Theoretical Computer Science 851 (2021) 39–61

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

# Safe functional systems through integrity types and verified assembly

Michael Christensen [a,*], Joseph McMahan [b,1], Lawton Nichols [a], Jared Roesch [b,1], Timothy Sherwood [a], Ben Hardekopf [a]

[a] University of California, Santa Barbara, United States of America
[b] University of Washington, Seattle, United States of America

ARTICLE INFO

*Article history:*
Received 31 October 2019
Received in revised form 26 May 2020
Accepted 18 September 2020
Available online 23 September 2020
Communicated by C.S. Calude

*Keywords:*
Formal methods
Computer architecture
Functional programming
Binary analysis

ABSTRACT

Building a trustworthy life-critical embedded system requires deep reasoning about the potential effects that sequences of machine instructions can have on full system operation. Rather than trying to analyze complete binaries and the countless ways their instructions can interact with one another — memory, side effects, control registers, implicit state, etc. — we explore a new approach. We propose an architecture controlled by a thin computational layer designed to tightly correspond with the lambda calculus, drawing on principles of functional programming to bring the assembly much closer to myriad reasoning frameworks, such as the Coq proof assistant. This approach allows assembly-level verified versions of critical code to operate safely in tandem with arbitrary code, including imperative and unverified system components, without the need for large supporting trusted computing bases. We demonstrate that this computational layer can be built in such a way as to simultaneously provide full programmability and compact, precise, and complete semantics, while still using hardware resources comparable to normal embedded systems. To demonstrate the practicality of this approach, our FPGA-implemented prototype runs an embedded medical application which monitors and treats life-threatening arrhythmias. Though the system integrates untrusted and imperative components, our architecture allows for the formal verification of multiple properties of the end-to-end system. We present a proof of correctness of the assembly-level implementation of the core algorithm in Coq, the integrity of trusted data via a non-interference proof, and a guarantee that our prototype meets critical timing requirements.

© 2020 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

## 1. Introduction

Embedded devices are ubiquitous, with many now playing roles that support human health, well-being, and safety. The critical nature of these systems — automotive, medical, cryptographic, avionic — is at odds with the increasing complexity of embedded software overall: even simple devices can easily include an HTTP server for monitoring purposes. Traditional

* Corresponding author.
*E-mail addresses:* mchristensen@cs.ucsb.edu (M. Christensen), jmcmahan@cs.washington.edu (J. McMahan), lawtonnichols@cs.ucsb.edu (L. Nichols), jroesch@cs.washington.edu (J. Roesch), sherwood@cs.ucsb.edu (T. Sherwood), benh@cs.ucsb.edu (B. Hardekopf).

[1] Contributions were made while a graduate student at UC Santa Barbara.

https://doi.org/10.1016/j.tcs.2020.09.039
0304-3975/© 2020 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

processor interfaces are inherently global and stateful, making the task of isolating and verifying critical application subsets a significant challenge. Architectural extensions have been proposed that enhance the power, performance, and functionality of systems, but in contrast, we cover the first architecture designed with formal program analysis as a core motivating principle.

High-level, functional languages offer a remarkable ability to reason about the behavior of programs, but are often unsuited to low-level embedded systems, where reasoning must be done at the assembly level to give a full picture of the code that will actually execute. At a high level, in a language designed for verification, reasoning typically requires relying on a language run-time that can be prohibitive for resource-constrained or real-time embedded systems, and/or require the assumption that thousands of lines of untrusted code in the language stack are correct.

Another approach is to directly model the processor interface by giving formal semantics to the ISA. However, reasoning about binary behavior on traditional architectures is difficult and often left incomplete. Unless *all* program components and architectural behaviors are included, any piece outside the expected model could mutate a piece of machine state and violate the assumptions of the verification effort. Even assuming that never happens, using a verified compiler, assuming other modules are correct, using only a subset of the ISA and assuming the rest is unused, program-specific reasoning is still difficult — i.e., reasoning about C still means reasoning about pointers, memory mutation, and countless imperative, effectful behaviors.

We propose a system where the critical code can execute at the assembly level in a way that is very similar to the underlying computational model upon which proof and reasoning systems are already built. Under such a mode of computation, properties such as isolation, composition, and correctness can be reasoned about incrementally, rather than monolithically. However, instead of requiring a complete reprogramming of all software in a system, we instead examine a novel system architecture consisting of two cooperating layers: one built around a traditional imperative ISA, which can execute arbitrary, untrusted code, and one built around a novel, complete, purely functional ISA designed specifically to enable reasoning about behavior at the binary level. Application behaviors that are mission critical can be hoisted piecemeal from the imperative to the functional world as needed.

Our proposed system, the Zarf Architecture for Recursive Functions [1], observes the following properties:

1. The functional ISA, "Zarf," is devoid of all global or mutable state, and provides a compact, complete, and mathematical semantics for the behavior of instructions;
2. The imperative ISA is strictly separated from the functional ISA, connected only via a communication channel through which the system components can pass values;
3. The subset of the application which operates on Zarf can be verified and reasoned about without regard to the operation of the imperative components, meaning that *only* the critical components need to be ported and modeled;
4. Reasoning on the functional ISA is provably composable — i.e., two separate pieces can be statically shown to never interfere with each other.

To demonstrate the usefulness of this platform, we develop, model, and test a sample application which implements an Implantable Cardio-Defibrillator (ICD) — an embedded medical device which is implanted in a patient's chest cavity, monitors the heart, and administers shocks under certain conditions to prevent or counter cardiac arrest. Though ICDs provide life-saving treatment for patients with serious arrhythmia, these devices, along with other embedded medical devices, have seen thousands of recalls due to dangerous software bugs [2,3]. By leveraging this two-layer approach, we are able to formally verify the correctness of a low-level implementation of the core functions in Coq and directly extract executable assembly code without needing software runtimes. The ISA semantics allow us to construct an integrity type system and formally prove that the rest of the code never corrupts the inputs or outputs of the critical functions. Furthermore, the functional abstraction built in to the binary code allows us to bound worst-case execution time, even in the face of garbage collection. Taken altogether, we have an embedded medical application whose core components have been proven correct, where non-interference is guaranteed, where real-time deadlines are assured to be met, and where C code can execute arbitrary auxiliary functions in parallel for monitoring. The high-level system architecture is shown in Fig. 1.

Given the significant amount of related efforts in verification and ISA design, we begin by summarizing how our work differs from previous efforts in the fields of verification and architecture (Section 2). We then describe the Zarf platform in more detail and describe a hardware implementation, which runs the application on an FPGA (Section 3). Details of our embedded ICD software application and the ways it can leverage the properties of the Zarf platform are described next (Section 4), followed by a precise definition of Zarf's semantics (Section 5). We then discuss the verification of multiple properties of the critical sub-components of the ICD, covering correctness, timing, and non-interference (Section 6). Finally, we evaluate this system architecture and approach, presenting hardware resource requirements of the novel ISA, and examine the performance loss of the verified components when compared to an unverified C alternative (Section 7), and conclude (Section 8).

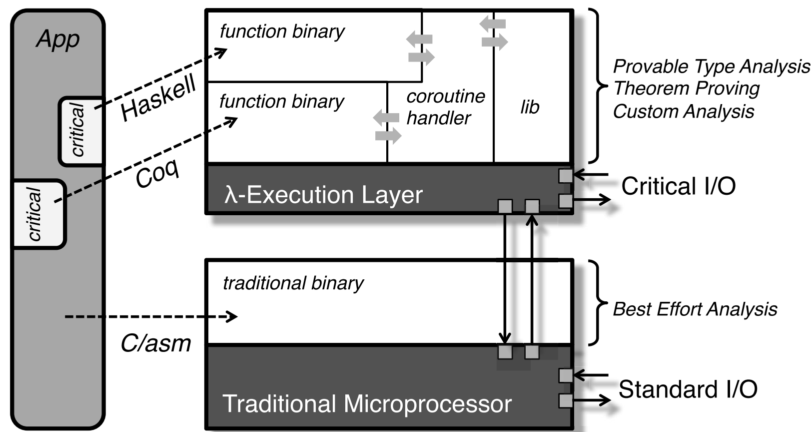


**Fig. 1.** High-level Zarf system architecture: by dividing the system into two hardware realms — one that provides a precise, mathematical semantics for reasoning about program behavior, and the other a standard imperative core for legacy software — we can formally verify and otherwise reason about critical subsets of applications without needing to model and verify the entire program.

## 2. Related work

### 2.1. Verification

Our dual ISA approach, where one is untrusted and the other trusted, draws in part on Rushby's work on security kernels [4]. He separates machine components into virtual "regimes" and proves isolation. Having done so, Rushby can then show that security is maintained when introducing clearly defined and limited channels of communication whose information flow can be tracked. Our imperative and functional ISAs behave as separate components, communicating only through a specified, dedicated channel, thus eliminating any insecure information flow — via memory contamination, for example.

Our security type system draws from the work done on the Secure Lambda (SLam) calculus by Heintze and Riecke [5] and its further development by Abadi et al. in their Core Calculus of Dependency [6]. It also draws inspiration from Volpano [7] et al., who created a type system for secure information flow for an imperative block-structured language. By showing that their type system is sound, they show the absence of flow from high-security data to lower-security output, or similarly, that low-security data does not affect the integrity of higher-security data. Other seminal work on secure information flow via the formulation of a type system includes Denning [8], Goguen [9], Pottier's information flow for ML [10], and Sabelfeld and Myers's survey on language-based information flow security [11].

Productive, expressive high-level languages that are also purely functional are excellent source platforms for Zarf. Even languages like Haskell, though, can have occasional weaknesses that can lead to runtime type-errors. Subsets such as Safe Haskell [12] shore up these loopholes, and provide extensions for sandboxing arbitrary untrusted code. Zarf provides isolation guarantees at the ISA level and does not require runtimes, but relies on languages like Safe Haskell for source code development.

Previous work on ISA-level verification has often involved either simplified or incomplete models of the architecture. These can be in the form of new "idealized" assembly-like languages: Yu et al. [13] use Coq to apply Hoare-style reasoning for assembly programs written in a simple RISC-like language. They also provide a certified memory management library for the machine they describe. Chlipala presents Bedrock, a framework that facilitates the implementation and verification of low-level programs [14], but limits available memory structures.

Kami [15] is a platform for specifying, implementing, and verifying hardware designs in Coq. By making the language of design the same as the language of verification, the gap that traditionally exists between high-level specification and actual hardware implementation is minimized. A key distinction between Kami and our work is that Zarf focuses on building a processor specifically to make software verification easier, while Kami focuses on the task of specifying and building verifiable hardware. The two are very complementary — you can build a traditional imperative machine using Kami (e.g. their pipelined RISC-V processor implementation), and you can build a Zarf machine using traditional hardware design.

Verification has also been done for subsets of existing machines. For example, a "substantial" subset of the Motorola MC68020 interface is modeled and used to mechanically prove the correctness of quicksort, GCD, and binary search [16]; other examples include a formalization of the SPARC instruction set, including some of the more complex properties, such as branch delay slots [17]; and subsets of x86 [18]. One of the biggest efforts to date has been a formal model of the ARMv7 ISA using a monadic specification in HOL4 [19]. Moore developed Piton, a high level assembly language and a verified compiler for the FM8502 microprocessor [20], which complemented the verification work done on the FM8502 implementation [21]. These are large efforts because of the difficulty in reasoning about imperative systems. At higher levels of abstraction, entire journal issues have been devoted to works on Java bytecode verification [22].

In addition to proofs on machine code for existing machines, it is also possible to define new assembly abstractions that carry useful information. Typed assembly as an intermediate representation was previously identified as a method for Proof-Carrying Code [23], where machine-checked proofs guarantee properties of a program [24]. Typed assemblies and intermediate representations have seen extensive use in the verification community [25,14,26,27] and have been extended with dependent types [28], allowing for more expressive programs and proofs at the assembly level.

Verified compilers are a popular topic in the verification community [29–32], the most well-known example being CompCert [33], a verified C compiler. Verified compilers are usually equipped with a proof of semantics preservation, demonstrating that for every output program, the semantics match those of the corresponding input program. A verified compiler does not provide tools for, nor simplify the process of doing, program-specific reasoning. One needs a secondary tool-chain for reasoning about source programs, such as the Verified Software Toolchain (VST) [34] for CompCert. These frameworks often have a great cost, mandating the use of sophisticated program logics, such as higher-order separation logic in VST, in order to fully reason about possible program behaviors.

Further, in many systems, it's possible that not all source code is available; without being able to reason about binary programs, guarantees made on a piece of the source program (and preserved by the verified compiler) may be violated by other components. Extensions to support combining the output of verified compilers, such as separate compilation and linking, are still an active research area [35,36]. As work on verified compilers requires a semantic model of the ISA, it is complemented by our work, which gives complete and formal semantics for an ISA.

Previous work at the intersection of verification and biological systems has attempted to improve device reliability through modeling efforts. This includes work that formulates real-time automata models of the heart for device testing [37], formal models of pacing systems in Z notation [38], quantitative and automated checking of the interaction of heart-pacemaker automata to verify pacemaker properties [39], and semi-formal verification by combining platform-dependent and independent model checking to exhaustively check the state space of an embedded system [40]. Our work is complemented by verification works such as these that refine device specification by taking into account device-environment interactions.

### 2.2. Architecture

The SECD Machine [41] is an abstract machine for evaluating arithmetic expressions based in the lambda calculus, designed in 1963 as a target for functional language compilers. It describes the concept of "state" (consisting of a Stack, Environment, Control, and Dump) and transitions between states during said evaluation. Interpreters for SECD run on standard, imperative hardware. Hardware implementations of the SECD Machine have been produced [42], which explore the implementation of SECD at the RTL and transistor level, but present the same high-level interface. The SECD hardware provides an abstract-machine semantics, indicating how the machine state changes with each instruction. Our verification layer makes machine components fully transparent, presenting a higher-level small-step operational semantics, where instructions affect an abstract environment, and a big-step semantics, which immediately reduces each operation to a value. These latter two versions of the semantics are more compact, precise, and useful for typical program-level reasoning.

