Semantics-directed Prototyping of Hardware Runtime Monitors

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*—Building memory protection mechanisms into embedded hardware is attractive because it has the potential to neutralize a host of software-based attacks with relatively small performance overhead. A hardware monitor, being at the lowest level of the system stack, is more difficult to bypass than a software monitor and hardware-based protections are also potentially more fine-grained than is possible in software: an individual instruction executing on a processor may entail multiple memory accesses, all of which may be tracked in hardware. Finally, hardware-based protection can be performed without the necessity of altering application binaries. This article presents a proof-of-concept codesign of a small embedded processor with a hardware monitor protecting against ROP-style code reuse attacks. While the case study is small, it indicates, we argue, an approach to rapid-prototyping runtime monitors in hardware that is quick, flexible, and extensible as well as being amenable to formal verification.**

***Index Terms*—Reconfigurable architectures, Hardware security, High level synthesis, Model driven development**

I. Introduction

Given their growing prevalence across a wide swath of application domains (e.g., automotive, military, medical, Internet-of-Things, etc.), embedded systems have increasingly become targets for security exploitation. Embedded systems are also lightweight systems—of necessity and by design—and, therefore, lack the computing resources to support the "industrial strength" security protections of larger computing systems. Consequently, any approach to securing embedded systems must also be lightweight, in terms of both resource consumption (e.g., power consumption, FPGA fabric real estate, etc.) and development cost.

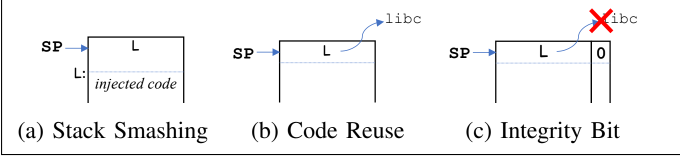
Development costs for security mechanisms suited to the diversity of embedded systems could be kept lower if development tool flows supported "software engineering virtues" of abstraction, modularity, extensibility, etc.: the more quickly a particular protection mechanism can be adapted or extended to meet the needs of a particular embedded system, the less expensive it is. Tool flows with software engineering virtues can reduce time-to-deployment for embedded hardware generally and, in particular, for security mechanisms. However, it is generally recognized that reconfigurable technology has a "programmability" problem [1] and therein lies the challenge: can we develop useful security mechanisms for embedded hardware that are also suitably adaptable, reprogrammable, and extensible?

This paper substantiates the affirmative answer to this question, presenting a proof-of-concept codesign of a processor with a hardware runtime monitor enforcing a stack integrity mechanism that protects against code reuse attacks. The approach we take relies heavily on the pure functional languages—Haskell and ReWire—because of their indigenous "software engineering virtues" (abstraction, modularity, expressiveness, etc.) and because of their susceptibility to formal methods. ReWire is a functional hardware description language (HDL) which is a subset of the Haskell functional language—i.e., every ReWire program is a Haskell program, but not vice versa.

This paper explores Haskell and ReWire as vehicles for the model-driven codesign of a processor and memory protection runtime monitor in hardware. Both the processor and monitor are "roughed out" in Haskell first in Sections IV and V. The advantage of this is significant: developing the hardware prototypes *first* as functional programs in Haskell is relatively quick, because the developer can leverage Haskell tools and concepts (e.g., its strong type system, GHC interpreter, stepwise development, etc.) as a means of quickly formulating, testing, and debugging new features of the codesigned models. Compilation to VHDL can then be accomplished by the ReWire compiler, as long as the codesigned Haskell models are within the ReWire subset.

It is the authors' intention to make this article accessible to readers with no experience in functional programming and, that being said, we will endeavor to explain Haskell/ReWire notation throughout. Some material has been included for completeness' sake, understanding the details of which are not necessary to comprehending this work and we label that material accordingly. All of the code discussed in this article is available online.

Section II describes the protection mechanism in detail. Section III presents a high-level overview of the case study to make its significance apparent without delving into the details. The processor UNSAFE is defined in ReWire in Section IV. The codesign of the stack integrity monitor is described in Section V. Related work is discussed in Section VI and future work and conclusions are presented in Section VII.



