Language Abstractions for Hardware-based Control-Flow Integrity Monitoring

William L. Harrison
Department of Electrical Engineering & Computer Science
University of Missouri
Columbia, Missouri, USA
william.lawrence.harrison@gmail.com

Gerard Allwein
US Naval Research Laboratory
Washington, DC, USA
gerard.allwein@nrl.navy.mil

***Abstract*—Control-Flow Integrity (CFI) is a software protection mechanism that detects a class of code reuse attacks by identifying anomalous control-flows within an executing program. Hardware-based CFI has the promise of the security benefits of CFI without the performance overhead and complexity of software-based CFI: generally speaking, hardware-based monitors are more difficult to bypass, offer lower performance overheads than software-based monitors, and, furthermore, hardware-based CFI can be performed without the necessity of altering application binaries or instrumenting language compilers. Although hardware-based CFI is an active area of research and there is a growing literature describing CFI strategies at a high-level, there is, to the authors' best knowledge, no work on languages specially tailored to the specification and implementation of CFI monitors. This article presents a proof-of-concept domain-specific language with built-in abstractions for expressing control-flow constraints along with a compiler that targets the functional hardware description language ReWire. While the case study is small, it indicates, we argue, an approach to rapid-prototyping hardware-based monitors enforcing CFI that is quick, flexible, and extensible as well as being amenable to formal verification.**

I. Introduction

Embedded systems are everywhere, deployed in a wide variety of systems: automotive, medical, military, Internet-of-Things, etc. Embedded systems are lightweight, having restricted resources—in terms of both computing power and development costs—of necessity and by design compared to larger "industrial strength" computing systems. Embedded systems are also increasingly the focus of security exploitation [1]. The development cost of protections for embedded systems must likewise be kept lightweight: this article explores the application of ideas from programming language design to the construction of security mechanisms for embedded hardware in pursuit of this goal.

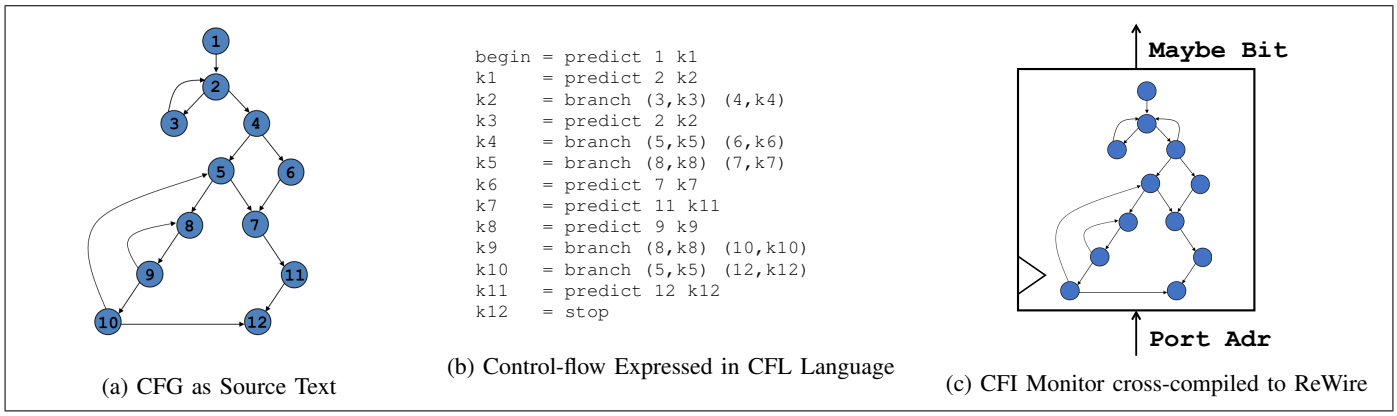
This article presents a proof-of-concept domain-specific language with built-in abstractions for expressing and generating run-time monitors in reconfigurable hardware that enforce control-flow integrity. Control-Flow Integrity (CFI) is an approach to software security [2] in which changes in control-flow within a program in execution are compared to a control-flow graph (CFG) for the program: control-flow not described by the program's CFG indicates an ongoing control-flow hijacking attack. Broadly speaking, there are two main technical challenges in implementing a CFI monitor: (1) the identification of an accurate control-flow graph (CFG) for the program in question and (2) the representation strategy of the monitor itself. There is a considerable body of research on CFI hardware [3] and software [4] implementation strategies since its inception.