The SKI Reduction Machine [43] was a hardware platform whose machine code was specially designed to do reductions on simple combinators, this being the basis of computation. Like our verification layer, it was garbage-collected and its language was purely applicative. The goal was to create a machine with a fast, simple, and complete ISA. The choice to use the "simpler" SKI model means that machine instructions are a step removed from the typically function-based, mathematical methods of reasoning about programs. Our functional ISA, while also simple and complete, chooses somewhat more robust instructions based on function application; though the implementation is more complicated, modern hardware resources can easily handle the resulting state machine, giving a simple ISA that is sufficiently high-level for program reasoning.

The most famous work on hardware support for functional programming was on Lisp Machines [44–46]. Lisp machines provided a specialized instruction set and data format to efficiently implement the most common list operations used in functional programming. For example, Knight [46] describes a machine with instructions for Lisp primitives such as CAR and CADR, and also for complex operations like CALL and MOVE. While these machines partially inspired this work, Lisp Machines are not directly applicable to the problem at hand. Side-effects on global state at the ISA level are critical to the operation of these machines, and while fast function calls are supported, the stepwise register-memory-update model common to more traditional ISAs is still a foundation of these Lisp Machine ISAs. In fact, several commercial Lisp Machine efforts attempted to capitalize on this fact by building Lisp Machines as a thin translation layer on top of other processors.

Flicker also dealt with architectural support for a smaller TCB in the presence of untrusted, imperative code, but did so with architectural extensions that could create small, independent, trusted bubbles within untrusted code [47]. Our architecture is almost inverted, with a trusted region providing the main control, calling out to an untrusted core as needed. Previous works such as NoHype [48] dealt with raising the level of abstraction of the ISA and factoring software responsibilities into the hardware. Our verification layer shares some of these characteristics, but deals with verification instead of virtualization, as well as being a complete, self-contained, functional ISA.

Previous work has explored the security vulnerabilities present in many embedded medical devices, as well as zero-power defenses against them [49–51]. The focus of our work is analysis and correctness properties, and we do not deal with security.

## 3. Hardware architecture and ISA

Our system relies on two separate layers, running two different ISAs, connected only by a data channel. This allows one of the layers to be specialized to the execution of machine code with 1) a compact, precise, and complete semantics highly amenable to proofs, and 2) the ability to compose verified pieces safely. It is entirely possible that all code in the system be written to be purely functional and run on Zarf: the ISA for this layer is complete. However, embedded devices often contain a mix of software, including legacy code or nice-to-have features that do not affect the application's behavior, such as relaying data and diagnostic information to outside receivers. With a two-layer approach, we can run imperative code that is orthogonal to the operation of critical application components while still connecting with the vetted, functional code in a structured way. This, in turn, allows code to be formally verified piecemeal, with functions "raised" into Zarf as deemed necessary.

The following subsections describe the interface and construction of Zarf, including the reasons we take an approach much closer to the lambda calculus underlying most software proof techniques, how we capture this style of execution in an instruction set, the semantics for that instruction set, and more practical considerations such as I/O, errors, and ALU functions.

### 3.1. Design goals

Normal, imperative architectures have been difficult to model, and the task of composing verified components is still an open problem [35,36]. We identify the following features as undesirable and counterproductive to the goal of assembly-level verification:

1. Large amounts of global machine state (memory, stack, registers, etc.) directly accessible to instructions, all of which must be modeled and managed in every proof, and which inhibit modularity: state may be modified by code you haven't seen.
2. The mutable nature of machine state, which prevents abstraction and composition when reasoning about functions or sets of instructions.
3. A large number of instructions and features: a complete model must incorporate all of them (e.g., fully modeling the behavior of the ARMv7 was 6,500 lines of HOL4 [19]).
4. Arbitrary control flow, which often requires complex and approximate analyses to soundly determine possible control flows [52].
5. Unenforced function call conventions, meaning one must prove that every function respects the convention.
6. Implicit instruction semantics, such as exceptions where "jump" becomes "jump and update registers on certain conditions."

To avoid these traits, we design an interface that is small, explicit in all arguments, and completely free of state manipulation and side effects — with the exception of I/O, which is necessary for programs to be useful. Without explicit state to reference (memory and registers), standard imperative operations become impossible, and we must raise the level of abstraction. Instead of imperative instructions acting as the building blocks of a program, our basic unit is the *function*. This is a major departure from a typical imperative assembly, where the notion of a "function" is a higher-level construct consisting of a label, control flow operations, and a calling convention enforced by the compiler — but which has no definition in the machine itself. By bringing the definition of functions to the ISA level, they become not just callable "methods" that serve to separate out independent routines, but are actually strict functions in the mathematical sense: they have no side effects, never mutate state, and simply map inputs to outputs. This change allows us to attach precise and formal semantics to the ISA operations.

### 3.2. Description and semantics

Zarf's functional ISA is effectively an **a)** untyped, **b)** lambda-lifted, **c)** administrative normal form (ANF) lambda calculus. Those limitations are a result of the implementation being done in real hardware: **a)** to avoid the complexity of a hardware typechecker, the assembly is untyped[2]; **b)** because every function must live somewhere in the global instruction memory, only top-level declarations of functions are allowed (lambda-lifted); **c)** because the instruction words are fixed-width with a static number of operands, nested expressions are not allowed and every sub-expression must be bound to its own variable (ANF). The abstract syntax of Zarf assembly is given in Fig. 2.

All words in the machine are 32-bits. Each binary program starts with a magic word, a word-length integer $N$ stating how many functions are contained in the program, and then a sequence of $N$ functions. Each function starts with an informational word that lets the machine know the "fingerprint" of the function (including the number of arguments expected

[2] The original ISA definition as presented in the paper was untyped; in later work, we extended the ISA to include type annotations and a hardware typechecker.

$$x \in \mathit{Variable} \quad n \in \mathbb{Z} \quad \mathit{fn}, \mathit{cn} \in \mathit{Name}$$

$$\begin{aligned}
p \in \mathit{Program} &::= \overrightarrow{\mathit{decl}}\ \textbf{fun}\ \text{main} = e \\
\mathit{decl} \in \mathit{Declaration} &::= \mathit{cons} \mid \mathit{func} \\
\mathit{cons} \in \mathit{Constructor} &::= \textbf{con}\ \mathit{cn}\ \vec{x} \\
\mathit{func} \in \mathit{Function} &::= \textbf{fun}\ \mathit{fn}\ \vec{x}\ = e \\
e \in \mathit{Expression} &::= \mathit{let} \mid \mathit{case} \mid \mathit{res} \\
\mathit{let} \in \mathit{Let} &::= \textbf{let}\ x = \mathit{id}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \\
\mathit{case} \in \mathit{Case} &::= \textbf{case}\ \mathit{arg}\ \textbf{of}\ \overrightarrow{\mathit{br}}\ \textbf{else}\ e \\
\mathit{res} \in \mathit{Result} &::= \textbf{result}\ \mathit{arg} \\
\mathit{br} \in \mathit{Branch} &::= \mathit{cn}\ \vec{x} \Rightarrow e \mid n \Rightarrow e \\
\mathit{id} \in \mathit{Identifier} &::= x \mid \mathit{fn} \mid \mathit{cn} \mid \mathit{op} \\
\mathit{arg} \in \mathit{Argument} &::= n \mid x \\
\mathit{op} \in \mathit{PrimOp} &::= + \mid - \mid \times \mid \div \mid = \\
&\quad \mid < \mid \leq \mid \wedge \mid \vee \mid \bar{\wedge} \mid \underline{\vee} \\
&\quad \mid \oplus \mid \ll \mid \gg \mid \texttt{sra} \mid \neg \mid \texttt{getint} \mid \texttt{putint}
\end{aligned}$$

**Fig. 2.** The Abstract Syntax for Zarf's functional ISA. A program is a set of function and constructor declarations, where functions are composed solely of `let`, `case`, and `result` expressions, and constructors are tuples with unique names. Case expressions contain branches and serve as the mechanism for both control flow and deconstruction of constructor forms. An arrow over any metavariable (e.g. $\vec{x}$) signifies a list of zero or more elements. *op* refers to a function that is implemented in hardware (such as ALU operations); though the execution of the function invokes a hardware unit instead of a piece of software, the functional interface is identical to program-defined functions.

and how many locals will be used) and a word-length integer $M$ to specify that the body of the function is $M$ words long. The remaining $M$ words of the function are then composed entirely of the individual instructions of the machine.

Each function, as it is loaded, is given a unique and sequential identifier. These function identifiers are the only globally visible state in the system and serve as both a kind of name and a kind of pointer back to the code. Other functions can refer to, test, and apply arguments to function identifiers. There are two varieties of function identifiers: those that refer to full functions that contain a body of code, and "constructors," which have no body at all. Constructors are essentially stub functions and cannot be executed. However, just like other functions, you can apply arguments to them. These special function identifiers thus can serve as a "name" for software data types, where arguments are the composed data elements. (In more formal terms, you can use our constructors to implement algebraic data types.)

The words defining the body of a function are built out of just three instructions: `let`, `case`, and `result`, which we will describe below. Unlike RISC instructions, `let` and `case` can be multiple words long (depending on the number of arguments and branches, respectively). However, unlike most CISC instructions, each piece of the variable length instruction is also word-aligned and trivial to decode.

Zarf has no programmer-visible registers or memory addresses, but instructions will still need to reference particular data elements. Instructions can refer to data by its source and index, where the source is one of a predefined set — e.g., *local* and *arg*, which serve a purpose similar to the stack on a traditional machine. The *local* and *arg* indices might be analogous to stack offsets, while the actual addresses themselves are never visible.

The primary ways of generating Zarf assembly are via extraction from Coq and writing it by hand. We also have a Haskell compiler that supporting a subset of basic Haskell constructs. In our experience, Zarf assembly code resembles a typical functional programming language like desugared Haskell or OCaml, and the resultant expressibility makes directly writing assembly relatively easy; the user doesn't need to worry about memory address calculations, maintaining register or stack state across function calls, or the myriad other things that make programming traditional ISAs tedious and error-prone. For more information on automatic Coq extraction, see our discussion of the ICD implementation in Section 6.

Fig. 5 gives the complete ISA behavior using a big-step semantics, which explains how each instruction reduces to a value. This semantics uses eager evaluation for simplicity; though the current hardware implementation uses lazy semantics, the difference is not observable in our application because I/O interactions are localized to a specific function and always evaluated immediately. The semantics use assembly keywords for readability; Fig. 3 shows how the assembly maps one-to-one with the binary encoding, and Fig. 7 shows how low-level Coq code can be directly converted to our assembly.

### 3.3. Instruction set

The `let` instruction applies a function to arguments and assigns it a *local* identifier. The first word in the `let` instruction indicates a function identifier or closure object and the number of argument words that follow. Note that unlike a function "call", `let` does not immediately change the control flow or force evaluation of arguments; rather it creates a new structure in memory (closure) tying the code (function identifier) to the data (arguments), which, when finally needed, can actually be evaluated (using lazy evaluation semantics). Additionally, the `let` instruction allows partial application, meaning that new functions (but not function identifiers) can be dynamically produced by applying a function identifier to some, but not all, of its arguments.

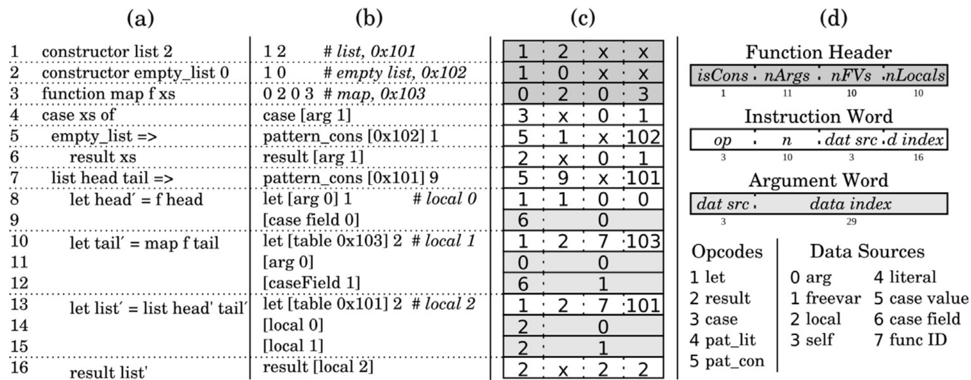


**Fig. 3.** How the high-level assembly instructions are directly compiled into a Zarf binary for Zarf execution. This example shows the `map` function, along with the `list` constructors, in (a) high-level untyped assembly, (b) machine assembly, and (c) binary. **(a)** The standard linked-list definition requires just two constructors: a `list` is either empty or a 2-element struct containing a head (a value) and a tail (a list) [lines 1-2]. The function `map` takes a function and a list as arguments [line 3]; it builds a new list, applying the function to each list element. If the argument list is empty, it returns an empty list [lines 5-6]. Otherwise, if the list matches against the head/tail constructor [line 7], it applies the function it was given to the list element [lines 8-9], calls `map` recursively on the list tail [lines 10-12], builds a new list [lines 13-15], and yields that new list [line 16]. The `else` branch is not shown. **(b)** In the lowering to machine assembly, names are replaced with local indices, addressing a value on the locals stack (e.g., list′ becomes local 2 [line 13]). Function allocations are broken up so that each argument occupies its own word. **(c)** The binary is a direct mapping from the assembly in (b), simply translating ops to opcodes and data sources to integer identifiers. 'x' indicates an unused field. **(d)** Binary encoding. Each word of the binary is either the start of a function, the start of an instruction, or an argument word in a `let` instruction. With no architecturally visible state, data is accessed with a scoped system where the program identifies source and index; all data references use the same source/index pattern.