(a) Stack Smashing (b) Code Reuse (c) Integrity Bit

Fig. 1: (a) With classic stack smashing, code is injected (e.g., via buffer overflow). (b) Code reuse attacks inject control-flow rather than code. (c) Affixing a shadow bit to track integrity.

```
data UNSAFE                  decode :: W8 -> UNSAFE
     = Push W4               decode w = case w of
     | Call                    0b0001d3d2d1d0 -> Push 0bd3d2d1d0
     | Ret                     0b0110____     -> Call
     | Add                     0b0111____     -> Ret
     | Output                  0b0000____     -> Add
     | Load                    0b0010____     -> Load
     | Store                   0b0011____     -> Store
     | Jump                    0b0100____     -> Jump
     | Bnz W4                  0b0101d3d2d1d0 -> Bnz 0bd3d2d1d0
     | Nop                     0b1000____     -> Output
                               _              -> Nop
```

Fig. 2: Instruction Set Mnemonics as Haskell Datatype (l) and Its Decode Function in Haskell (r). `W4` and `W8` are the built-in ReWire types for 4 and 8 bit words.

II. Stack Integrity & the UNSAFE Instruction Set

This section motivates the stack integrity mechanism for which we develop a monitor as well as the instruction set, UNSAFE, used in this case study. UNSAFE is named in contradistinction to SAFE [2] to emphasize the absence of protection mechanisms in UNSAFE's semantics.

A. Stack Integrity Check

This section presents an overview of a memory protection mechanism that is a countermeasure against "code reuse" style attacks. The stack integrity mechanism presented here was integrated into the "SAFE" formal semantics [2] for a simple instruction set architecture, about which we will have more to say later.

Fig. 1a illustrates a classic "stack smashing" code injection attack. Code is injected into the host system's stack or heap at address `L` via, perhaps, a buffer-overflow exploit, and then control is shifted to that payload in some manner (in Fig. 1a, executing a "return from call" instruction would accomplish this shift of control). A number of countermeasures against code injection attacks were proposed and widely adopted, including DEP (data execution prevention) a.k.a. $W \oplus X$ (write xor execute). $W \oplus X$ insists that, for any address a, a program may write data to a or jump to a, but not both, and thereby effectively prevents code injection attacks.

Code reuse attacks inject control-flow instead of code as a means of circumventing $W \oplus X$-style protections. As illustrated in Fig. 1b, the attack contrives to configure the stack for a call into a codebase presumed to exist on the target host (here, the `libc` library). This tactic forms the basis of "return-into-lib" exploits and their progeny [3].

A countermeasure against code reuse style attacks (Fig. 1c) attaches an extra integrity bit, invisible from software, to mark whether stack items are legitimate addresses to return to. An address is legitimate if, for example, it is a return address pushed onto the stack resulting from executing a call instruction and, in which case, its integrity bit is set to `1`. However, if the stack item were the result of a normal calculation, it might be regarded as an illegitimate code pointer and, as such, the underlying hardware should set its integrity bit set to `0`. Treating that stack item as a legitimate address (e.g., "returning into it" as depicted in Fig. 1c) should be recognized as an integrity fault by the underlying hardware and handled accordingly. For example, the code snippet, "`push 0b1100; ret`", should always generate an integrity fault because the data word stored by `push` will have the `0` integrity bit affixed (`push` not being a call instruction), and so executing `ret` will set off the alarm. The extra integrity bit affixed to each stack item is an example application of shadow memory [4], [5].

B. Designing a Monitor for Stack Integrity

Fleshing out the monitor design for enforcing the aforementioned stack integrity protection mechanism requires making some reasonable assumptions about the underlying architecture and instruction set. We assume a Harvard architecture—i.e., code and data are stored in distinct address spaces. The choice of instruction set is important as well because the memory access patterns of call and return instructions as well as their encodings vary across instruction sets. Memory access patterns are important for this particular protection mechanism because the monitor must recognize when a word being:

1) stored in memory is a return address being pushed; and
2) read from memory will be used as a return address.

Recognizing these situations requires observing when a call or return is being executed (which depends on the instruction encodings and access to the instruction word arriving from instruction memory) as well as determining which memory access occurring during call or return execution involves the writing or reading (resp.) of code addresses. The recognition of these access patterns will be elaborated on further below in Section V.

C. The UNSAFE Instruction Set

From the preceding discussion, it is clear that monitoring for stack integrity necessarily depends on the instruction set semantics. The opposite—that the instruction set semantics depends on the monitor somehow—should not be the case. It is a methodological goal that instruction set semantics (or, rather, the processor implementing those semantics) be independent of the monitoring activities in the sense that, in the absence of an anomalous situation, the observable behavior of the processor should be equivalent whether monitored or not. This is what is achieved in this proof-of-concept study.