This article focuses on challenge (2), although from a novel perspective—that of programming language design and domain-specific languages—and the result we present answers the question: how can a control-flow monitor for a program be derived directly and automatically from its CFG? The CFG for a program is taken as input, translated to a domain-specific language for expressing control-flow constraints called CFL (for Control-Flow Language). CFL is a DSL "embedded" in the Haskell functional language: it is defined in terms of Haskell and leverages Haskell's infrastructure (e.g., its type system and implementation).

Embedding in Haskell provides two immediate benefits for CFL and, consequently, for generation of hardware-based CFI monitors. Firstly, CFL programs, being embedded in Haskell, can leverage any number of techniques or tools associated with Haskell (e.g., strong-typing, automated test generation, formal verification via equational reasoning, etc.). Secondly, CFL programs may be translated to synthesizable VHDL using ReWire, which is a functional hardware description language also embedded in Haskell.

This research is based in functional languages and embedded domain-specific languages, but it is the authors' intention to make this article accessible to readers with no experience in functional programming whatsoever. We will endeavor to explain Haskell/ReWire notation throughout. Readers may consult the online codebase for this paper for further details. The remainder of this section considers related work and presents a high-level, non-technical summary of the results. Section II motivates the design and implementation of the CFL language. Section III summarizes the results presented here and discusses ongoing and future directions.

Related Work: Classic "stack-smashing" buffer-overflow exploits copy code into the targeted system which is then, one way or another, contrived to be executed. Countermeasures—e.g., data execution prevention (DEP) also known as write XOR execute (W⊕X)—were devised and widely deployed that



(a) CFG as Source Text

(b) Control-flow Expressed in CFL Language

(c) CFI Monitor cross-compiled to ReWire

Fig. 1: Language Abstractions for Hardware-based Control-Flow Integrity Monitoring. Throughout this article, instruction addresses are represented, without loss of generality, as integers rather than as machine words.

effectively prevent such code injection attacks. Code re-use attacks—e.g., return-into-libc attacks and their progeny [5]—circumvent DEP/W⊕X by hijacking control-flow and re-using the existing codebase on the targeted system. CFI, first proposed by Abadi et al. [2], detects such code re-use attacks as hijacked control-flow invariably diverges from the expected control-flow. There is a continuum of CFI strategies from "fine-grained," in which control-flow must strictly adhere to expectations at the instruction-level, to "coarse-grained", in which adherence is looser and tracked at the level of code regions (e.g., basic blocks).

The majority of CFI research is based in software in which applications and CFI monitoring code are woven together, where "weaving" is accomplished by, for example, instrumenting language compilers to include control-flow monitoring checks or by using binary translation techniques to alter applications with monitoring code. An excellent survey of software-based methods can be found in Burow et al. [4]. Software-based CFI suffers from implementation complexity as modifying language compilers and/or binary translation can be quite challenging techniques in, and of, themselves. Furthermore, recent work on the effectiveness of software-based CFI [6]–[8] indicates that many software-based approaches can be circumvented.

Indirect control-flow instructions—i.e., those with register targets like "`jmp eax`" in x86—are the source of imprecision in the calculation of CFGs, because it is impossible, via static analysis, to determine exactly the set of all possible flows from it. Accurate CFG generation is key to the success of any CFI approach; see Tan and Jaeger [9] for further discussion. To render the CFL flow as general as possible, this article treats CFGs as source input, leaving the issue of CFG generation for future work (discussed in Section III below).