The `case` instruction provides pattern-matching for control flow. It takes a value, then makes a set of equality comparisons, one for each "pattern" provided. The first word of the case instruction indicates a piece of data to evaluate. As we need an actual value, this is the point in execution that forces evaluation of structures created with `let` — however, it is evaluated only enough to get a value with which comparisons can be made; specifically, until it results in either an integer or a constructor object.[3] The words following the instruction encode patterns (`pattern_literal` and `pattern_cons`) against which to match the case value. If the case value exactly equals the literal value or function (i.e. constructor) identifier, execution proceeds with the next instruction; otherwise, it skips the number of words specified in the pattern argument. A matching `pattern_else` is required for every `case` which will be executed when no other matches are found (and demarcates the end of the `case` instruction encoding). Case/pattern sequences not adhering to the encoding described are malformed and invalid — e.g., you cannot skip to the middle of a branch, or have a case without an else branch.

The `result` instruction is a single word, indicating a single piece of data that the current function should yield. Every branch of every function must terminate with a result instruction (disallowing re-convergent branches means the simple pattern-skip mechanism is all that is necessary for control flow). Functions that do not produce a value do not make sense in an environment without side effects, and so are disallowed. After a `result`, control flow passes to the `case` instruction where the function result was required.

We realize that this is a departure from traditional hardware instructions and suggest reference to Fig. 3 to help ground our descriptions in a concrete example. Fig. 3 shows a small function, `map`, written in high-level assembly, machine assembly, and encoded as a binary. A more thorough description of the semantics of each of these instructions is found in Section 5.

### *3.4. Built-in functions, I/O, and errors*

ALU operations are, for the most part, already purely mathematical functions — they just map inputs to an output. The Zarf functional ISA is built around the notion of function calls, so no new mechanism or instructions are needed to use the hardware ALU. Invoking a hardware "add" is the same as invoking a program-supplied function. In our prototype, function indices less than 256 (0x100) are reserved for hardware operations; the first program-supplied function, main, is 0x100, with functions numbered up from there. During evaluation, if the machine encounters a function with an index less than 0x100, it knows to invoke the ALU instead of jumping to a space in instruction memory.

The only two functions with side-effects in the system, input and output, are also primitive functions. The input function takes one argument (a port number) and returns a single word from that port; the output function takes two arguments, a port and a value, and writes its result to the port, returning the value written. Since data dependencies are never violated in function evaluation, software can ensure I/O operations always occur in the right order even in a pure functional environ-

[3] More precisely, evaluation of that argument will always produce a result in Weak Head-Normal Form (WHNF), but never a lambda abstraction.

ment by introducing artificial data dependencies; this is the principle underlying the I/O monad [53,54], used prominently in languages like Haskell.

In a purely functional system there are no side effects, and thus no notion of an "exception". For program-defined functions, this just requires that every branch of every case return a value (that value could be a program-defined error). However, some invalid conditions resulting from a malformed program can still occur at runtime. To respect the purely functional system, these must cause a non-effectful result that is still distinguishable from valid results. Our solution is to define a "runtime error constructor" in the space of reserved functions. Every function, both hardware- and software-defined, can potentially return an instance of the error constructor. The ISA semantics are undefined in these error cases, because it's very easy to avoid — compiling from any Hindley-Milner typechecked language will guarantee the absence of runtime type errors [55,56].

## 4. System software

This section describes the software architecture across the two realms (functional and imperative) of the system, and provides an overview of the ICD and the functional coroutines.

### *4.1. Functional vs. imperative*

As our system is composed of two small and separate computational layers, the software is split across two different ISAs. For existing applications, or applications prototyped for existing platforms, the decision of which components to migrate to Zarf represents a trade-off of increased abstraction and verification capability for additional development effort and some decrease in performance. Section 7 provides some quantitative worst-case bounds for this trade-off.

Zarf runs a small microkernel based on cooperative coroutines [57,58] to handle the scheduling and communication of different software components. This allows us to more easily group and reason about code in terms of higher-level behaviors — i.e., the small surface area of each coroutine means they can be considered (and occasionally verified) in blocks, as collections of functions with a single specification and interface. The cooperative nature of the system is a design choice that allows us to avoid interrupts, which would complicate proofs of a single coroutine's behavior. Timing analysis (section 6.2) ensures each coroutine always returns control.

Zarf enables reasoning about these coroutines at the assembly and binary level. Section 6 demonstrates different properties that can be verified. The integrity type system allows a developer to statically prove that a given set of coroutines (and the microkernel itself) will execute in cooperation without one coroutine corrupting values important to another. This *composability* of verification is extremely difficult on traditional architectures, as the global and mutable nature of all state makes it quite easy for any software component to affect any other.

The imperative layer — which can be any embedded CPU, but for our purposes is a Xilinx MicroBlaze processor — runs whatever pieces of the software are not placed on Zarf. This allows for monitoring software, low-level drivers, communication protocols, and other complex, imperative code to exist and run without requiring modeling or pure-functional implementations. As this area of the system is untrusted and unverified, anything on which the critical components depend should be rewritten to run on Zarf.

In our sample application, three application coroutines are run on Zarf: one that handles the core ICD application, an I/O routine that handles the timing of reading the values from the patient's heart and outputting when shocks should occur, and a routine that sends values to the monitoring software on the imperative layer. The system operates in real-time, reading a single value from the heart, running ECG and ICD processing, and communicating the resulting value back out. In our application, the monitoring software tracks the number of times treatment occurs, and, when prompted from its communication channel, will output that number. This imperative software could be arbitrarily complex and handle more complicated monitoring and diagnosis, communication drivers to communicate with the outside world, or other features; as it is a standard imperative core, any embedded C code can be easily compiled for it with an off-the-shelf compiler.

### *4.2. ICD*

ICDs are small, battery-powered, embedded systems which are implanted in a patient's chest cavity and connect directly with the heart. For patients with arrhythmia and at risk for heart failure, an ICD is a potentially life-saving device. Currently, the primary use of ICDs is to detect dangerous arrhythmias (such as ventricular tachycardia, or VT) and administer pacing shocks (anti-tachycardia pacing, or ATP). These shocks help prevent the acceleration in heart rate leading to ventricular fibrillation, a form of cardiac arrest.

From 1990 to 2000, over 200,000 ICDs and pacemakers were recalled due to software issues [2]. Between 2001 and 2015, over 150,000 implanted medical devices were recalled by the FDA because of life-threatening software bugs [3]. However, ICDs are credited with saving thousands of lives; for patients who have survived life-threatening arrhythmia, ICDs decrease mortality rates by 20-30% over medication [59–61]. Currently, around 10,000 new patients have an ICD implanted each month [62], and around 800,000 people are living with ICDs [63].

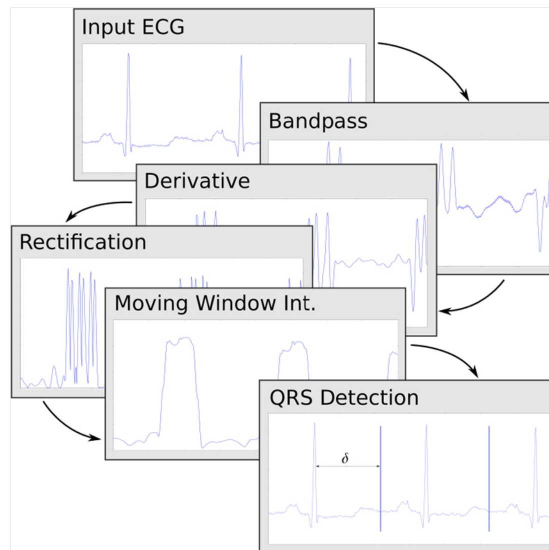


**Fig. 4.** The ECG takes input signals sampled at 200 Hz and filters them multiple times, after which the peaks are classified and the rate of heartbeat determined. These values are fed to an ATP (antitachycardia pacing) procedure, which decides if ventricular tachycardia is occurring based on current and previous heart rate, and administers pacing shocks to prevent acceleration and ventricular fibrillation (a form of cardiac arrest).

The core of our ICD is an embedded, real-time ECG algorithm that performs QRS[4] detection on raw electrocardiogram data to determine the timing between heartbeats. We work off of an established real-time QRS detection algorithm [64], which has seen wide use and been the subject of studies examining its performance and efficacy [65]. An open-source update of several versions of the algorithm [66] is available; we use the results of this open-source work as the basis of our algorithm's specification as well as the C alternative. After the ECG algorithm detects the pacing between heartbeats, the ATP function checks for signs of ventricular tachycardia and, if found, administers a series of pacing shocks. We implement the VT test and ATP treatment published in [67].

The I/O coroutine is passed the output of the previous iteration of the ICD coroutine. A hardware timer is used to ensure that I/O events occur at the correct frequency. When the correct time has elapsed (5 ms), the I/O coroutine outputs the given value and reads the next input value. It yields this value to the microkernel.

This input is then passed through to the ICD coroutine, which implements a series of filter passes to detect the spacing between QRS complexes (Fig. 4 illustrates the ECG filter passes). If 18 of the last 24 beats had periods less than 360 ms (corresponding to a heart rate greater than 167 bpm), the ICD coroutine moves into a treatment-administering state, where it outputs three sequences of eight pulses at 88% of the current heartrate, with a 20 ms decrement between sequences. This is designed to prevent continued acceleration and restore a safe rhythm.

The monitoring software, which runs on the MicroBlaze, receives the output of the ICD coroutine each cycle. A command can be given on the diagnostic input channel for the software to output the number of times treatment has occurred.

I/O events occur at a fixed frequency of 200 Hz. Timing analysis in Section 6.2 confirms that, after an input event, the entire cycle of each coroutine running and yielding, including garbage collection, is able to conclude well within the 5 ms window, meaning that the entire system is always able to meet its real-time deadline.

## 5. ISA semantics

Zarf has the core goal of providing concise, mathematical semantics for its hardware ISA. These can be found in Fig. 5, which gives the complete ISA behavior using a big-step semantics, explaining how each instruction reduces to a value. This semantics uses eager evaluation for simplicity; though the current hardware implementation uses lazy semantics, the difference is not observable in our application because I/O interactions are localized to a specific function and always evaluated immediately.

The semantics are discussed in more detail in the following subsections. Note that terms introduced in the abstract syntax (Fig. 2) are used in the semantics. Each rule (or helper function (Fig. 6)) is applicable in a different case, depending on what is under evaluation; the scenarios are all mutually exclusive, meaning that there is always exactly one rule that can (and should) be applied at every step.

[4] The "QRS complex" is made up of the rapid sequence of Q, R, and S waves corresponding to the depolarization of the left and right ventricles of the heart, forming the distinctive peak in an ECG.

$$c \in Constructor = Name \times \overrightarrow{Value} \qquad clo \in Closure = (\lambda\vec{x}.e) \times \overrightarrow{Value}$$

$$v \in Value = \mathbb{Z} \uplus Constructor \uplus Closure \qquad \rho \in Env = Variable \rightarrow Value$$

$$\frac{\vdash e \Downarrow v}{\vdash \overrightarrow{decl}\ \textbf{fun}\ \text{main} = e \Downarrow v}\ (\text{PROGRAM}) \qquad \frac{v = \rho(arg)}{\rho \vdash \textbf{result}\ arg \Downarrow v}\ (\text{RESULT})$$

$$\frac{\vec{v}_1 = \rho(\overrightarrow{arg}) \quad v_2 = \texttt{applyCn}(cn, \vec{v}_1) \quad \rho[x \mapsto v_2] \vdash e \Downarrow v_3}{\rho \vdash \textbf{let}\ x = cn\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow v_3}\ (\text{LET-CON})$$

$$\frac{\begin{array}{c} fn \notin \{\textbf{getint}, \textbf{putint}\} \quad \textbf{fun}\ fn\ \ \vec{x}_2\ = e_2 \in \overrightarrow{decl} \\ \vec{v}_1 = \rho(\overrightarrow{arg}) \quad v_2 = \texttt{applyFn}((\lambda\vec{x}_2.e_2, []), \vec{v}_1, \rho) \quad \rho[x_1 \mapsto v_2] \vdash e_1 \Downarrow v_3 \end{array}}{\rho \vdash \textbf{let}\ x_1 = fn\ \overrightarrow{arg}\ \textbf{in}\ e_1 \Downarrow v_3}\ (\text{LET-FUN})$$

$$\frac{v_1 = \rho(x_2) \quad \vec{v}_2 = \rho(\overrightarrow{arg}) \quad v_3 = \texttt{applyFn}(v_1, \vec{v}_2, \rho) \quad \rho[x_1 \mapsto v_3] \vdash e \Downarrow v_4}{\rho \vdash \textbf{let}\ x_1 = x_2\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow v_4}\ (\text{LET-VAR})$$

$$\frac{\vec{v}_1 = \rho(\overrightarrow{arg}) \quad v_2 = \texttt{applyPrim}(op, \vec{v}_1) \quad \rho[x \mapsto v_2] \vdash e \Downarrow v_3}{\rho \vdash \textbf{let}\ x = op\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow v_3}\ (\text{LET-PRIM})$$

$$\frac{(cn, \vec{v}_1) = \rho(arg) \quad (cn\ \vec{x} \Rightarrow e_1) \in \overrightarrow{br} \quad \rho[\vec{x} \mapsto \vec{v}_1] \vdash e_1 \Downarrow v_2}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_2}\ (\text{CASE-CON})$$

$$\frac{n = \rho(arg) \quad (n \Rightarrow e_1) \in \overrightarrow{br} \quad \rho \vdash e_1 \Downarrow v}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v}\ (\text{CASE-LIT})$$

$$\frac{\begin{array}{c} ((cn, \vec{v_1}) = \rho(arg) \quad (cn\ \vec{x} \Rightarrow e_1) \notin \overrightarrow{br}) \vee (n = \rho(arg) \quad (n \Rightarrow e_1) \notin \overrightarrow{br}) \\ \rho \vdash e_2 \Downarrow v_2 \end{array}}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_2}\ (\text{CASE-ELSE})$$

$$\frac{\begin{array}{c} n_2 \text{ is input from port } n_1 \\ \rho[x \mapsto n_2] \vdash e \Downarrow v \end{array}}{\rho \vdash \textbf{let}\ x = \textbf{getint}\ n_1\ \textbf{in}\ e \Downarrow v}\ (\text{GETINT}) \qquad \frac{\begin{array}{c} n_1 \text{ is a port} \\ n_2 = \rho(arg) \quad \rho[x \mapsto n_2] \vdash e \Downarrow v \end{array}}{\rho \vdash \textbf{let}\ x = \textbf{putint}\ n_1\ arg\ \textbf{in}\ e \Downarrow v}\ (\text{PUTINT})$$