The UNSAFE instruction set used in this proof-of-concept is precisely that of Azevedo de Amorim et al. [2]. UNSAFE is a simple stack machine language, represented in Fig. 2 (l). Fig. 2 (r) presents the instruction `decode` function.

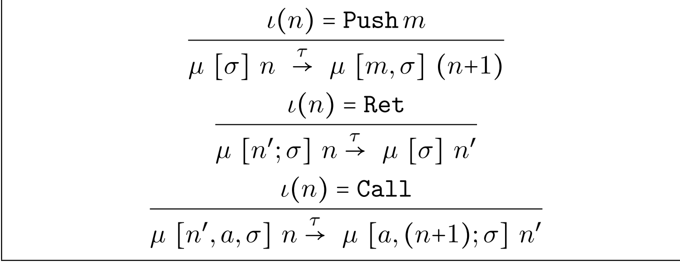
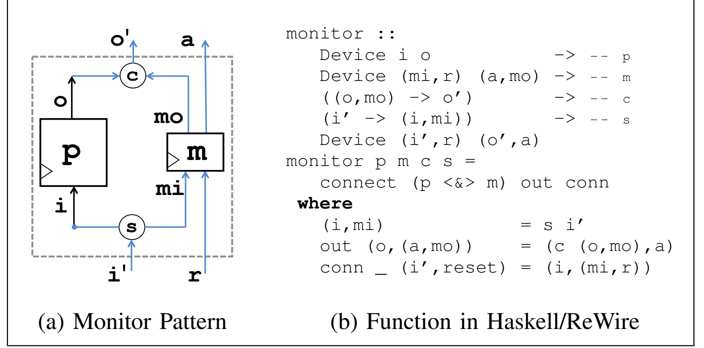


Fig. 3: Simplified form of the SAFE Semantics with Security/Integrity Checks (taken from Fig. 3 of Azevedo de Amorim et al [2]).



(a) Monitor Pattern (b) Function in Haskell/ReWire

Fig. 4: (a) Generic Hardware Monitor Pattern and (b) Its Realization as a function in Haskell/ReWire.

Note on Notation: The first line of the `decode` declaration is its type declaration, which indicates that `decode` is a function that takes a `W8` word as input and produces an `UNSAFE` instruction as output. The second line takes the function argument, `w` and pattern matches against it with a `case` expression; each subsequent line has the form "*pattern* -> *value*". If the *pattern* matches `w`, it returns *value* and pattern matching proceeds in top to bottom order. The underscore "_" is a wildcard pattern. We have taken some liberties here with the concrete syntax of Haskell patterns for readability's sake; in particular, there is no built-in syntax for binary numbers.

D. UNSAFE & Its SAFE Semantics

The ReWire semantics of UNSAFE is presented in part in Fig. 8 in Section IV and we defer its discussion until then. The remainder of this section discusses the original SAFE semantics [2] for UNSAFE instruction set. This section is not strictly necessary to the understanding of this article, and, although it does motivate the memory model presented in Section IV-B, it assumes some familiarity with programming language semantics.

There is an enormous body of work on the semantics of imperative software languages that are organized around a notion of state transformer in one form or another. That is, an imperative action is defined as taking an input state and transforming it into an output state and composed actions "thread" the state through actions in succession. There are denotational, relational, and operational formulations of this idea that view as well as that are so well-known as to obviate the need for further description.

The standard software view of state—a.k.a., "memory" or "store"—is incomplete in the context of hardware where memories are generally devices operating in parallel with the processor. A processor reading and writing to and from memory involves a potentially sequence of interactions in parallel according to a hand-shaking protocol of some form and such parallelism, bound up as it is intrinsically with a notion of timing, simply does not fit within the usual notion of state transformer.

Figure 3 presents the formal specification of several UNSAFE instructions as a small-step semantics; this is a simplified form of the published semantics with the infrastructure for information flow control erased. At first glance, this semantics is just what one would expect for a simple stack language. The transition relation takes an instruction memory (ι), a memory (μ), a stack ($[\sigma]$), and a program counter (n), and produces an output (in these cases, τ) and resulting memory, stack, and program counter.

There is a notationally-subtle stack integrity mechanism integrated into these rules. Notice that the $\texttt{Push}\, m$ instruction puts m at the top of the stack with a "," while the `Call` instruction pushes the return address $(n{+}1)$ on the stack with a ";". Furthermore, the `Ret` rule only applies to a ";" address (i.e., one created by `Call`) and so The distinction between "," and ";" in the semantics serves to cause misbehaving programs to "get stuck"; i.e., to reach a configuration in which no rules apply to signify the breaking of policy by a program. The semantics in Fig. 3 gives a crisp impression as to what the stack integrity mechanism from earlier in this section is intended to do, its model of memory is still that of software—i.e., a state transformer. As such, it does not provide an path forward to hardware implementation.

