECS*, vol. 16, no. 2, pp. 33:1–33:25, Jan. 2017.

[8] C. Salama, G. Malecha, W. Taha, J. Grundy, and J. O'Leary, "Static consistency checking for Verilog wire interconnects—using dependent types to check the sanity of Verilog descriptions," *Higher-Order and Symbolic Computation*, vol. 24, no. 1-2, pp. 81–114, 2011.

[9] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journ. on Sel. Areas in Commun.*, vol. 21, no. 1, Jan. 2003.

[10] J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI International, Tech. Rep. CSL-92-02, December 1992.

[11] J. Goguen and J. Meseguer, "Security policies and security models," in *Symposium on Security and Privacy*. IEEE, 1982, pp. 11–20.

[12] W. Harrison and G. Allwein, "Semantics-directed prototyping of hardware runtime monitors," in *Proc. RSP*, Oct. 2018.

[13] E. Czeck, R. Nanavati, and J. Stoy, "Reliable design with multiple clock domains," in *Proc. MEMOCODE*, 2006, pp. 139–148.