**Fig. 5.** Big-Step Semantics for Zarf's functional ISA. Our semantics is a ternary relation on an environment; a let, case, or result expression; and the value to which this expression evaluates. Evaluation begins with the main function's body. $\rho[x \mapsto v]$ returns an updated copy of the environment with $x$ mapped to $v$. **getint** gets an integer from a specified port, and **putint** puts an integer onto a specified port; both are the only mechanisms for I/O in the system.

$$\texttt{applyFn}((\lambda\vec{x}_1.e, \vec{v}_1), \vec{v}_2, \rho) = \begin{cases} v & \text{if } |\vec{v}_2| = 0,\ |\vec{v}_1| = |\vec{x}_1|, \text{ and } \rho[\vec{x}_1 \mapsto \vec{v}_1] \vdash e \Downarrow v \\ (\lambda\vec{x}_1.e, \vec{v}_1) & \text{if } |\vec{v}_2| = 0 \text{ and } |\vec{v}_1| < |\vec{x}_1| \\ \texttt{applyFn}((\lambda\vec{x}_1.e, \vec{v}_1 :+ \texttt{hd}(\vec{v}_2)), \texttt{tl}(\vec{v}_2), \rho) & \text{if } |\vec{v}_2| > 0 \text{ and } |\vec{v}_1| < |\vec{x}_1| \\ \texttt{applyFn}((\lambda\vec{x}_2.e', \vec{v}_3), \vec{v}_2, \rho) & \text{if } |\vec{v}_2| > 0,\ |\vec{x}_1| = |\vec{v}_1|, \text{ and } \rho[\vec{x}_1 \mapsto \vec{v}_1] \vdash e \Downarrow (\lambda\vec{x}_2.e', \vec{v}_3) \end{cases}$$

$$\texttt{applyCn}(cn, \vec{v}) = \begin{cases} (cn, \vec{v}) & \text{if } (\textbf{con}\ cn\ \vec{x}) \in \overrightarrow{decl} \text{ and } |\vec{v}| = |\vec{x}| \\ (\lambda\vec{x}.\textbf{let}\ c = cn\ \vec{x}\ \textbf{in result}\ c, \vec{v}) & \text{if } (\textbf{con}\ cn\ \vec{x}) \in \overrightarrow{decl} \text{ and } |\vec{v}| < |\vec{x}| \end{cases}$$

$$\rho(arg) = \begin{cases} n & \text{if } arg = n \\ v & \text{if } arg = x \text{ and } (x \mapsto v) \in \rho \end{cases}$$

$$\texttt{applyPrim}(op, \vec{v}_1) = \begin{cases} v & \text{if } |\vec{v}_1| = \texttt{arity}(op) \text{ and } v = \texttt{eval}(op, \vec{v}_1) \\ (\lambda\vec{x}_1.\textbf{let}\ x_2 = op\ \vec{x}_1\ \textbf{in result}\ x_2, \vec{v}_1) & \text{if } |\vec{v}_1| < \texttt{arity}(op) \text{ and } |\vec{x}_1| = \texttt{arity}(op) \end{cases}$$

**Fig. 6.** Big-Step Semantics Helpers for Zarf's functional ISA. `applyFn` (`applyCn`) performs function (constructor) application. $\vec{x} :+ y$ appends $y$ to the end of $\vec{x}$, creating a new list. $|\vec{x}|$ means length of the list $\vec{x}$. `eval` returns the value of applying a primitive operation to arguments. Because functions are lambda-lifted, our version of closures track the list of values to be applied upon saturation, rather than an entire environment like normal closures.

### 5.1. Names and programs

A `Constructor` is a unique name and a list of zero or more values. Constructors serve as software data types, as a simple system for building up more complex data objects. The name indicates the "type" of the constructor, encoded statically as a unique integer, which the machine uses at runtime to distinguish constructors of different types.

A `Closure` is a function object, tying a function to a list of zero or more values, which are the arguments that have already been supplied. Closures allow for the dynamic construction of function objects from statically defined functions: e.g., applying the argument `1` to the static binary function `add` creates a new closure, which expects one argument, that performs the function $\lambda x.x+1$.

A `Value` is either an integer, a constructor, or a closure. The machine uses one bit at runtime to track which values are primitives and which are objects (either constructors or closures), and identifies constructor types with their name (unique type integer), but is otherwise untyped.

An `Environment` is a semantic entity mapping variables (local names) to values (integers, constructors, and closures). The semantics are high-level and do not specify how the machine should implement the behavior of the environment, but function arguments and local variables fall into the environment of each function. The set of functions (declared at the top level) are stored in a list of declarations $\overrightarrow{decl}$, which is essentially a map from the function's name to its parameter list and body.

The `PROGRAM` rule states that there should be a set of zero or more function and constructor declarations, and one function `main` with a body expression $e$. Given the declarations and main function, and application of the semantic rules, we can reduce $e$ to a value.

### 5.2. Result

The `RESULT` rule states that, given the current function environment and a `result` instruction with argument $arg$, we can reduce the current function execution to a single value $v$ using the environment to look up $arg$, if $arg$ is a variable, or simply returning it, if $arg$ is a number.

### 5.3. Let

A `let` instruction will be reduced using one of four rules: `LET-FUN`, `LET-CON`, `LET-VAR`, or `LET-PRIM`. The first is used for static program function application; i.e., applying arguments to a program-defined function (which excludes I/O and hardware functions). Similarly, the second is used for static constructor application; i.e., applying arguments to a program-defined constructor. The third is used to apply arguments to a runtime value, which will be a closure expecting additional arguments. The final is application on primitive (hardware ALU) functions. I/O functions (`getint` and `putint`) have separate rules.

`LET-FUN` is used when the instruction under evaluation is a `let` instruction applying zero or more arguments to a program-defined function $fn$. Its premises state that the function should not be `getint` or `putint`, that the function should be defined in the program declarations, that the arguments should all be reducible to a sequence of values $\overrightarrow{v_1}$ using the current function environment, that application of the `applyFn` helper rule on the body of $fn$ with arguments $\overrightarrow{v_1}$ will result in a value $v_2$, and finally, that binding a new local variable to $v_2$ will allow us to reduce the remainder of the instructions in the current function to a value.[5] The final premise of the rule ($\rho[x_1 \mapsto v_2] \vdash e_1 \Downarrow v_3$) continues the execution: $e_1$ is the remainder of the instructions, where the environment now includes a mapping from $x_1$ to the newly calculated value $v_2$. A premise of this format occurs in every rule for non-terminal instructions (everything but `PROGRAM` and `RESULT`); the semantics treat the instructions as recursive, such that each instruction "points" to the rest of the instructions.

The premises for `LET-CON` are very similar to those of `LET-FUN`, with the primary difference coming in `applyCn`. While function applications can be oversaturated, we disallow oversaturation of constructor applications for simplicity; we haven't found this overly restrictive at all in practice. Otherwise, the rules behave similarly, storing the value that results from applying arguments to a constructor into the environment $\rho$ before continuing to evaluate the next expression $e$ to a value $v_3$. In this way, closures and constructors are structurally the same; both are function identifiers with a sequence of arguments. The difference is that closures are already fully evaluated.

`LET-VAR` is used when a `let` instruction applies arguments to a dynamic (runtime) value $x_2$. The premises state that $x_2$ should be reducible via the environment to a value $v_1$, that the sequence of arguments are reducible to a sequence of values $\overrightarrow{v_2}$, that applying the arguments $\overrightarrow{v_2}$ to the object $v_1$ with the `applyFn` operation will result in a value $v_3$, and that binding a local variable to that value will allow us to reduce the remainder of the instructions to a value. The `applyFn` is

[5] As these are Big-Step semantics, rules indicate how expressions reduce directly to values, rather than giving step-by-step instructions for performing the reduction. In evaluating `LET-FUN`, for example, the rule does not instruct you to "call" into the indicated function, but rather just states that the function reduces to a value (as it must, eventually), then uses the value; as we are writing mathematical expressions, the reduction is assumed to occur immediately. One can still "execute" the semantics by stepping through, evaluating operands as necessary using the rules in places that the semantics simply reduce immediately to a value.

written to only accept closures as its first arguments; in calling it, there is an implicit premise that $v_1$ is a closure object. $x_2$ reducing to any other type of value (an integer or constructor) is a runtime error, and can occur only if the program is not well-typed.

`LET-PRIM` is similar to `LET-FUN`, but is used only when the static function is a primitive operation. These functions must be treated differently because there is no program definition for `add`, or `sub`, or `mult`; during machine execution, the hardware ALU handles them. The semantics define ALU operations the same way as the machine (as 32-bit modular mathematical operations). The premises of `LET-PRIM` contain no function declaration; in addition, they invoke `applyPrim` instead of `applyFn`. Otherwise, the application is similar: the function call reduces to a value, and binding that value to a local variable will allow us to reduce the remainder of the instructions in the function under evaluation.

### 5.4. I/O

`GETINT` is the rule for the hardware-defined input function; it is invoked when a `let` instruction uses `getint` in a program, which takes a single argument $n_1$. The premises state that reading a value from port $n_1$ should return an integer $n_2$; binding that value to a local variable, we can proceed with evaluation.

`PUTINT` is the rule for the output function (also hardware-defined); it takes two values: a port number $n_1$, and an *arg* that should be output. The premises use the environment to reduce *arg* to a value; not stated (as it is a side effect) is that the hardware sends this integer value to the indicated port. The integer sent is also used as the value to which the instruction reduces, so it is bound to a local variable, and evaluation proceeds.

### 5.5. Case

We use two rules for the evaluation of case instructions (`CASE-CON` and `CASE-LIT`, for constructor and integer scrutinees, respectively); in addition, the rule `CASE-ELSE` handles the “else” branch for each of the two case varieties.

These `CASE` rules are invoked when a `case` instruction is under evaluation; which rule is used depends on what the scrutinee reduces to: if it is a constructor, `CASE-CON` is used, while `CASE-LIT` handles integers. A `case` instruction includes a series of zero or more branches, each of which has a constructor name or integer as a guard, and one `else` branch. Exactly one branch will always execute. Since constructor “names” become unique integers, constructor and integer matches must be encoded differently to distinguish which variety each branch is meant to match.

The first premise of `CASE-CON` indicates that the scrutinee *arg* must reduce to a constructor; the second says to take the expression from the branch with the matching constructor name and use that for the remainder of the function, which will reduce to a value. `CASE-LIT` is similar, but requires the *arg* to reduce to an integer, and takes the expression from the branch that attempts to match exactly that integer.

The `CASE-ELSE` rule is invoked when no matching branch is found for the `case` instruction (indicated in the disjuncted premises); there, it indicates to take the expression from the `else` branch in the instruction and use that for the remainder of the function.

### 5.6. `apply` *helper functions*

`applyFn` is perhaps the most complicated rule, because it is where currying is handled — different actions must be taken if too few, too many, or just enough arguments are supplied to a function application. The helper rule takes a closure and a list of zero or more values, which will be arguments to the closure.

1. If zero arguments are supplied, and there are exactly enough values already saved and ready to be applied in the closure, then simply feed those values as the arguments to the function, reducing the function body to a value, and return that.
2. If zero arguments are supplied, and there are not enough saved values in the closure, return the same closure.
3. If at least one argument is supplied, and there are not enough saved values in the closure, recursively call into `applyFn`, taking the first argument from the list and appending it to the list of saved values. Either the argument list will run out before the function is saturated (resulting in case 2), or the function will eventually be saturated (resulting in case 1 or 4).
4. If at least one argument is supplied, and there are exactly enough saved values already in the closure, then we evaluate the closure (which must, if the program is well-typed, result in another closure), then recursively call `applyFn` with the new closure and the same argument list.

`applyCn` handles applications of arguments to constructors. The first rule is if exactly enough arguments are applied to the constructor application; in that case, a constructor containing those values is built and returned. The second handles the case where fewer values were supplied than expected (partially saturating a constructor); in this case, a closure is returned to capture the values already supplied, and when additional values are applied, it will invoke `applyCn` again to check if the constructor has all fields necessary to be built. We note that this appears to be dynamically creating syntax (the rule

**(a)**
```
CoFixpoint threshold_rec_hl
    (xs: Stream Z) (pBCnt tmpPeak: Z) ...
        : (Stream Z) :=
  let x := Str_nth 0 xs in
  match filter_pks_hl pBCnt tempPeak x with
    | (x', preBlankCt', tempPeak') => ...
```
⇓ **(b)**
```
CoFixpoint threshold_rec_ll
    (x pBCnt tmpPeak ... : Z) :=
  let fres := filter_pks_ll pBCnt tmpPeak x in
  match filterres with
    | mk_tuple3 preBlankCt' tmpPeak' x' => ...
```
⇓ **(c)**
```
fun threshold_rec x pBCnt TmpPeak ... =
  let fres = filter_pks pBCnt tmpPeak x in
  case fres of {
    tuple3 preBlankCt' tmpPeak' x' => ...
```

**Fig. 7.** Extraction of verified application components, summarized for a small excerpt. **(a)** The high-level Coq specification is written to operate on `Streams` (infinite lists); values are pulled from the front of the stream. **(b)** An intermediate version is written in Coq which operates on integers instead of streams, and unfolds nested operations so each function call and arithmetic operation takes one line. This intermediate version is proven equivalent in Coq to the high-level specification — meaning that repeated recursive application of (b) will always output the same sequence of values as (a). **(c)** A simple extractor just replaces the keywords in (b) to produce valid assembly code that can run on Zarf.

indicates placing a `let` instruction into the created closure), but as every constructor has a finite number of fields, these functions are all known statically.