III. Semantics-Directed Prototyping of a Hardware Runtime Monitor

This section presents an high-level overview of the methodology developed in this paper with the purpose of providing the reader with "roadmap" indicating where we are going before delving deeply into the details. A generic monitor pattern is described in Section III-A and then formulated as a Haskell/ReWire function `monitor` in Section III-B. In Section III-C, by applying `monitor` to the processor `unsafe` and monitor `stack_integrity` (both developed below in Sections IV and V, resp.), a new processor `safer` is created in which stack integrity is monitored. Finally, in Section III-D illustrates how the operation of both the unmonitored and monitored processors may be simulated in Haskell.

A. Generic Monitor Pattern

Fig. 4a presents a diagrammatic design of a class of hardware monitors. The main elements in the design pattern 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,

```
sequence = do
   i_1     <- operation_1
        ⋮
   i_(n-1) <- operation_(n-1)
   operation_n
```

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

(a) Haskell/ReWire “do” notation. (b) ReWire’s `signal` operator.

Fig. 5: Notation: Chaining and Signaling in Haskell/ReWire.

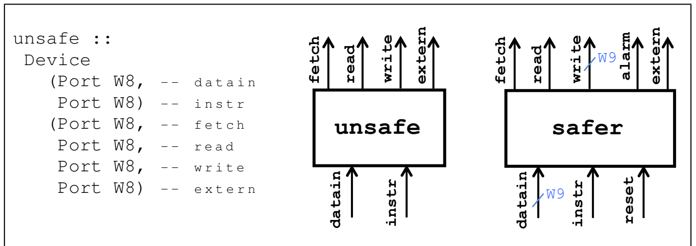


Fig. 6: The `unsafe` & `safer` processors.

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 `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.

Note on Notation.: The double colon :: is read “has type”; e.g., “`x` :: `a`” says that expression or variable `x` has type `a`. 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.

B. Expressing the Monitor Pattern in Haskell/ReWire

The design pattern in Fig. 4a is generic in the sense that it parameterizes over processor `p`, monitor `m`, and the sample and combine functions, `s` and `c`. This monitor pattern can be expressed in Haskell/ReWire directly in Fig. 4b. by the `monitor` function. The processor and monitor are placed together in parallel with the ReWire parallelism operator, `p <&> m`, and are then connected with the `connect` operator (called `refold` previously [6]). The connection and output functions, `conn` and `out`, are defined in terms of the sample and combine functions, `s` and `c`, and passed to `connect`.

N.b., the significance of the code in Fig. 4b is that monitor design patterns can be expressed directly as Haskell functions, so that designs created by applying `monitor` are themselves executable Haskell code, thereby enabling testing directly on the source code. The reader need not understand this code in detail.

C. Instantiating a Generic Monitor Pattern

Section IV and V describe the codesign of the `unsafe` processor and `stack_integrity` monitor. These are devices with the following types:

```
unsafe :: Device (Port W8, Port W8)
                 (Port W8, Port W8, Port W8, Port W8)
stack_integrity :: Device (Port W8,Port Bit,Bit) (Bit,Bit)
```

These `Port` types characterize the types of the inputs and outputs of `unsafe` and `stack_integrity`, but their precise meaning is not important at this point.

What is significant is that we can now create the design of the the `safer` processor by function application:

```
safer = monitor unsafe stack_integrity combine sample
  where
    combine = ...
    sample  = ...
```

The `safer` processor is a `stack_integrity`-monitored form of `unsafe` and is portrayed in Fig. 6 (r). N.b., the data input and output ports for `safer` now are 9 bits wide, the additional bit accommodating the integrity bit. The `safer` processor also includes a new `reset` input and `alarm` output. It is also executable Haskell code from which synthesizable VHDL may be produced via the ReWire compiler.

D. Testing `unsafe` and `safer`

Note on Notation: ReWire has built-in word types (e.g., `W4`, `W8`, and `W9` for 4, 8, and 9 bit words, resp.). It also has a built-in `Bit` type and we also define a `Port` type constructor as well:

```
data Bit    = C | S   -- clear and set, resp.
data Port a = Val a | DC | Cmp
```

A value of type `Port W8`, for example, will have the form: `Val w` for some `w` :: `W8`, `DC`, or `Cmp`. Form `Val w` signifies that `w` is available at the port, `DC` stands for “don’t care”, and `Cmp` stands for complete signifying a completed write operation.