An adjacency matrix representation of a CFG will be quite sparse, because, for any address `a`, there will be only a small handful of addresses `a′` for which there is a link from `a` to `a′`. One challenge of hardware-based CFI derives from this fact and has a negative impact on table-based approaches [10] to hardware CFI. Mao and Wolf [11] consider more efficient encodings of the adjacency matrix based on hashing. FSM-based approaches [12], [13] overcome the CFG sparseness problem by, in effect, inlining the permissible control-flow. A similar approach is taken with CFL, in which each control-flow constraint is expressed directly. At any point in CFL execution, there is a single such constraint being monitored and deviation from expected control-flows is detected at that single point. Importantly, this "inlining approach" obviates the need to represent the entire adjacency matrix form of the CFG. Muench et al [14] propose hardware-assisted CFI using transactional memory. Christoulakis et al. [15] integrate CFI capabilities directly into a SPARC SoC; their system, HCFI, weaves CFI monitoring directly into the hardware itself rather than a separate hardware monitor. For an excellent survey of hardware-based methods for CFI, please see Clercq and Verbauwhede [3].

High-level synthesis (HLS) from functional languages [16], [17] is a proposed remedy for the "programmability" problem [18] in reconfigurable technology. ReWire is a functional hardware description language that 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 [19], its support for equational reasoning about reconfigurable hardware [19], [20], and its use as a target for embedded DSLs [21]. ReWire is intended as a tool for producing high assurance hardware and the current work is a step towards formally verifying the security and integrity properties of monitored systems (although the current work does not address formal verification, leaving it for future work). To the best knowledge of the authors, this work is the first application of functional language-based HLS to the design and implementation of hardware CFI monitors.

Language Abstractions for Hardware-based Control-Flow Integrity Monitoring: Fig. 1 presents a high level overview of the language-based approach to generating CFI monitors in hardware described in this article. Fig. 1a takes as input source the CFG of the program to be monitored. This directed graph is transformed into a CFL program (Fig. 1b),

where CFL is the domain-specific language for expressing control-flow constraints. Each CFL program has a reserved start symbol, `begin`. A CFL constraint "`predict 2 k2`" means intuitively: "wherever I am, if the instruction address to be executed next is `2`, then proceed to `k2`; otherwise, sound the alarm." We describe CFL's syntax, semantics, and implementation in detail in Section II.

CFL program structure mirrors that of Haskell/ReWire—i.e., it is a set of (mutually recursive) equations. CFL programs are embedded in Haskell/ReWire by providing definitions for operations `predict` and `branch` as well as interface code for enabling and resetting the monitor; this is described in more detail below in Section II-E. This embedding is simply a compiler, taking CFL programs into Haskell/ReWire programs. The result of compiler the CFL program in Fig. 1a is a Haskell/ReWire program defining `cfimon`, which is the CFI monitor. Before continuing, we explain some relevant Haskell/ReWire syntax.

Note on Notation: ReWire has a built-in type constructor for devices, `Device`; e.g., `d :: Device i o` signifies that `d` describes a clocked device that, on each clock cycle, consumes an input and produces an output of types `i` and `o`, resp. The double colon `::` is read "has type"; e.g., "`x :: a`" says that expression or variable `x` has type `a`.

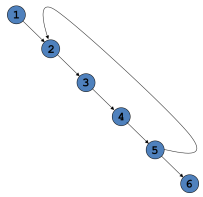
The type of the monitor generated by the Haskell/ReWire embedding is: `cfimon :: Device (Port Adr) (Maybe Bit)` and it is illustrated in Fig. 1c. The type `Adr` stands for the instruction address type appropriate to a particular application. The address type `Adr` is, for the purposes of this presentation, left unspecified, although typical instances would be a machine word of some fixed size—e.g., ReWire has built-in word types (e.g., `W8`, `W16`, and `W32` for 8, 16, and 32 bit words, resp.). Without loss of generality, addresses of type `Adr` are written as integers throughout. A detailed discussion of this type occurs in Section II-C, but, for now, think of `cfimon` as a synchronous device that accepts instruction addresses on its input port (i.e., values of type `Port Adr`) and produces each cycle an output of type `Maybe Bit`. Outputs of this type have the form `Nothing` (meaning `cfimon` is not operating) and `Just b`, signifying that `cfimon` is operating; if bit `b` is clear (set), then control-flow is normal (anomalous).