`applyPrim` handles evaluation of primitive operations. The first case is invoked when the correct arguments have been supplied for that particular operation, and simply evaluates the operation according to the rule for 32-bit modular arithmetic, returning the answer. Similar to `applyCn`, we must account for the case where the function did not receive enough arguments; as there, we create a new closure to capture the arguments supplied, and the operation can be evaluated once all arguments are received (the second case).

The $\rho(arg) = \ldots$ helper function is for convenience, simply stating that if *arg* is an integer, return that value; otherwise, it must be a name mapped to some value in the environment, in which case that value should be returned.

## 6. Verification

We separate the verification of the embedded ICD application into three parts: verification of the correctness of the ICD coroutine, a timing analysis to show that the assembly meets timing requirements in the worst case, and a proof of non-interference between the trusted ICD coroutine and untrusted code outside of it.

### 6.1. Correctness

We first implement a high-level version of the application's critical algorithms (the ECG filters and ATP procedure) in Gallina, the specification language of the Coq theorem prover [68], using this version as our specification of functional correctness. This specification operates on streams — a datatype that represents an infinite list — by taking a stream as input and transforming it into an output stream. By sticking to a high-level, abstract specification, we can be more confident that we have specified the algorithm correctly. An ICD implementation cannot operate on streams, as all data is not immediately available; instead, it takes a single value, yields a single value, and then repeats the process.

The form of the correctness proof is by refinement: first, we create a Coq implementation of the ICD algorithm that is "lower-level" than the Coq specification. This lower-level implementation operates on machine values rather than streams, isolates function applications to let expressions, and avoids the use of "if-then-else" expressions, among other trivially-resolved differences. We then create an extractor that converts this lower-level Coq code directly into executable Zarf functional assembly code (see Fig. 7). If, for all possible input streams, we can prove that the output stream produced by the high-level Coq specification is the same sequence of values produced by the lower-level implementation, we can conclude that the program we run on Zarf is faithful to the high-level Coq specification. This proof of equivalence between the two Coq implementations is done by induction and coinduction over the program, showing that if output has matched up to point $N$, and the computation of value $N$ is equivalent, then value $N+1$ will be equivalent as well. As compared to extracting for an imperative architecture, we avoid needing to compile functional operations to an imperative ISA and do not require a large software runtime — or any software runtime at all. The translation simply replaces Coq keywords with Zarf assembly keywords, which is possible because the low-level Coq specification is in the A-normal form that Zarf requires. For example, the Coq keyword `CoFixpoint` would be textually replaced with `fun`, `match` replaced with `case`, etc.

We begin constructing our proof by first defining the relevant datatypes and expressions of the ISA as inductively defined mathematical objects:

```
Inductive data : Set :=
  | local : nat -> data
  | arg   : nat -> data
  | caseValue : data
  | caseField : nat -> data
  | literal : Z -> data
  | function  : string -> data
  | functionApplication : string -> list data -> data.

Inductive zarf_inst : Set :=
  | apply' : data -> list data -> zarf_inst
  | case   : data -> list alt -> zarf_inst
  | ret    : data -> zarf_inst
with alt : Set :=
  | literalAlt : Z -> list zarf_inst -> alt
  | constructorAlt : string -> list zarf_inst -> alt.

Inductive zarf_table : Set :=
  | function_table : string -> nat -> list zarf_inst -> zarf_table
  | constructor_table : string -> nat -> zarf_table.
```

We're then able to define a small interpreter that executes the instruction semantics (several constructors have been omitted for brevity and clarity of presentation):

```
Inductive zarf_run_function' : string -> list data -> list data -> data ->
    list zarf_inst -> data -> Prop :=
| H_apply' x res ys zs (args:list data) f locals caseVal :
    zarf_run_function' f args (locals ++ execApply x ys args locals caseVal) caseVal
        zs res -> zarf_run_function' f args locals caseVal (apply' x ys :: zs) res
...
| H_matchesLiteral xrep z zs alts d res (l:list zarf_inst) f args locals caseVal x :
    getData x args locals caseVal = Some xrep ->
    matchesCase xrep (literalAlt d l) = true ->
    zarf_run_function' f args locals xrep l res ->
    zarf_run_function' f args locals caseVal
        (case x (literalAlt d l :: z :: alts) :: zs) res
| H_ret x args locals caseVal f xrep zs : getData x args locals caseVal = Some xrep
    -> zarf_run_function' f args locals caseVal (ret x :: zs) xrep .

Inductive zarf_run' (t : zarf_table) (args : list data) : data -> Prop :=
| Run_function f arity insts res : t = function_table f arity insts ->
                zarf_run_function' f args [] (function "error") insts res ->
                zarf_run' t args res.
```

The `zarf_run_function'` interpreter is made up of several cases, each of which correspond to rules in the big-step semantics defined in Fig. 5. For example, the first case, `H_apply'`, defines what function application means. It says that if we add a thunk to our local environment that is the result of executing the apply operation (`locals + execApply x ys args locals caseVal`), and *then* execute the rest of the instructions `zs`, we have successfully executed an apply statement followed by the rest of the instructions `zs`.

We declare several axioms asserting the correctness of the ISA's built-in functions and their equivalence to built-in Coq operations. These built-in functions (like add, multiply, etc., labeled *PrimOp* in Fig. 2) are ultimately the base operations employed by all user-defined functions, and we reference them during the proof of correctness of the higher-level ECG algorithms later on. We also define a few axioms related to list and pair construction, whose correspondence to the user-made Zarf function equivalents are trivial. Here are a few assorted examples:

```
Axiom cons_rep : forall l' A (x':A) xs', l' == x' :: xs' -> exists x xs,
    l' = (functionApplication "cons" [x; xs]) /\ x == x' /\ xs == xs'.
Axiom snd_same : forall (A B : Type) x (x' : A*B), x == x' ->
    functionApplication "snd" [x] == snd x'.
Axiom add_same : forall x y x' y', x == x' -> y == y' -> x + y ==
    functionApplication "add" [x'; y'].
```

We can then begin defining and proving things about non-builtin functions, like append, reverse, index, and various matrix multiplication operations. Here's an example of defining the simplest user-defined function, `id`, which is used often in Zarf assembly for assigning a constant numeric value to a variable (because `let` expressions purely operate over function identifiers (see Fig. 2)). Proof bodies are omitted for brevity:

```
Definition zarf_id : zarf_table :=
  function_table "id" 1 [ret (arg 0)].

Axiom id_eta : forall x res, zarf_run' zarf_id [x] res ->
    functionApplication "id" [x] = res.

Theorem equivalence_id :
  forall A (x:A) x', x == x' ->
    exists res, zarf_run' zarf_id [x'] res /\ res == x.

Lemma id_same :
  forall (A : Type) x (x' : A), x == x' ->
    functionApplication "id" [x] = x.
```

We then proceed to define several ECG helper functions, like the low pass filter, at both a high-level stream-based abstraction and the lower-level implementation which operates on machine values, and verify their equivalence (and thus the correctness of the machine's implementation of the specification):

```
(* Define Zarf implementation of the low pass filter as a series of Zarf instr. *)
Definition zarf_lpf : zarf_table :=
  function_table "lpf" 0 [ apply' (function "lpf_rec") ...<arguments>...].
Definition zarf_lpf_rec : zarf_table :=
  function_table "lpf_rec" 13 [
    apply' (function "mult") [arg 1; literal 2];
    ...
    apply' (function "tuple2") [local 7; local 6];
    ret (local 8)
  ].

(* Define the high-level Coq stream-based specification of the low pass filter *)
CoFixpoint low_pass_filter_rec (xs: Stream Z) (ys: list Z) : (Stream Z) := ...

(* Declare that the result of the running the "lpf_rec" Zarf function
is the same that results from running the low-pass filter function
through our Coq-defined Zarf interpreter *)
Axiom lpf_rec_eta : forall xs' res, zarf_run' zarf_lpf_rec xs' res ->
    functionApplication "lpf_rec" xs' = res.

(* Define the non-stream-based LPF function *)
CoFixpoint low_pass_filter_rec2
  (y2 y1 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0: Z) : Tuple2 Z :=
  let y1m2 := Z.mul 2 y1 in
  let x5m2 := Z.mul 2 x5 in
  let t0 := Z.sub y1m2 y2 in
  ...

(* Prove the two implementations are equivalent *)
Theorem same_low_pass_filter_rec_and_rec2 :
  forall xs ret1 ret2
         y2 y1 x10' x9' x8' x7' x6' x5' x4' x3' x2' x1' x0',
    (forall i, i >= 0 -> i <= 10 -> Str_nth i xs =
         nth i [x10';x9';x8';x7';x6';x5';x4';x3';x2';x1';x0'] 0%Z)%nat ->
    low_pass_filter_rec xs [y2;y1] = ret1 ->
    low_pass_filter_rec2 y2 y1 x10' x9' x8' x7' x6' x5' x4' x3' x2' x1' x0' = ret2
    -> StreamTuple2Same 10 xs ret1 ret2.
...
```

After we've proven the `same_low_pass_filter_rec_and_rec2` theorem, we can convert the low-level low pass filter implementation directly into Zarf assembly. In this example, the assembly version of `low_pass_filter_rec2` would look like:

```
fun lpf_rec y2 y1 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0 =
    let y1mult2 = mult y1 2 in
    let x5mult2 = mult x5 2 in
    let t0 = sub y1mult2 y2 in
    ...
```

The preceding snippets of code have been just a sample of the entire set of proofs we wrote. The full proofs of correctness of the assembly-level critical ECG and ATP functions take under 2,500 lines of Coq. The implementations are converted line-for-line into Zarf assembly code, which is combined with assembly for the microkernel and other coroutines.

In total, the Trusted Code Base for the correctness proof includes: the hardware, the Coq proof assistant, and the small extractor that converts the low-level Coq code into Zarf functional assembly code. All other code is untrusted and may be incorrect, and the proof will still hold. The high-level ISA and clearly-defined semantics make this very small TCB possible, allowing the exclusion of language runtimes, compilers, and associated tooling that is frequently present in the TCB in verification efforts.

### 6.2. Timing

With a knowledge of how the Zarf hardware executes each instruction, we create worst-case timing bounds for each operation. In general, in a functional setting, unbounded recursion makes it impossible to statically predict execution time of routines. Though our application uses infinite recursion to loop indefinitely, the goal is to show that each iteration of the loop meets the real-time deadline; within that loop, each coroutine is executed only once, and no functions call into themselves. This allows us to compute a total worst-case execution time for the sum of all the instructions by extracting the worst-case route through the hardware state machine to execute each possible operation. For example, applying two arguments to a primitive ALU function and evaluating it has a maximum runtime of 30 cycles — this includes the overhead of constructing an object in memory for the call, performing a function call, fetching the values of the operands, performing the operation, marking the reference as “evaluated” and saving the result, etc. In an average case, only a fraction of the possible overhead will actually be invoked (see section 7 for CPI averages).

Hardware garbage collection is a complicating factor on timing. GC can be configured to run at specific intervals or when memory usage reaches a certain limit; for our application, to guarantee real-time execution, the microkernel calls a hardware function to invoke the garbage collector once each iteration. To reason about how long the garbage collection takes, we bound the worst-case memory usage of a single iteration of the application loop. The hardware implements a semispace-based trace collector, so collection time is based on the live set, not how much memory was used in all. For the trace-collector state machine, each live object takes N+4 cycles to copy (for N memory words in the object), and it takes 2 cycles to check a reference to see if it’s already been collected. We bound the worst-case by conservatively assuming that all the memory that is allocated for one loop through the application might be simultaneously live at collection time, and that every argument in each function object may be a reference which the collector will have to spend 2 cycles checking.

From the static analysis, we determine that the worst execution of the entire loop is 4,686 cycles, not including garbage collection. Garbage collection is bounded by a worst-case of 4,379 cycles, making a total of 9,065 cycles to run one iteration of system — or 181.3 µs on our FPGA-synthesized prototype running at 50 MHz, falling well-within the real-time deadline of 5 ms.

### 6.3. Non-interference

Because the ICD coroutine has been proven correct (Section 6.1), we treat its output as trusted. This output must then travel through the rest of the cooperative microkernel until it reaches the outside world via the I/O coroutine’s `putint` primitive. In order to guarantee the integrity of this data (meaning it is never corrupted nor influenced by less-trusted data), we rely on a proof of non-interference. Non-interference means that “values of variables at a given security level $\ell \in \mathcal{L}$ can only influence variables at any security level that is greater than or equal to $\ell$ in the security lattice $\mathcal{L}$” [69]. In a standard security lattice, **L** (low-security) $\sqsubseteq$ **H** (high-security), meaning that high-security data does not flow to (or affect) low-security output. In our application, however, we are concerned with integrity; our lattice is composed of two labels, **T** (trusted) and **U** (untrusted), organized such that **T** $\sqsubseteq$ **U**. Therefore, our integrity non-interference property is that untrusted values cannot affect trusted values [11].