The RSP18 codebase for this paper includes two testing functions for each processor in Haskell:

```
testunsafe :: [UNSAFE] -> Int -> [Port W8]
testsafer  :: [UNSAFE] -> Int -> [(Port W8, Bit)]
```

Each test function creates code and data memory devices, places them in parallel with the processor, and connects them appropriately. A call, `testunsafe p c`, assembles the UNSAFE code `p`, loads it into the code memory, and executes `unsafe` for `c` cycles. What is returned is a `c`-length list recording all of the outputs to `unsafe`’s output port (this port can be written to by the `Output` instruction; see the RSP18 codebase for details). `testsafer` is defined similarly, although its outputs also include an alarm bit, which, if set, indicates that the monitor in `safer` has detected a stack integrity fault.

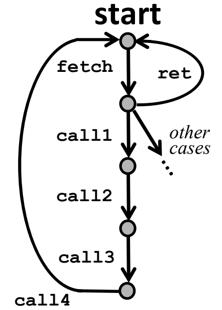
Within `safer`, the monitor should not interfere with the normal operation of the `unsafe` processor and we can test this proposition directly on the respective processor designs in using the Glasgow Haskell Compiler’s interpreter, `GHCi`. Below, we run an UNSAFE program, `fibcode`, which calculates and `Output`s each item of the fibonacci sequence, for 1000 cycles; we filter out the valid outputs using Haskell’s `filter` function:

```
async_read a = do
                i <- signal (DC,Val a,DC,DC)
                wait_read a i
  where
    wait_read a (datain,_) =
      case datain of
         Val w -> return w
         _     -> do
                    i <- signal (DC,Val a,DC,DC)
                    wait_read a i

async_write a w = do
                   i <- signal (DC,Val a,Val w,DC)
                   wait_write a w i
  where
    wait_write a w (datain,_) =
      case datain of
         Cmp -> return ()
         _   -> do
                  i <- signal (DC,Val a,Val w,DC)
                  wait_write a w i

    popM = do                   pushM w = do
      sp <- getSP                 sp <- getSP
      putSP (sp-1)                let sp' = sp+1
      async_read sp               putSP sp'
                                  async_write sp' w
```

Fig. 7: Memory Model: Read and Write Operations

```
unsafe = do
  pc <- getPC
  iw <- async_fetch pc   -- *fetch*
  let instr = decode iw
  exec instr
  unsafe

exec Call = do
  pc  <- getPC
  pc' <- popM              -- *call1*
  arg <- popM              -- *call2*
  putPC pc'
  pushM (pc + 1)           -- *call3*
  pushM arg                -- *call4*
exec Ret = do
  pc <- getPC
  ra <- popM               -- *ret*
  putPC ra
     ⟨. . . other cases elided . . .⟩
```

(a)



(b)

Fig. 8: (a) Fetch-Decode-Execute loop for UNSAFE and (b) Memory Access Patterns for `Call` and `Ret` Instructions.

```
GHCi> filter isVal (testunsafe fibcode 1000)
   [Val 0b00000000,Val 0b00000001,Val 0b00000001,
    Val 0b00000010,Val 0b00000011,Val 0b00000101,
    Val 0b00001000]
GHCi> filter (isVal . fst) (testsafe fibcode 1000)
   [(Val 0b00000000,0),(Val 0b00000001,0),
    (Val 0b00000001,0),(Val 0b00000010,0),
    (Val 0b00000011,0),(Val 0b00000101,0),
    (Val 0b00001000,0)]
```

Here, `GHCi` is the interpreter prompt. Notice that both processors return the same sequence of numbers, although `safer`'s outputs also include a 0 alarm signal, meaning that no stack integrity faults occurred.

This code snippet `push 0b1100; ret` (call it `badcode`) should set off the alarm in `safe`:

```
GHCi> testsafe badcode 1000
   [(DC,0),(DC,0),(DC,0),(DC,0),(DC,0),(DC,0),
    (DC,0),(DC,0),(DC,0),(DC,0),(DC,0),(DC,0),
    (DC,0),(DC,1),...]
```

The first component of each of the outputs is `DC` ("don't care") as `badcode` never executes an `Output` instruction, but the second is 0 for the first thirteen cycles, but on the fourteenth cycle, the alarm is set to 1. This is the cycle the middle of executing `ret` when the illegitimate target label is returned from the data memory.