Because the `cfimon` is executable in Haskell, we can then write a test harness and test cases for `cfimon` completely in Haskell (see the codebase for complete details). The test harness has the following type:

```
test :: Device () (Port Adr)          ->
        Device (Port Adr) (Maybe Bit) ->
        [Maybe Bit]
```

An application of the harness, `test tst cfimon`, pipes the outputs from test device `tst` to the inputs of `cfimon`. We can devise a test device in Haskell/ReWire, call it `good`, that generates the following sequence of `Port Adr` outputs:

```
Enable, DontCare, DontCare, PC 1, DontCare,
PC 2, DontCare, DontCare, PC 3, DontCare,
PC 4, Reset, DontCare,...
```



(a) Input CFG

```
begin = predict 1 k1
k1    = predict 2 k2
k2    = predict 3 k3
k3    = predict 4 k4
k4    = predict 5 k5
k5    = branch (2,k2) (6,k6)
k6    = stop
```

(b) Generated CFL Program

Fig. 2: Running Example: CFI Monitor Generation. On the left is a simple CFG used as a running example and, on the right, is the CFL program generated automatically from it.

Or, we can devise a test device, call it `bad`, that generates the following outputs:

```
Enable, DontCare, DontCare, PC 1, DontCare,
PC 6, DontCare, DontCare, PC 3, DontCare,
PC 4, Reset, DontCare,...
```

Here, `Enable` and `Reset` makes `cfimon` begin and end scanning, resp., `PC a` signifies the fetch of instruction address `a`, and `DontCare` signifies no meaningful input.

Note that, as shown, the `good` (`bad`) output sequence do (do not) include anomalous control-flows according to the CFG in Fig. 1a. Using the GHCi Haskell interpreter, we may then perform the tests (underlined text is typed by user, the rest is produced by GHCi):

```
ghci> test good cfimon
   [Nothing,Just 0,Just 0,Just 0,Just 0,
    Just 0,Just 0,Just 0,Just 0,Just 0,
    Just 0,Just 0,Nothing,...
ghci> test bad cfimon
   [Nothing,Just 0,Just 0,Just 0,Just 0,
    Just 0,Just 1,Just 1,Just 1,Just 1,
    Just 1,Just 1,Nothing,...
```

Note that the `good` test never generates a `Just 1`, but the `bad` test does in the cycle following its producing `PC 6`. Once satisfied with `cfimon`, it can be translated to synthesizable VHDL using the ReWire compiler.

II. CFL: A Domain-Specific Language for Control-Flow Constraints

This section describes the Haskell/ReWire embedding of CFL that enables the testing of monitors using Haskell as well as their compilation to synthesizable VHDL by the ReWire compiler. We endeavor to describe Haskell/ReWire notation at a high level as we proceed although, of necessity, the description is very high-level. Throughout this section, we refer to a simple running example shown in Fig. 2 and use it to illustrate the structure, semantics, and implementation of CFL. Doing so allows the basic ideas to be provided without unnecessary excursions into technicalities of programming language design—readers may refer to the online codebase for more details.

A. CFG Representation for the CFL Flow

A simple example will illustrate the structure of the CFG representation. Consider the CFG in Fig. 2a which is defined below in Haskell as:

```
runningex :: CFG Adr
runningex = (1,es)
  where
    es = [1 :-> 2, 2 :-> 3, 3 :-> 4, 4 :-> 5,
          5 :=> (2,6), Halt 6]
```

A CFG is a pair consisting of the start address and a list of edges (resp., `1` and `es` above). Each node carries an address (i.e., `1` through `6`). Edges come in three forms, each corresponding to the number of successors the source node has. In Fig. 2a, node `1` has the single successor node `2`, and, hence, the edge `1 :-> 2` is in `es`. Similarly, node `5` has exactly two successors, nodes `2` and `6`, and, hence, the edge `5 :=> (2,6)` is in `es`. Node `6` has no successors in Fig. 2a, and so it is represented by `Halt 6` edge. The type declaration of edges is parameterized over the address type `a` and is given by the following Haskell declaration:

```
data Edge a = a :-> a       -- one successor
            | a :=> (a,a)   -- two successors
            | Halt a        -- no successors
type CFG a  = (a,[Edge a])
```

A CFG is a pair, `(a,es)`, where `a` is the initial address of the CFG and `es :: [Edge a]` (read "[...]" as "list of").