To prove this about Zarf, we create a simple integrity type system that provides a set of typing rules to determine and verify the integrity type of each expression, function, and constructor in a program. After providing trust-level annotations in a few places and constraining the normal Zarf semantics slightly to make type-checking much easier, we can run a type-checker over the resulting Zarf code to know whether it maintains data integrity. We extend the original Zarf syntax to allow for these type annotations, as follows:

$$\ell, \text{PC} \in \mathit{Label} ::= \mathbf{T} \mid \mathbf{U} \quad \tau \in \mathit{Type} ::= \mathbf{num}^{\ell} \mid (cn, \vec{\tau}) \mid (\vec{\tau} \to \tau)$$
$$\Gamma \in \mathit{Env} = \mathit{Identifier} \to \mathit{Type}$$

$$\frac{\Gamma[i_1 \mapsto \tau_1, \ \ldots, i_n \mapsto \tau_n] \vdash e : \tau}{\Gamma \vdash (\mathbf{fun}\ fn\ i_1 : \tau_1, \ldots, i_n : \tau_n\ : \tau = e) : (\tau_1, \ \ldots, \ \tau_n) \to \tau} \ (\text{FUNC})$$

$$\frac{\tau_1 = \Gamma(id) \quad \vec{\tau_2} = \Gamma(\vec{arg}) \quad \tau_3 = \texttt{applyType}(\tau_1, \vec{\tau_2}) \quad \Gamma[x \mapsto \tau_3] \vdash e : \tau_4}{\Gamma \vdash \mathbf{let}\ x = id\ \vec{arg}\ \mathbf{in}\ e : \tau_4} \ (\text{LET})$$

$$\frac{(cn, \vec{\tau_1}) = \Gamma(arg) \quad (cn\ \vec{x}\ \Rightarrow\ e_1) \in \vec{br} \quad \Gamma[\vec{x} \mapsto \vec{\tau_1}] \vdash e_1 : \tau_2}{\Gamma \vdash \mathbf{case}\ arg\ \mathbf{of}\ \vec{br}\ \mathbf{else}\ e_0 : \tau_2} \ (\text{CASE-CONS})$$

$$\frac{\begin{array}{c}\mathbf{num}^{\ell} = \Gamma(arg) \quad \Gamma \vdash e_1 : \tau_1 \ \ldots \ \Gamma \vdash e_n : \tau_n \quad \Gamma \vdash e_0 : \tau_0 \\ \tau = (\tau_0 \sqcup \tau_1 \sqcup \ldots \sqcup \tau_n) \bullet \ell\end{array}}{\Gamma \vdash \mathbf{case}\ arg\ \mathbf{of}\ n_1 \Rightarrow e_1\ \ldots\ n_n \Rightarrow e_n\ \mathbf{else}\ e_0 : \tau} \ (\text{CASE-LIT})$$

$$\frac{\tau = \Gamma(arg)}{\Gamma \vdash \mathbf{result}\ arg : \tau} \ (\text{RESULT}) \qquad \frac{\Gamma[x \mapsto \mathbf{num}^{\mathbf{T}}] \vdash e : \tau}{\Gamma \vdash \mathbf{let}\ x = \mathbf{getint}\ n\ \mathbf{in}\ e : \tau} \ (\text{GETINT})$$

$$\frac{\mathbf{num}^{\ell} = \Gamma(arg) \quad \Gamma[x \mapsto \mathbf{num}^{\ell}] \vdash e : \tau}{\Gamma \vdash \mathbf{let}\ x = \mathbf{putint}\ n\ arg\ \mathbf{in}\ e : \tau} \ (\text{PUTINT})$$

**Fig. 8.** Integrity Typing Rules. A type is inductively defined as either a labeled number, a singleton constructor, or a function constructed of these types. The type environment maps variables, function, and constructor names to types. Since all functions are annotated with their types, type checking proceeds by ensuring that the return type of a function is the same as the type deduced by checking the function's body expression with the function's parameter types added to the type environment. $\sqcup$ denotes the join of two types, and $\bullet$ denotes the joining of a type's integrity label with another.

$$\begin{aligned}
\ell, \text{PC} &\in \mathit{Label} ::= \mathbf{T} \mid \mathbf{U} \\
\tau &\in \mathit{Type} ::= \mathbf{num}^{\ell} \mid (cn, \vec{\tau}) \mid (\vec{\tau} \to \tau) \\
\mathit{func} &\in \mathit{Function} ::= \mathbf{fun}\ fn\ x_1 : \tau_1, \ldots, x_n : \tau_n\ : \tau = e \\
\mathit{cons} &\in \mathit{Constructor} ::= \mathbf{con}\ cn\ x_1 : \tau_1, \ldots, x_n : \tau_n
\end{aligned}$$

Specifically, following the spirit of Abadi et al. [6] and Simonet [70], types are inductively defined as either labeled numbers, or functions and constructors composed of other types. Our proof of soundness on this type system follows the approach done in work by Volpano et al. [7]. We show that if an expression $e$ has some specific type $\tau$ and evaluates to some value $v$, then changing any value whose type is less-trusted than $e$'s type results in $e$ evaluating to the same value $v$; thus, we show that arbitrarily changing untrusted data cannot affect trusted data. We prove soundness case-wise over the three types of expressions in our language, combining our evaluation semantics with our security typing rules.

*6.3.1. Integrity type system*

The integrity type system is found in Fig. 8, with its helper functions found in Fig. 9. Our integrity lattice is composed of two elements, **T** and **U** (trusted and untrusted, respectively), such that $\mathbf{T} < \mathbf{U}$ (opposite of a normal *security* lattice). We extended the Zarf ISA by requiring function and constructor type annotations. Constructors, which previously were untyped, are now singleton types: each constructor declaration defines a type, but that constructor is the sole inhabitor of the type. This restriction eliminates **case** expressions as sources of control flow when casing on a constructor type (since we know statically that the **case** expression's scrutinee will be a single unique value, and therefore also statically know which branch will be taken); note that this also eliminates the consideration of the **else** branch in a case expression on a constructor type. Instead, **case** expressions in this lightly-typed Zarf are solely for binding a constructor's internal values to variables (via deconstruction). Though this causes a loss in expressive power in the general case (constructors must be singleton types), our microkernel was designed without the need for this type of control flow.

**case** expressions whose scrutinee is a number, however, still allow for control flow (since the value of an **num** is *not* known ahead of time); therefore, the type of this form of case expression is the join of all of its branch types. The type of the scrutinee, which is significant in a security analysis, is here irrelevant — there are no implicit flows for integrity. Because we do not use union types, another small restriction we enforce is that each branch in a case expression must result in the same base type (i.e. all must either type-check to a **num**, $(cn, \vec{\tau})$, or $\vec{\tau} \to \tau$), such that we may join them together properly (see Fig. 10).

$$\texttt{applyType}((\vec{\tau}_1 \to \tau), \vec{\tau}_2) = \begin{cases} \tau & \text{if } |\vec{\tau}_1| = 0 \text{ and } |\vec{\tau}_2| = 0 \\ (\vec{\tau}_1 \to \tau) & \text{if } |\vec{\tau}_1| > 0 \text{ and } |\vec{\tau}_2| = 0 \\ \texttt{applyType}(\vec{\tau}_3 \to \tau, \vec{\tau}_4) & \text{if } \vec{\tau}_1 = \tau_1 :: \vec{\tau}_3,\ \vec{\tau}_2 = \tau_2 :: \vec{\tau}_4, \text{ and } \tau_2 \le \tau_1 \\ \texttt{applyType}(\vec{\tau}_3 \to \tau_4, \vec{\tau}_2) & \text{if } |\vec{\tau}_1| = 0,\ |\vec{\tau}_2| > 0, \text{ and } \tau = (\vec{\tau}_3 \to \tau_4) \end{cases}$$

$$\Gamma(\mathrm{n}) = \mathbf{num}^{\mathrm{PC}} \qquad \Gamma(id) = \begin{cases} (\vec{\tau} \to (cn, \vec{\tau})) & \text{if } id = cn \text{ and } \mathbf{con}\ cn\ \vec{x}{:}\vec{\tau} \in \overrightarrow{decl} \\ (\vec{\tau} \to \tau) & \text{if } id = fn \text{ and } \mathbf{fun}\ fn\ \vec{x}{:}\vec{\tau}\ {:}\tau = e \in \overrightarrow{decl} \\ \tau & \text{if } id = x \text{ and } (x \mapsto \tau) \in \Gamma \end{cases}$$

**Fig. 9.** Integrity Typing Rules Helpers. $\Gamma$ is a helper function that gets the type of an argument, and `applyType` applies a function type to argument types. Applying a helper function that takes one argument to a list of arguments is shorthand for mapping that function over the list.

$$\mathbf{num}^{\ell} \sqcup \mathbf{num}^{\ell'} = \mathbf{num}^{\ell \sqcup \ell'}$$

$$(cn, \vec{\tau}) \sqcup (cn, \vec{\tau}) = (cn, \vec{\tau})$$

$$(\vec{\tau} \to \tau) \sqcup (\vec{\tau}' \to \tau') = (\vec{\tau} \sqcup \vec{\tau}' \to \tau \sqcup \tau')$$

$$\mathbf{num}^{\ell} \bullet \ell' = \mathbf{num}^{\ell \sqcup \ell'}$$

**Fig. 10.** Joining Two Types. The $\bullet$ operator is used to join a type's label with another label; if the type that the label is being joined with is not a **num**, the label will be joined with each of the type's inner types until a base **num** is reached. Joining two lists of types is equal to the pairwise join of their elements. Constructor join is trivial because constructors are singletons whose type never changes, and only equal constructors can be compared.

$$\frac{\ell \sqsubseteq \ell'}{\mathbf{num}^{\ell} \le \mathbf{num}^{\ell'}}\ (\text{NUM}) \qquad \frac{\vec{\tau}' \le \vec{\tau} \quad \tau \le \tau'}{(\vec{\tau} \to \tau) \le (\vec{\tau}' \to \tau')}\ (\text{FUNC})$$

$$\frac{}{(cn, \vec{\tau}) \le (cn, \vec{\tau})}\ (\text{CONS}) \qquad \frac{\tau_1 \le \tau_2 \quad \tau_2 \le \tau_3}{\tau_1 \le \tau_3}\ (\text{TRAN})$$

**Fig. 11.** Subtyping Rules. One type is a subtype of another if their base types are equal and, in the case of the base num type, the first's label is lower in the integrity lattice than the other. A list is a subtype of another if pairwise each element of the first is a subtype of the corresponding element in the other list.

The integrity label associated with a **num** depends on the integrity level of the code that created it: untrusted code can only create numbers of type $\mathbf{num}^{\mathbf{U}}$, while trusted code can create trusted numbers (which can be treated as untrusted numbers via subtyping; see Fig. 11). Primitive operations (add, subtract, etc.) are treated as named functions contained within the set of declarations $\overrightarrow{decl}$. The type of primitive operators is dependent on the trust level of the caller: for example, the type of `add` is $\mathbf{num}^{\ell_1} \to \mathbf{num}^{\ell_2} \to \mathbf{num}^{\ell_1 \sqcup \ell_2 \sqcup \mathrm{PC}}$, where PC represents the trust level of the current program location (we assume its value can be tracked and changed outside of the type system proper). This all implies that untrusted code cannot use the primitive operations to create any type of trusted value (regardless of the types of the numbers an untrusted caller uses), thus restricting untrusted code's ability to obtain trusted values to (1) the `getint` function (which in our application is data straight from the heart monitor) and (2) by calling trusted functions which return trusted values.

This system will verify integrity for a value with singular endpoints — i.e., for the code being checked, it is received at one point and sent at one point. More complex annotations and treatment of values, like an arbitrary number of mutually untrusted but critical values passing through an arbitrary number of trusted and untrusted regions, can be guaranteed with this type-system via piecewise checking. By guaranteeing each link in the chain one-at-a-time, the integrity of the chain is verified.

The soundness proof of the integrity type system proceeds by cases on the three forms of expressions.

**Lemma 1** *(Case expression soundness). If* $\forall x\ e_0\ e_1\ arg_1\ arg_2\ \tau_0\ \tau_1\ cn_1\ \vec{t}_1\ v_1\ v_2\ \rho\ \Gamma$, *where*

1. $e_0 = (\textbf{\textit{case}}\ \mathrm{arg}_1\ \textbf{\textit{of}}\ \overrightarrow{br}\ \textbf{\textit{else}}\ e_1)$
2. $\Gamma \vdash e_0 {:} \tau_0\ \wedge\ \rho \vdash e_0 \Downarrow v_1\ \wedge\ \rho(x) = arg_1$
3. $\Gamma \vdash arg_2 {:} \tau_1\ \wedge\ \tau_1 > \tau_0 \wedge \rho \vdash arg_2 \Downarrow\ v_2$

*then* $\rho[x \mapsto v_2] \vdash e_0 \Downarrow v_1$.

**Proof.** 1. If $\Gamma \vdash arg_1 {:} \mathbf{num}^{\ell}$, then $\forall n\ e_2\ \ldots\ e_m$, $e_0 = (\mathbf{case}\ n\ \mathbf{of}\ n_2 \Rightarrow e_2\ \ldots\ n_m \Rightarrow e_m\ \mathbf{else}\ e_1)$, and either $\ell = \mathbf{T}$ or $\ell = \mathbf{U}$. We show that regardless of $arg_2$'s level when it is of type **num**, it cannot be changed and therefore $e_0$'s value doesn't change.

(a) If $\exists \ell_1 \in \tau_0$ s.t. $\ell_1 = \mathbf{T}$, then by typing rule CASE-LIT and the rule for join, $n$'s integrity label is $\mathbf{T}$. Therefore, $arg_1$ cannot both equal $n$ and be arbitrarily changed to some expression $arg_2$ because it is not an expression whose type

label is less trusted than the type of the entire expression (i.e. $\mathbf{num}^{\mathbf{T}} \ngeq \tau_0$). Thus we cannot replace $arg_1$ with $arg_2$, so in this case the value of $e_0$ remains the same, as desired. Since $e_1$ through $e_{m+1}$ are expressions whose soundness with respect to the type system can be considered separately through Lemmas 1, 2, and 3, we do not consider them here.
(b) If $\exists \ell_1 \in \tau_0$ s.t. $\ell_1 = \mathbf{U}$, then by our definition of the $\mathbf{T}-\mathbf{U}$ integrity lattice, there can be no values whose type is greater than $\tau_0$ ($arg_1$ included) that we can change. Therefore, $e_0$ remains unchanged, satisfying our conclusion.