IV. The UNSAFE Processor

This section presents the design of the UNSAFE processor, which implements the UNSAFE instruction set described above in Section II-C. The UNSAFE processor is a ReWire device, `unsafe`, declared in Fig. 6 (l), and it is portrayed graphically in Fig. 6 (r).

Notes on Notation: Haskell/ReWire uses "do notation" to chain together `Device` operations (Fig. 5a) and we will see examples of what constitutes an "operation" below. Given n operations, we can chain them 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`.

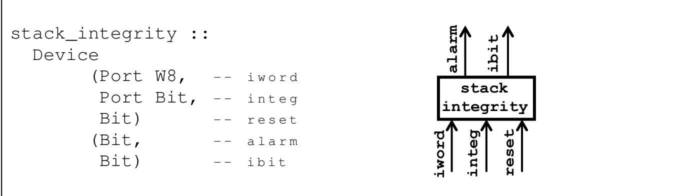
The `signal` operation in ReWire is important, as it is used by a `Device` to to receive input and produce output each clock cycle. For example, the code snippet in Fig. 5b sets the output port to `o` and waits to receive the new input `i`. Intuitively speaking, the <− above indicates the dividing line between clock cycles. The `signalo` 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`. There are other operations for manipulating registers (e.g., `getPC` and `putPC` for reading and writing the program counter and the like) as well.

A. The UNSAFE Processor

The UNSAFE processor (Fig. 6) takes inputs, (`datain`, `instr`) of type (`Port W8`, `Port W8`), in which ports `datain` and `instr` are intended to be connected to the data and instruction memories, resp. The processor produces outputs, (`fetch`, `read`, `write`, `extern`), of type (`Port W8`, `Port W8`, `Port W8`, `Port W8`). The output port `fetch` is connects to instruction memory and may signal the address of the next instruction to be fetched. The `read` and `write` output ports connect to data memory. The port `extern` is for external output.

B. Defining Memory Read, Fetch, and Write

Fig. 7 contains the ReWire code for the data memory operations. The operation (`async_read a`) signals a request to read address `a` in data memory (i.e., (`DC`, `Val a`, `DC`, `DC`)) and then waits for `datain` to return a `Val w`. During the wait loop, it continues to signal the read request. Once the valid word `w` has been received, it is returned. The (`async_write a w`) operation sets the output ports to signify a write of word `w` to address `a` (i.e., (`DC`, `Val a`, `Val w`, `DC`)) and, then, it enters a wait loop in



```
stack_integrity ::
  Device
       (Port W8,   -- iword
        Port Bit,  -- integ
        Bit)       -- reset
       (Bit,       -- alarm
        Bit)       -- ibit
```


Fig. 9: Stack Integrity Monitor.

which it continues to signal the write request until a complete (i.e., `Cmp`) is returned on `datain`. The `async_fetch` operation behaves almost identically to the `async_read` operation; its definition is included in the RSP18 codebase. Fig. 7 also contains some ReWire "microcode" for `unsafe` for popping and pushing to data memory. N.b., `pushM` is not the `Push` instruction.

Fig. 8 contains the definition for the `unsafe`, which boils down to a fetch-decode-execute loop. Fig. 8a also shows the data memory access patterns for the `Call` and `Ret` instructions. The state machine in Fig. 8b records these access patterns and is used directly in the codesign of `stack_monitor` below.

V. Codesigning The Integrity Monitor

This section presents the codesign of the stack integrity monitor for the `unsafe` processor, described in the previous section. The monitor is represented in Fig. 9 by its type (left) and as a diagram (right). The monitor inputs have the type (`Port W8`, `Port Bit`, `Bit`), and, for a typical input (`iword`, `integ`, `reset`),

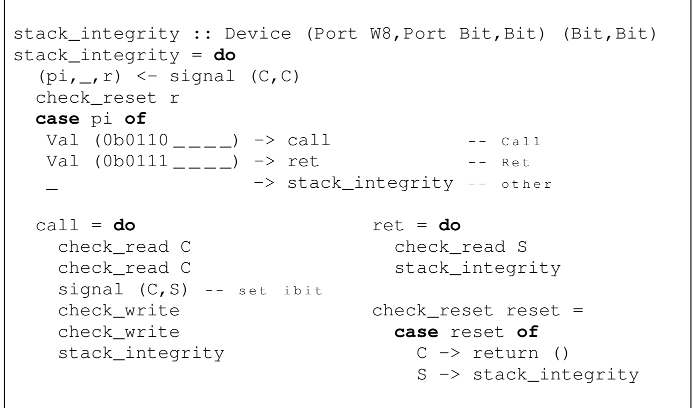
- `iword` is the instruction port to be to be connected to instruction memory. By inspecting `iword`, or its top 4 bits, the monitor may ascertain whether a relevant instruction (i.e., `Call` or `Ret`) is going to be executed;
- `integ` is the integrity bit port. If this is (`Val b`), then `b` is the integrity bit of a word just received from data memory;
- `reset` is the external reset bit.