A control-flow graph (e.g., Fig. 1a and Fig. 2a) is a directed graph expressing the expected control-flows within a program. For our purposes, we assume that the vertices (i.e., nodes) within the graph contain only the addresses of instructions within the program in question. An edge, `n`→`m`, in the graph indicates that, if executing the instruction at address `n`, the instruction at address `m` may be executed next with no intervening instructions. Conversely, if an edge, `n`→`m`, is not present in the CFG, then the change of control from `n` to `m` is not permitted. CFGs are a fundamental data structure within language compilers and, in that context, they may be decorated with all manner of data (e.g., results of static analyses) that we do not include. For language compilation, the CFG vertices will frequently denote *basic blocks*, which are straight-line code (i.e., no intermediate jumps or calls) with single points of entry and exit within the program being compiled. The CFG format we assume represents single instructions which is consistent with "fine-grained" CFI, although looser, "coarse-grained" CFI has been explored [4] which might make use of basic block information.

For any node `n` in a CFG, the number of edges proceeding from it has a fixed upper bound as a consequence of the semantics of the typical machine languages. An instruction at a termination point for the program would have no edges proceeding from it. A non-control-flow instruction—e.g., "`push eax`" in x86—has a single next instruction and would, therefore, induce precisely one directed arc in the CFG. A control-flow instruction—e.g., "`jz` *label*" in x86—would induce precisely two directed arcs in the CFG.

B. Generating CFL programs from CFGs

The input CFG is translated into CFL by mapping a CFG edge `a :-> a'` into an CFL equation of the form `ka = predict a' ka'`, where `ka` and `ka'` are fresh variables. A CFG edge `a :=> (a1,a2)` is mapped into a CFL equation of the form `ka = branch (a1,ka1) (a2,ka2)`, where `ka`, `ka1`, and `ka2` are fresh variables. This mapping on edges is defined by the pseudo-Haskell function `e2cfl`:

```
e2cfl :: Edge Adr -> CFL
e2cfl (a :-> a')       = ka = predict a' ka'
e2cfl (a :=> (a1,a2)) = ka = branch (a1,ka1) (a2,ka2)
e2cfl (Halt a)         = ka = stop
```

To write the actual definition of `e2cfl`, we would have to introduce the abstract syntax for CFL, and this seems like an unnecessary detour in that the definition above conveys the essence of the definition found in the codebase. For `(i,es) :: CFG Adr`, the CFL constraint equation for the reserved word `begin` is given similarly depending on whether `i` has one, two, or no successors.

C. The Device Type of a CFI Monitor

The Haskell/ReWire type of a CFI monitor is `Device (Port Adr) (Maybe Bit)`. The `Bit` type and `Port` type constructor are defined as:

```
data Bit    = C | S  -- clear and set, resp.
data Port a = PC a | DontCare | Enable | Reset
```

A value of type `Port W8`, for example, will have the form: `PC w` for some `w :: W8`, `DontCare`, `Enable`, or `Reset`. Form `PC w` signifies that instruction address `w` is available at the port. Obviously, `DontCare` stands for a non-informative "don't care" input. Inputs `Enable` and `Reset` indicate that the CFI monitor should begin and cease, respectively, monitoring control-flow.

The output type of a CFI monitor is `Maybe Bit`. The `Maybe` type constructor, built-in to Haskell/ReWire, is defined below:

```
data Maybe a = Just a | Nothing
```

A `Nothing` output signifies that control-flow monitoring is not currently underway. A `Just b` output signifies that control-flow monitoring is currently being performed and `b :: Bit` is its current status: `C` meaning "*all flow legal so far*" and `S` meaning "*illegal flow has occurred*."