2. If $\Gamma \vdash arg_1 : (cn, \vec{\tau_1})$, then $\forall cn_3 \ \ldots \ cn_n \ \vec{x}_3 \ \ldots \ \vec{x}_n \ e_3 \ \ldots \ e_n$,
$e_0 \ = \ (\textbf{case}\ (cn, \vec{\tau_1})\ \textbf{of}\ cn_3\ \vec{x}_3 \Rightarrow e_3\ \ldots\ cn_n\ \vec{x}_n \Rightarrow e_n\ \textbf{else}\ e_1)$.
   - We know by the operational semantics (restricted to accommodate this type system, with singleton constructor types) that which branch we case on is determined entirely by the constructor that $arg_1$ evaluates to, and **not** the values contained within that constructor. Therefore, changing the expressions within any constructor will result in the same branch being taken, such that $e_0$ evaluates to the branch's right-hand-side expression. Therefore, we cannot choose to replace $arg_1$ with another arbitrary $arg_2$ when $\Gamma \vdash arg_1 : (cn, \vec{\tau_1})$.
   - Let $(cn_3\ \vec{x}_3 \Rightarrow e_3)$ be the matching branch (where $cn = cn_3$). Based on the previous bullet point, we know that changing the expressions of any other branches will not change the value of the entire case expression, so we focus on this particular branch as an example. We must show that $\forall \tau_3, \exists x_3 \in \vec{x}_3$ s.t. $\Gamma \vdash x_3 : \tau_3 > \tau_0$, changing the value that $x_3$ maps to in $\rho$ does not change the value that $e_3$ evaluates to; that is, $\rho[x_3 \mapsto v_3] : e_3 \Downarrow v$, where $\rho(arg_2) = v_3$. Since $e_3$ is an expression, its soundness is either covered by Lemma 1 (by induction) or Lemmas 2 or 3. □

**Lemma 2** *(Result expression soundness). If* $\forall x\ e_0\ arg_1\ arg_2\ \tau_1\ \tau_2\ v_1\ \rho\ \Gamma$, *where*

1. $e_0 \ = \ (\textbf{\textit{result}}\ arg_1) \wedge \rho \vdash e_0 \Downarrow v_1$
2. $\Gamma \vdash arg_1 : \tau_1 \wedge arg_2 = \rho(arg_1)$
3. $\rho \vdash arg_2 : \tau_2 \wedge \tau_2 > \tau_1 \wedge \rho \vdash arg_2 \Downarrow v_2$

*then* $\rho[(x \mapsto v_2)] \vdash e_0 \Downarrow v_1$

**Proof.** The **result** expression is used for wrapping a value into a single expression containing that value. Therefore, changing the value of $arg_1$ to $arg_2$ would change the resultant value $v_1$ that $e_0$ is given, contradicting our result. As another point, by the typing rule RESULT, **result**'s type is precisely the type of $arg_1$, meaning there are no values within $e_0$ to change that would not cause us to violate (3) above. Therefore, the value of $arg_1$ must equal the value of $arg_2$ such that value of $e_0$ cannot change. □

**Lemma 3** *(Let expression soundness). If* $\forall e_0\ e_1\ x\ id\ arg_1\ arg_2\ \overrightarrow{arg_3}\ \overrightarrow{arg_4}\ \tau_1\ \tau_2\ v_1\ v_2\ v_3\ \vec{v}_3\ \vec{v}_4\ v_5\ \rho\ \Gamma$, *where*

1. $e_0 \ = \ (\textbf{\textit{let}}\ x = id\ \overrightarrow{arg_3}\ \textbf{\textit{in}}\ e_1)$
2. $\Gamma \vdash e_0 : \tau_1 \ \wedge \ \rho \vdash e_0 \Downarrow v_1$
3. $\Gamma \vdash id : \vec{\tau} \rightarrow \tau$
4. $\rho(\overrightarrow{arg_3}) = \vec{v}_3$
5. $\Gamma(arg_1) = \tau_2 \ \wedge \ \tau_2 > \tau_1$
6. $arg_0 \in \overrightarrow{arg_3} \ \wedge \ \overrightarrow{arg_4} = \overrightarrow{arg_3} - arg_0 + arg_1$
7. $\rho(\overrightarrow{arg_4}) = \vec{v}_4$
8. $(id \in \overrightarrow{cons} \wedge \texttt{applyCn}(id, \vec{v}_3) = v_2) \vee (\texttt{applyFn}(id, \vec{v}_3, \rho) = v_2)$
9. $(id \in \overrightarrow{cons} \wedge \texttt{applyCn}(id, \vec{v}_4) = v_3) \vee (\texttt{applyFn}(id, \vec{v}_4, \rho) = v_3)$
10. $v_2 = v_3$
11. $\rho[x \mapsto v_3] \vdash e_1 \Downarrow v_5$

*then* $v_1 = v_5$.

**Proof.** By cases on $id$:

1. If $id$ is a primitive function (add, multiply, etc.), then $v_2 \neq v_3 \iff \overrightarrow{arg_3} \neq \overrightarrow{arg_4}$. By the typing rule of primitives, the type $\tau$ that the function returns is the least upper bound of all of its arguments, including $arg_1$, meaning by definition, both the value and type of the primitive operation are entirely dependent on all arguments. Therefore, there cannot exist an $arg_2$ that allows us to substitute it for $arg_1$ whose type is less trusted than $\tau$ without changing the entire value $v_1$.
2. If $id$ is a constructor, then $id$ has the type $\overrightarrow{\tau} \rightarrow (cn, \vec{\tau})$. $id$'s return type is determined statically and does not change throughout program execution. Therefore, there does not exist a subexpression in $\overrightarrow{arg_3}$, or more generally, in $e_0$, that can be changed without changing the type of the constructor, which would contradict our having the same values after evaluation.

3. If *id* is a non-recursive function composed solely of case and result expressions and applications of primitive functions and constructors used in let expressions, then by (1), (2), Lemmas 1 and 2 and induction on Lemma 3, we know *id* must be sound. By extension, if *id* calls a function that fulfills these requirements, one can unfold the called function's contents in order to see that the resultant value $v_2$ satisfies this case.
4. If *id* is a recursive function or calls a function which calls *id* (i.e. mutual recursion), it is possible that the function call never terminates and therefore never results in a single value. The soundness of $e_0$ must then be guaranteed via induction on possible expressions, proven in the previous lemmas. We know statically that the type of *id* is of the form $\vec{\tau} \rightarrow \tau$, so we are guaranteed via simplification rules in the `apply` helper functions that types of $\overrightarrow{arg}_3$ must be equal to or subtypes of $\vec{\tau}$, or otherwise our operational semantics would get stuck. By induction, any recursive calls made in $e_1$ must also satisfy this lemma, meaning that the actual arguments $\overrightarrow{arg}_3$ are used properly, otherwise $e_0$ wouldn't type check to type $\tau_1$ by getting stuck.

By proving that $v_2$'s value does not change when less-trusted values change, we can safely continue with the evaluation of $e_1$, which will be a **case**, **result**, or **let**, all of which are handled in Lemma 1, Lemma 2, and Lemma 3, respectively. □

**Theorem 1** *(Integrity type system soundness). Our integrity type system is sound if, given some expression* ***e*** *of type $\tau$ which evaluates to some value* ***v****, we can show that we can arbitrarily change any (or all) expressions in* ***e*** *which are less trusted than $\tau$ so that* ***e*** *still evaluates to* ***v****; i.e., untrusted data does not affect trusted data.*

*Formally, if* $\forall e_1\ e_2\ e_3\ e_4\ \tau_1\ \tau_2\ \tau_3\ v\ \rho\ \Gamma$, *where*

1. $\rho \vdash e_1 \Downarrow v \ \wedge\ \Gamma \vdash e_1 : \tau_1$
2. $e_2 \in \mathtt{subexprs}\,(e_1) \ \wedge\ \Gamma \vdash e_2 : \tau_2 \ \wedge\ \tau_2 > \tau_1$
3. $\Gamma \vdash e_3 : \tau_3 \ \wedge\ \tau_3 \geq \tau_2$

*then* $\rho \vdash e_1[e_3/e_2] \Downarrow v$

**Proof.** There are just three types of expressions: **let**, **case**, and **result**. By Lemma 1, we show that **case** expressions (the vehicle for control-flow) are sound. By Lemma 2, we show that **result** expressions are sound. Likewise, by Lemma 3, we show that **let** expressions (the vehicle for function application) are sound. Thus, we have exhaustively shown soundness of all expressions. Furthermore, we can see that when these expressions are composed according to the abstract syntax, with the additional typing annotations and a few restrictions, any well-typed Zarf program has the property of non-interference with respect to integrity, even while using a simplistic type system such as that explained here. □

### 6.4. Programmer responsibility

We have demonstrated that there are varying degrees of responsibility a Zarf programmer can take when writing their application, each involving greater effort. The first is doing the minimum: the programmer writes their program in Zarf assembly. A major advantage of Zarf is that the application automatically gains the benefits of memory and control-flow safety inherent in the ISA, properties that other ISAs don't easily offer. Any well-formed application that runs on Zarf gets these properties without any additional programmer involvement.

The second degree of responsibility that can be taken is writing the application's specification in Coq and automatically lowering it to Zarf to prove its correctness. This approach involves a non-trivial amount of proof-writing, but since the ISA resembles the language of verification very closely, we argue that the amount of work involved relative to doing so over other imperative ISAs is significantly less. Since high-level specification and verification of critical applications is common practice, this level of programmer responsibility is not usual. Gladly, however, any future proof efforts might not need to be entirely application-specific. Given the exercise of proving the correctness of the Zarf implementation of the ICD algorithm, we now have a set of theorems and proofs showing the equivalence between common user-made Zarf functions and Coq versions. It is conceivable that verification in Coq of future Zarf applications could reuse this underlying work.

The third degree of responsibility involves proving additional properties over the system, beyond the aforementioned safety and correctness. We demonstrated this by laying a security type system over the ISA, somewhat restricting it (like all type systems are wont to do) in exchange for the added property of non-interference. Because the process of writing a type system and checker are sufficiently general, we can see additional type systems or analyses being made over the base Zarf ISA relatively easily.

Finally, there is the issue of determining which parts of an application should go into each hardware execution realm. Zarf has two execution realms due in part to the assumption that users might want to include legacy or high performance, non-critical code; this code can run on the imperative ISA. However, any activity providing critical functionality for safe operation should happen in the functional processor. Veridrone [71] is an example of another project beyond our ICD that might benefit from this approach; that project uses both a lower-performance core safety control system and a higher-performance unverified version that is more energy-efficient and allows for smoother flying.

**Table 1**
Resource usage of Zarf and basic MicroBlaze (3-stage pipeline), the two layers of Zarf, when synthesized for a Xilinx Artix-7 FPGA. In total, the logic of Zarf uses 29,980 gates.

| Resource | Zarf | MicroBlaze |
|---|---|---|
| LUTs | 4,337 | 1,840 |
| FFs | 2,779 | 1,556 |
| Cycle Time | 20 ns (50 MHz) | 10 ns (100 MHz) |

## 7. Evaluation

To validate our designs, we download the Zarf hardware specification onto a Xilinx Artix-7 FGPA and run our sample application. For a comparison, we also run a completely unverified C version of the application on a Xilinx MicroBlaze on the same FPGA. Hardware synthesis results are summarized in Table 1.

The hardware description of Zarf is more complex than a simple embedded CPU, with 66 total states of control logic (4 deal with program loading, 15 with function application, 18 with function evaluation, and 29 with garbage collection). In all, the combinational logic takes 29,980 primitive gates (roughly the size of a MIPS R3000), or 4,337 LUTs when synthesized for an Artix-7 FPGA (less than 7% of the available logic resources). Estimated on 130 nm, the combinational logic takes up .274 mm$^2$. Though larger than very simple embedded CPUs, Zarf is still quite a bit smaller than many common embedded microcontrollers.

From a dynamic trace of several million cycles, the ICD application exhibited the following average CPI for each instruction type. `Let` instructions had an average of 5.16 arguments and took on average 10.36 cycles. `Case` instructions averaged at 10.59 cycles; each branch head in a `case` takes exactly 1 cycle to check if the branch matches. `Results` took 11.01 cycles on average. The total dynamic CPI across the trace was 7.46 (or 11.86 if garbage collection time is included). Approximately one third of the dynamic instructions were branch heads.

The C version of the ICD application on the MicroBlaze takes fewer than one thousand cycles for each iteration of the application. The analysis in section 6.2 discusses the worst-case runtime of the Zarf application, which is around 9,000 cycles or 180 μs (though much faster in the typical case). This is in addition to a longer cycle time (see Table 1). When compared to the carefully optimized and tiny MicroBlaze, our experimental prototype uses approximately twice the hardware resources, and the application is around 20x slower in the worst case than MicroBlaze in the common case — but is still over 25 times faster than it needs to be to meet the critical real-time deadlines, all while adding invaluable guarantees about the correctness of the most critical application components and assurance of non-interference between separate functions.

## 8. Conclusion

As computing continues to automate and improve the control of life-critical systems, new techniques which ease the development of formally trustworthy systems are sorely needed. The system approach demonstrated in this work shows that deep and *composable* reasoning directly on machine instructions is possible when the architecture is amenable to such reasoning. Our prototype implementation of this concept uses Zarf to control the operation of critical components in a way that allows assembly-level verified versions of critical code to operate safely in close partnership with more traditional and less-verified system components without the need to include run-times and compilers in the TCB. We take a holistic approach to the evaluation of this idea, not only demonstrating its practicality through an FPGA-implemented prototype, but furthermore showing the successful application of three different forms of static analysis at the assembly level of Zarf.

As we move to increasingly diverse systems on chip, heterogeneity in semantic complexity is an interesting new dimension to consider. A very small core supporting highly critical workloads might help ameliorate critical bugs, vulnerabilities, and/or excessive high-assurance costs. A core executing the Zarf ISA would take up roughly 0.002% of a modern SoC. Our hope is that this work will begin a broader discussion about the role of formal methods in computer architecture design and how it might be embraced as a part, rather than an afterthought, of the design process.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1239567, 1162187, and 1563935.

We'd like to thank Sung-Yee Guo, Nicholas Brown, Benjamin Campbell, Tristan Konologie, Bingbin 'Clara' Liu, and Benjamin Spitz for their work on related system infrastructure, and Kyle Dewey, for his system and type system critiques.

## References


[1] J. McMahan, M. Christensen, L. Nichols, J. Roesch, S.-Y. Guo, B. Hardekopf, T. Sherwood, An architecture supporting formal and compositional binary analysis, Comput. Archit. News 45 (1) (2017) 177–191, https://doi.org/10.1145/3093337.3037733.

[2] R. Mangharam, H. Abbas, M. Behl, K. Jang, M. Pajic, Z. Jiang, Three challenges in cyber-physical systems, in: 2016 8th International Conference on Communication Systems and Networks (COMSNETS), 2016, pp. 1–8.