The monitor outputs have the type (`Bit`, `Bit`), and, for a typical output (`alarm`, `ibit`),

- `alarm` is the alarm bit, which, when set, signifies that a stack integrity fault of the kind described above in Section II has occurred;
- `ibit` is the shadow integrity bit, which should be set only during the execution of a `Call` instruction and, within that timespan, only during the memory write associated with the pushing of the return label.

Fig. 10 contains the ReWire code for the integrity monitor. Initially, the monitor signals (`C`, `C`). During its execution, it periodically checks the reset bit with the routine `check_reset`, which will restart the monitor if the reset bit is set. If it receives a valid instruction word which decodes to `Call` or `Ret` (i.e., one of the top two branches in the `case` expression), it proceeds to `call` or `ret`, resp.

The memory access pattern for a `Call` instruction is portrayed in Fig. 8b; it consists of two data memory reads (i.e.,



```
stack_integrity :: Device (Port W8,Port Bit,Bit) (Bit,Bit)
stack_integrity = do
  (pi,_,r) <- signal (C,C)
  check_reset r
  case pi of
   Val (0b0110____) -> call            -- Call
   Val (0b0111____) -> ret             -- Ret
   _                -> stack_integrity -- other

  call = do
    check_read C
    check_read C
    signal (C,S) -- set ibit
    check_write
    check_write
    stack_integrity

  ret = do
    check_read S
    stack_integrity

  check_reset reset =
    case reset of
      C -> return ()
      S -> stack_integrity
```

Fig. 10: The Integrity Monitor

calls to `async_read` from two successive `popMs`) followed by two data memory writes (i.e., calls to `async_write` from two successive `pushMs`). The code for the `call` tracker in Fig. 10 mirrors this pattern. Note that the `ibit` is set after the second `popM` (i.e., (`C`, `S`) is signaled) as the return address (i.e., (`pc` + 1) in Fig. 8a) is about to be pushed. The `call` tracker returns to `stack_integrity` after the `Call` instruction has been monitored. The routines for tracking individual memory accesses, `check_read` and `check_write`, are included for the sake of completeness in Fig. 11.

The `Ret` instruction makes a single memory read, corresponding to the `popM` in its definition in Fig. 8a. The address which is returned from data memory will have nine bits, eight for the return address (`ra` in Fig. 8a) and one for the integrity bit. If the integrity bit received during the execution of a `Ret` is 0, then `integrity_alarm` (Fig. 11) will be entered; that system can only leace that state if an external `reset` is set.

VI. Related Work

The construction of a number of processors in ReWire with reasonable performance characteristics (i.e., circuit size and clock speed) has been presented in previous publications [7], including the Xilinx PicoBlaze [8] and a representative subset of the DLX instruction set [6]. Previous work has also described the design and implementation of ReWire [7], its use as a target for embedded DSLs [9], its support for equational reasoning about reconfigurable hardware [7], and the expressiveness of its programming model. ReWire is intended as a tool for producing high assurance hardware and the current work enables the potential for formally verifying the security and integrity properties of monitored systems.

There are a number of systems for configuring protection policies for embedded hardware. μShield [10] is a mitigation system for memory corruption attacks on embedded COTS hardware. Huffmire et al. [11] present a domain-specific language for memory access control policies, based in regular expressions, and a compilation flow implementing it. Their DSL thus encapsulates a class of memory controllers with configurable security policies and, given previous work on

```
check_write = do
  (pw,pi,r) <- signal (C,C)
  check_reset r
  wait4complete (pw,pi,r)
  where
    wait4complete (_,pi,r)
     = do
        check_reset r
        case pi of
          Cmp -> return ()
          _    -> do
            i <- signal
                   (C,C)
            wait4complete i
```

```
check_read b = do
    (pw,pi,r) <- signal (C,C)
    check_reset r
    wait4valid (pw,pi,r)
  where
    wait4valid (_,pi,r) = do
      check_reset r
      case (b,pi) of
        -- return if Valid
        (C,Val _) -> return ()
        -- return if integ.
              bit set
        (S,Val S) -> return ()
        (S,Val C) ->
              integrity_alarm
        _            -> do
            i <- signal (C,C)
            wait4valid i
    integrity_alarm = do
      (_,_,r) <- signal (S,S)
      check_reset r
      integrity_alarm
```

Fig. 11: Memory Access Trackers

regular expression compilation in ReWire [9], ReWire might form an implementation path for their DSL approach.