D. High-level Structure of Generated CFI Monitors

Fig. 3 presents a high-level structural account of CFI monitors generated from CFL programs. The diagram has the shape of a state machine, although the diagram is at a higher level than that. In particular, each of the labelled boxes—`cfimon`, `begin`, `alarm`, and `stop`—are Haskell/ReWire definitions that will be given shortly. The dashed-lined box—labelled "Inlined CFG"—is the code generated by the Haskell/ReWire embedding of CFL discussed in detail in the next section.

The monitor starts at `cfimon`. If `cfimon` receives any input other than `Enable`, it makes no transition. On `Enable`, the monitor transitions to `begin`, which will start the CFI monitoring activity. Without loss of generality, the diagram in Fig. 3 assumes that the first instruction address to be checked is `1` and that the equation for `begin` is of the form, `begin = predict 1 k`; it could, in practise, be a `branch` or `stop` constraint. If the input address is not `1`, it transitions to `alarm`. Within the "Inlined CFG," any anomalous control-flows will result in transition to `alarm` while program termination will result in transition to `stop` (from where it will immediately

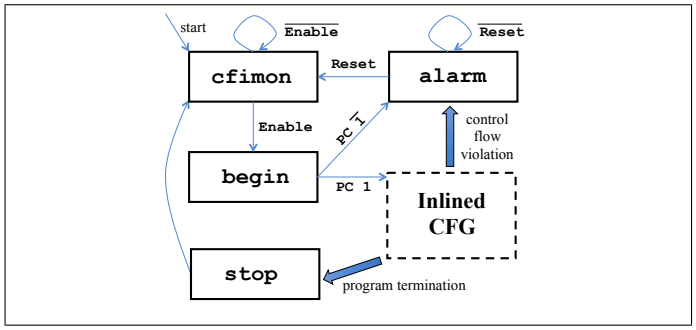


Fig. 3: Structure of a Generated CFI Monitor

transition back to `cfimon`). The monitor remains in `alarm` until it receives a `Reset` signal.

E. Haskell/ReWire Embedding of CFL

This section defines the Haskell/ReWire embedding of CFL. Part of the embedding consists of giving definitions for `predict` and `branch` operations shown in Figures 1b and 2b. Rather than give the definitions for the Haskell/ReWire embedding compiler (which can be found in the codebase), we describe the embedding by explaining its output on the running example of Fig. 2b.

Note on Notation: Haskell/ReWire use "do notation" to chain together `Device` operations (Fig. 4a) in which n operations are chained together using `do`. First, $\texttt{operation}_1$ is executed, producing value $\texttt{i}_1$, then $\texttt{operation}_2$ is executed, producing value $\texttt{i}_2$, and so on, until the final operation, $\texttt{operation}_n$, is reached. The value is produced by $\texttt{operation}_n$ is the value returned by `sequence`. The `signal` operation is used to receive input and produce output each clock cycle. The code snippet (Fig. 4b) sets the output port to `o` and waits to receive the new input `i`. Intuitively speaking, `signal`ing indicates the dividing line between clock cycles. The `signal o` occurs at the end of the current clock cycle and the receipt of input `i` marks the beginning of the next clock cycle in which `continue` may now process `i`.

The `cfimon` operation is a loop that waits until an `Enable` signal is received on the input port. While it waits, `cfimon` signals `Nothing` to indicate that the CFI monitor is inactive. Each cycle, it receives from the input port (`pa`) and pattern-matches against it. If `Enable`d, it continues to `begin`.

```
cfimon :: Device (Port Adr) (Maybe Bit)
cfimon = do
   pa <- signal Nothing
   case pa of
     Enable -> begin
     _      -> cfimon
```

The fourth line above takes the input from the port, `pa :: Port Adr` and pattern matches against it with a `case` expression. Each line in a `case` expression has the form "*pattern* `->` *value*". Above, if the *pattern* matches `pa`, it returns corresponding *value*. Pattern matching proceeds in top to bottom order. The underscore "_" is a wildcard pattern.