High-level synthesis (HLS) from functional languages [12], [13] is a commonly proposed remedy for this "programmability" problem [1] in reconfigurable technology. To the best knowledge of the authors, the current approach is the first application of HLS to the design and implementation of hardware runtime monitors. The construction of hardware-based monitors for security and integrity in embedded systems is currently an active area of research.

VII. Summary, Conclusions, and Future Work

We took the formal specification of an instruction set "off the shelf," converted it into a ReWire processor (i.e.,`unsafe`) while simultaneously using its memory access patterns to develop a run-time monitor enforcing a stack integrity mechanism (i.e., `stack_integrity`). Using the monitor design pattern from Fig. 4b, the `stack_integrity` monitor can be composed with the `unsafe` processor to form the `safer` processor (the complete definition is available in the RSP18 codebase). Developing and testing both processors was straightforward as both may be run using Haskell. Both the `unsafe` and `safer` processors may be synthesized with the ReWire compiler; follow-on research will investigate the performance impact of monitoring constructed according to our methodology.

All that being said, the work described herein is admittedly preliminary. We have developed ReWire models of the RISCV RV32I and Xilinx MicroBlaze instruction sets as a means of evaluating our methodology at scale. There are two other memory protection mechanisms from SAFE [2]—information flow control and user/supervisor instruction modes—that, as of this writing, we have not yet encoded as well as others from the literature (e.g., CFI [14], hardware-assisted taint-tracking [15], etc.) that we are beginning to experiment with. It will be interesting to see how well they fit into the monitor pattern expressed in Fig. 4 and what, if any, alterations or extensions they will require.

Another question of interest is to what extent these ReWire hardware monitors can be composed? There is a sense in which $\texttt{safer} = \texttt{unsafe} + \texttt{stack_integrity}$ and, if the reader will excuse the further abuse of notation, a natural question is how easily can we compose further monitors in this additive fashion, as in, for example, $\texttt{safer}' = \texttt{unsafe} + \texttt{m}_1 + \ldots + \texttt{m}_n$ for arbitrary monitors $\texttt{m}_i$? Fig. 8a portrayed the fetch-decode-execute loop of `unsafe` side-by-side with its memory access patterns in Fig. 8b. The description of access patterns has the flavor of a finite state automaton or, equivalently, a regular expression. A natural next question, therefore, asks if one can usefully generalize the work presented here as a domain-specific language for hardware monitoring in which the checking actions (e.g., `check_read` and `check_write` used in Fig. 10) as some sort of primitives? Previous work on regular expression compilation [9] would seem to provide a useful starting point.

References

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

[2] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," *Journal of Computer Security (JCS); Special Issue on Verified Information Flow Security*, vol. 24, no. 6, pp. 689–734, Dec. 2016.

[3] 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 Conference on Computer and Communications Security*, ser. CCS '07, 2007, pp. 552–561.

[4] V. Nagarajan and R. Gupta, "Architectural support for shadow memory in multiprocessors," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS VEE*, 2009, pp. 1–10.

[5] K. Vorobyov, J. Signoles, and N. Kosmatov, "Shadow state encoding for efficient monitoring of block-level properties," in *Proceedings of the 2017 ACM SIGPLAN ISMM*, 2017, pp. 47–58.

[6] I. Graves, "Device-level composition in ReWire," Ph.D. dissertation, University of Missouri, 2015.

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

[8] ——, "Semantics-directed machine architecture in ReWire," in *ICFPT*, December 2013, pp. 446–449.

[9] 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.

[10] A. Abbasi, J. Wetzels, W. Bokslag, E. Zambon, and S. Etalle, "μShield: Configurable code-reuse attacks mitigation for embedded systems," in *NSS*, 2017, pp. 694–709.

[11] T. Huffmire, T. Sherwood, R. Kastner, and T. Levin, "Enforcing memory policy specifications in reconfigurable hardware," *Computers & Security*, vol. 27, no. 5-6, pp. 197–215, 2008.

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

[13] 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.

[14] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced control-flow integrity," in *Proceedings of CODASPY '16*, 2016, pp. 38–49.

[15] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, 2007, pp. 284–292.