The `begin` operation signals `Just C` to indicate that the monitor is operational and has not observed a control-flow violation. If its input `pa` represents an instruction address (i.e., has form `PC a'`), it checks that the address is the starting address of the code and proceeds to `k1`; otherwise, it transitions to `alarm` to indicate the control-flow violation.

```
sequence = do
   i1     <- operation1
             ⋮
   i(n-1) <- operation(n-1)
   operationn
```

(a) Haskell/ReWire "do" notation.

```
do
  i <- signal o
  continue
```

(b) ReWire's `signal` operator.

Fig. 4: Notation: Chaining and Signaling in Haskell/ReWire.

```
begin :: Device (Port Adr) (Maybe Bit)
begin = do
   pa   <- signal (Just C)
   case pa of
     PC a' | a'==1      -> k1
           | otherwise  -> alarm
     DontCare           -> begin
     Enable             -> begin
     Reset              -> cfimon
```

The `alarm` operation repeatedly signals `Just S`, signifying that an anomalous control-flow has been identified, until it receives a `Reset` signal. Upon `Reset`, it transitions to `cfimon`, thereby restarting the monitor.

```
alarm :: Device (Port Adr) (Maybe Bit)
alarm = do
  pa <- signal (Just S)
  case pa of
    Reset -> cfimon
    _     -> alarm
```

The `stop` operation indicates that the program has terminated without incident and the monitor should cease operation and return to waiting for an `Enable` signal.

```
stop :: Device (Port Adr) (Maybe Bit)
stop = cfimon
```

We illustrate the Haskell/ReWire embedding of `predict`, `branch`, and `stop` constraints by the translation of several equations from Fig. 2b. Fig. 5 shows the translation of several such equations. Within the translation of `k1 = predict 2 k2`, the input from the port, `pa`, is received and checked using a `case` expression. If an instruction address `a'` is received, it transitions to `k2` if `a'` is the expected next address; if `a'` is not the expected address, a control-flow violation has occurred and it transitions to `alarm`. Given `DontCare` or `Enable`, it remains at `k1`. For `Reset`, it restarts the monitor by transitioning to `cfimon`. The other cases are similarly defined.

III. Summary, Conclusions, and Future Work

Development costs for security mechanisms suited to the diversity of embedded systems could be kept lower if development tool flows supported "software engineering virtues" of abstraction, modularity, extensibility, etc.: the more quickly a particular protection mechanism can be adapted or extended to meet the needs of a particular embedded system, the less expensive it is. This challenge motivates the approach taken

```
-- Embedding of: k1 = predict 2 k2
k1 :: Device (Port Adr) (Maybe Bit)
k1 = do
  pa   <- signal (Just C)
  case pa of
   PC a' | a'==2     -> k2
         | otherwise -> alarm
   DontCare          -> k1
   Enable            -> k1
   Reset             -> cfimon

-- Embedding of: k5 = branch (2,k2) (6,k6)
k5 :: Device (Port Adr) (Maybe Bit)
k5 = do
  pa   <- signal (Just C)
  case pa of
   PC a' | a'==2     -> k2
         | a'==6     -> k6
         | otherwise -> alarm
   DontCare          -> k5
   Enable            -> k5
   Reset             -> cfimon

-- Embedding of: k6 = stop
k6 :: Device (Port Adr) (Maybe Bit)
k6 = stop
```

Fig. 5: Haskell/ReWire Embedding of CFL: Running Example

here: i.e., the application of programming languages ideas—specifically, embedded domain-specific languages—to the construction of security mechanisms for embedded hardware.

This article presents a proof-of-concept domain-specific language, CFL, with abstractions expressing control-flow constraints along with an embedding into the ReWire functional HDL (and, consequently, an embedding into Haskell). These embeddings serve dual purposes as a means for both developing and implementing of hardware-based CFI monitors and, potentially, other varieties of hardware-based security monitors as well. Hardware-based security monitors (in this case, CFL monitors) may be type-checked, tested, and formally verified just as any Haskell program using existing tools and techniques (e.g., Haskell's strong type system, the GHC Haskell compiler, stepwise development, etc.). This paper has left formal verification for future work, but we would argue that it is significant, in and of itself, that hardware-based runtime monitors can be described in a framework in which there are many well-known paths forward for formal verification. Once design, testing, and verification goals are met, the monitors can be translated to synthesizable VHDL using the ReWire compiler.

What has not been addressed in this article is performance. Calculating accurate CFGs is a challenge in itself [9] and we have left performance analysis and tuning for follow-on research. Ongoing research has developed Haskell/ReWire models for RISC-V (specifically, the 32-bit integer ISA) and for the Xilinx MicroBlaze soft processor. In future work, we will leverage these ReWire models, along with tool suites for RISC-V and MicroBlaze, to perform an extensive "test and measure" study of the flow described in this article. All that being said, previous work has established that the ReWire compiler produces circuits with very good timing and space characteristics in the domain of regular expression compilation [21] and the CFI monitors developed here are quite similarly structured, so there is good reason to believe that similar performance characteristics are achievable.

References

[1] "The 5 worst examples of IoT hacking and vulnerabilities in recorded history." [Online]. Available: https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities/

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005, pp. 340–353.

[3] R. de Clercq and I. Verbauwhede, "A survey of hardware-based control flow integrity (CFI)," *CoRR*, vol. abs/1706.07257, 2017.

[4] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 16:1–16:33, Apr. 2017.

[5] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM CCS*, 2007, pp. 552–561.

[6] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *ACM Conference on Computer and Communications Security*, 2015.

[7] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15, 2015, pp. 161–176.

[8] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," *2014 IEEE Symposium on Security and Privacy*, pp. 575–589, 2014.

[9] G. Tan and T. Jaeger, "CFG construction soundness in control-flow integrity," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, ser. PLAS '17, 2017, pp. 3–13.

[10] Z. Guo, R. Bhakta, and I. G. Harris, "Control-flow checking for intrusion detection via a real-time debug interface," in *2014 International Conference on Smart Computing Workshops*, Nov 2014, pp. 87–92.

[11] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 847–854, June 2010.

[12] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 12, pp. 1295–1308, Dec 2006.

[13] M. Rahmatian, H. Kooti, I. G. Harris, and E. Bozorgzadeh, "Hardware-assisted detection of malicious software in embedded systems," *IEEE Embedded Systems Letters*, vol. 4, no. 4, pp. 94–97, Dec 2012.

[14] M. Muench, F. Pagani, Y. Shoshitaishvili, C. Kruegel, G. Vigna, and D. Balzarotti, "Taming transactions: Towards hardware-assisted control flow integrity using transactional memory," in *Research in Attacks, Intrusions, and Defenses*, F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds. Cham: Springer International Publishing, 2016, pp. 24–48.

[15] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced control-flow integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16. New York, NY, USA: ACM, 2016, pp. 38–49.

[16] P. Gammie, "Synchronous digital circuits as functional programs," *ACM Comput. Surv.*, vol. 46, no. 2, pp. 21:1–21:27, Nov. 2013.

[17] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," in *DAC*, 2012, pp. 1216–1225.

[18] D. Andrews, "Will the future success of reconfigurable computing require a paradigm shift in our research community's thinking?" Keynote, ARC, 2015, http://hthreads.csce.uark.edu/mediawiki/images/d/d8/Arc-presentation.pdf.

[19] A. Procter, W. L. 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.

[20] I. Graves, A. M. Procter, W. Harrison, and G. Allwein, "Provably correct development of reconfigurable hardware designs via equational reasoning," in *FPT*, 2015, pp. 160–171.

[21] I. Graves, A. Procter, W. Harrison, M. Becchi, and G. Allwein, "Hardware synthesis from functional embedded domain-specific languages: A case study in regular expression compilation," in *ARC*, 2015, pp. 41–52.