[3] S. Shuja, S.K. Srinivasan, S. Jabeen, D. Nawarathna, A formal verification methodology for DDD mode pacemaker control programs, J. Electr. Comput. Eng. (2015), https://doi.org/10.1155/2015/939028.

[4] J.M. Rushby, Proof of separability: a verification technique for a class of a security kernels, in: Proceedings of the 5th Colloquium on International Symposium on Programming, Springer-Verlag, London, UK, 1982, pp. 352–367, http://dl.acm.org/citation.cfm?id=647325.721663.

[5] N. Heintze, J.G. Riecke, The slam calculus: programming with secrecy and integrity, in: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98, ACM, New York, NY, USA, 1998, pp. 365–377, http://doi.acm.org/10.1145/268946.268976.

[6] M. Abadi, A. Banerjee, N. Heintze, J.G. Riecke, A core calculus of dependency, in: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '99, ACM, New York, NY, USA, 1999, pp. 147–160, http://doi.acm.org/10.1145/292540.292555.

[7] D. Volpano, C. Irvine, G. Smith, A sound type system for secure flow analysis, J. Comput. Secur. 4 (2–3) (1996) 167–187, http://dl.acm.org/citation.cfm?id=353629.353648.

[8] D.E. Denning, P.J. Denning, Certification of programs for secure information flow, Commun. ACM 20 (7) (1977) 504–513, https://doi.org/10.1145/359636.359712, http://doi.acm.org/10.1145/359636.359712.

[9] J.A. Goguen, J. Meseguer, Security policies and security models, in: Security and Privacy, 1982 IEEE Symposium on, 1982, p. 11.

[10] F. Pottier, V. Simonet, Information flow inference for ML, ACM Trans. Program. Lang. Syst. 25 (1) (2003) 117–158, https://doi.org/10.1145/596980.596983, http://doi.acm.org/10.1145/596980.596983.

[11] A. Sabelfeld, A.C. Myers, Language-based information-flow security, IEEE J. Sel. Areas Commun. 21 (1) (2006) 5–19, https://doi.org/10.1109/JSAC.2002.806121.

[12] D. Terei, S. Marlow, S. Peyton Jones, D. Mazières, Safe Haskell, in: Proceedings of the 2012 Haskell Symposium, Haskell '12, ACM, New York, NY, USA, 2012, pp. 137–148, http://doi.acm.org/10.1145/2364506.2364524.

[13] D. Yu, N.A. Hamid, Z. Shao, Building certified libraries for PCC: dynamic storage allocation, in: Proceedings of the 12th European Conference on Programming, Springer-Verlag, 2003, pp. 363–379.

[14] A. Chlipala, Mostly-automated verification of low-level programs in computational separation logic, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, ACM, New York, NY, USA, 2011, pp. 234–245, http://doi.acm.org/10.1145/1993498.1993526.

[15] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, Arvind, Kami: a platform for high-level parametric hardware specification and its modular verification, Proc. ACM Program. Lang. 1 (ICFP) (Aug. 2017), https://doi.org/10.1145/3110268.

[16] R.S. Boyer, Y. Yu, Automated correctness proofs of machine code programs for a commercial microprocessor, in: Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction, CADE-11, Springer-Verlag, London, UK, 1992, pp. 416–430, http://dl.acm.org/citation.cfm?id=648230.752650.

[17] N.G. Michael, A.W. Appel, Machine instruction syntax and semantics in higher order logic, in: Proceedings of the 17th International Conference on Automated Deduction, CADE-17, Springer-Verlag, London, UK, 2000, pp. 7–24, http://dl.acm.org/citation.cfm?id=648236.761384.

[18] A. Kennedy, N. Benton, J.B. Jensen, P.-E. Dagand, Coq: the world's best macro assembler?, in: Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming, PPDP '13, ACM, New York, NY, USA, 2013, pp. 13–24, http://doi.acm.org/10.1145/2505879.2505897.

[19] A. Fox, M.O. Myreen, A trustworthy monadic formalization of the armv7 instruction set architecture, in: Proceedings of the First International Conference on Interactive Theorem Proving, ITP'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 243–258.

[20] J.S. Moore, A mechanically verified language implementation, J. Autom. Reason. 5 (4) (1989) 461–492.

[21] W.A. Hunt Jr, Microprocessor design verification, J. Autom. Reason. 5 (4) (1989) 429–460.

[22] J. Autom. Reason. (2003).

[23] G.C. Necula, Proof-Carrying Code. Design and Implementation, Springer, 2002.

[24] A.W. Appel, Foundational proof-carrying code, in: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, LICS '01, IEEE Computer Society, Washington, DC, USA, 2001, p. 247, http://dl.acm.org/citation.cfm?id=871816.871860.

[25] J. Yang, C. Hawblitzel, Safe to the last instruction: automated verification of a type-safe operating system, in: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, ACM, New York, NY, USA, 2010, pp. 99–110, http://doi.acm.org/10.1145/1806596.1806610.

[26] T. Maeda, A. Yonezawa, Typed assembly language for implementing OS kernels in SMP/multi-core environments with interrupts, in: Proceedings of the 5th International Conference on Systems Software Verification, SSV'10, USENIX Association, Berkeley, CA, USA, 2010, p. 1, http://dl.acm.org/citation.cfm?id=1929004.1929005.

[27] M. Barnett, B.-Y.E. Chang, R. DeLine, B. Jacobs, K.R.M. Leino, Boogie: a modular reusable verifier for object-oriented programs, in: Proceedings of the 4th International Conference on Formal Methods for Components and Objects, FMCO'05, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 364–387.

[28] H. Xi, R. Harper, A dependently typed assembly language, in: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming, ICFP '01, ACM, New York, NY, USA, 2001, pp. 169–180, http://doi.acm.org/10.1145/507635.507657.

[29] A. Chlipala, A verified compiler for an impure functional language, in: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '10, ACM, New York, NY, USA, 2010, pp. 93–106, http://doi.acm.org/10.1145/1706299.1706312.

[30] G.C. Necula, Translation validation for an optimizing compiler, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00, ACM, New York, NY, USA, 2000, pp. 83–94, http://doi.acm.org/10.1145/349299.349314.

[31] P. Curzon, P. Curzon, A verified compiler for a structured assembly language, in: Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications, IEEE Computer, Society Press, 1991, pp. 253–262.

[32] M. Strecker, Formal verification of a Java compiler in Isabelle, in: Automated Deduction—CADE-18, Springer, 2002, pp. 63–77.

[33] X. Leroy, A formally verified compiler back-end, J. Autom. Reason. 43 (4) (2009) 363–446, http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

[34] A.W. Appel, Verified software toolchain, in: Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software, ESOP'11/ETAPS'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 1–17, http://dl.acm.org/citation.cfm?id=1987211.1987212.

[35] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, V. Vafeiadis, Pilsner: a compositionally verified compiler for a higher-order imperative language, in: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, ACM, New York, NY, USA, 2015, pp. 166–178, http://doi.acm.org/10.1145/2784731.2784764.

[36] T. Ramananandro, Z. Shao, S.-C. Weng, J. Koenig, Y. Fu, A compositional semantics for verified separate compilation and linking, in: Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP '15, ACM, New York, NY, USA, 2015, pp. 3–14, http://doi.acm.org/10.1145/2676724.2693167.

[37] Z. Jiang, M. Pajic, R. Mangharam, Cyber-physical modeling of implantable cardiac medical devices, Proc. IEEE 100 (1) (2012) 122–137, https://doi.org/10.1109/JPROC.2011.2161241.

[38] A.O. Gomes, M.V.M. Oliveira, Formal specification of a cardiac pacing system, in: A. Cavalcanti, D.R. Dams (Eds.), FM 2009: Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 692–707.

[39] T. Chen, M. Diciolla, M. Kwiatkowska, A. Mereacre, Quantitative verification of implantable cardiac pacemakers, in: Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd, IEEE, 2012, pp. 263–272.

[40] L. Cordeiro, B. Fischer, H. Chen, J. Marques-Silva, Semiformal verification of embedded software in medical devices considering stringent hardware constraints, in: 2009 International Conference on Embedded Software and Systems, 2009, pp. 396–403.

[41] P.J. Landin, The mechanical evaluation of expressions, Comput. J. 6 (4) (1964) 308–320, https://doi.org/10.1093/comjnl/6.4.308.

[42] B. Graham, SECD: Design issues, Tech. Rep., University of Calgary, 1989, http://hdl.handle.net/1880/46590.

[43] T.J. Clarke, P.J. Gladstone, C.D. MacLean, A.C. Norman, SKIM - the S, K, I reduction machine, in: Proceedings of the 1980 ACM Conference on LISP and Functional Programming, LFP '80, ACM, New York, NY, USA, 1980, pp. 128–135, http://doi.acm.org/10.1145/800087.802798.

[44] L.P. Deutsch, A lisp machine with very compact programs, in: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers Inc., 1973, pp. 697–703.

[45] P.M. Kogge, The Architecture of Symbolic Computers, McGraw-Hill, Inc., New York, 1991.

[46] T.F. Knight, Implementation of a list processing machine, Ph.D. thesis, Massachusetts Institute of Technology, 1979.

[47] J.M. McCune, B.J. Parno, A. Perrig, M.K. Reiter, H. Isozaki, Flicker: an execution infrastructure for tcb minimization, Oper. Syst. Rev. 42 (4) (2008) 315–328, https://doi.org/10.1145/1357010.1352625, http://doi.acm.org/10.1145/1357010.1352625.

[48] E. Keller, J. Szefer, J. Rexford, R.B. Lee, NoHype: virtualized cloud infrastructure without the virtualization, Comput. Archit. News 38 (3) (2010) 350–361, https://doi.org/10.1145/1816038.1816010, http://doi.acm.org/10.1145/1816038.1816010.

[49] D. Halperin, T.S. Heydt-Benjamin, B. Ransford, S.S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, W.H. Maisel, Pacemakers and implantable cardiac defibrillators: software radio attacks and zero-power defenses, in: 2008 IEEE Symposium on Security and Privacy (SP 2008), IEEE, 2008, pp. 129–142.

[50] S. Gollakota, H. Hassanieh, B. Ransford, D. Katabi, K. Fu, They can hear your heartbeats: non-invasive security for implantable medical devices, in: Proc. ACM Conf. SIGCOMM, 2011, pp. 2–13.

[51] T. Denning, K. Fu, T. Kohno, Absence makes the heart grow fonder: new directions for implantable medical device security, in: Proceedings of the 3rd Conference on Hot Topics in Security, HOTSEC'08, USENIX Association, Berkeley, CA, USA, 2008, 5, http://dl.acm.org/citation.cfm?id=1496671.1496676.

[52] B. Hardekopf, C. Lin, Flow-sensitive pointer analysis for millions of lines of code, in: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 289–298, http://dl.acm.org/citation.cfm?id=2190025.2190075.

[53] E. Moggi, Notions of computation and monads, Inf. Comput. 93 (1) (1991) 55–92.

[54] P. Hudak, J. Hughes, S. Peyton Jones, P. Wadler, A history of Haskell: being lazy with class, in: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, ACM, 2007, 12.

[55] R. Hindley, The principal type-scheme of an object in combinatory logic, Trans. Am. Math. Soc. 146 (1969) 29–60, http://www.jstor.org/stable/1995158.

[56] R. Milner, A theory of type polymorphism in programming, J. Comput. Syst. Sci. 17 (1978) 348–375.

[57] M.E. Conway, Design of a separable transition-diagram compiler, Commun. ACM 6 (7) (1963) 396–408, https://doi.org/10.1145/366663.366704, http://doi.acm.org/10.1145/366663.366704.

[58] A.L.D. Moura, R. Ierusalimschy, Revisiting coroutines, ACM Trans. Program. Lang. Syst. 31 (2) (2009) 6, https://doi.org/10.1145/1462166.1462167, http://doi.acm.org/10.1145/1462166.1462167.

[59] S.J. Connolly, M. Gent, R.S. Roberts, P. Dorian, D. Roy, R.S. Sheldon, L.B. Mitchell, M.S. Green, G.J. Klein, B. O'Brien, Canadian implantable defibrillator study (CIDS), Circulation 101 (11) (2000) 1297–1302, https://doi.org/10.1161/01.CIR.101.11.1297, http://circ.ahajournals.org/content/101/11/1297.full.pdf, http://circ.ahajournals.org/content/101/11/1297.

[60] T. A. versus Implantable Defibrillators (AVID) Investigators, A comparison of antiarrhythmic-drug therapy with implantable defibrillators in patients resuscitated from near-fatal ventricular arrhythmias, N. Engl. J. Med. 337 (22) (1997) 1576–1584, https://doi.org/10.1056/NEJM199711273372202, pMID: 9411221.

[61] J. Siebels, K.-H. Kuck, C. Investigators, Implantable cardioverter defibrillator compared with antiarrhythmic drug treatment in cardiac arrest survivors (the cardiac arrest study Hamburg), Am. Heart J. 127 (1994) 1139–1144, https://doi.org/10.1016/0002-8703(94)90101-5.

[62] Living with your implantable cardioverter defibrillator (ICD), http://www.heart.org/HEARTORG/Conditions/Arrhythmia/PreventionTreatmentofArrhythmia/Living-With-Your-Implantable-Cardioverter-Defibrillator-ICD_UCM_448462_Article.jsp, 09 2016.

[63] How many people have ICDs? http://asktheicd.com/tile/106/english-implantable-cardioverter-defibrillator-icd/how-many-people-have-icds/. (Accessed 24 October 2019).

[64] J. Pan, W.J. Tompkins, A real-time QRS detection algorithm, IEEE Trans. Biomed. Eng. BME-32 (3) (1985) 230–236, https://doi.org/10.1109/TBME.1985.325532.

[65] R.A. Álvarez, A.J.M. Penín, X.A.V. Sobrino, A comparison of three QRS detection algorithms over a public database, in: CENTERIS 2013 - Conference on ENTERprise Information Systems/ProjMAN 2013 - International Conference on Project MANagement/HCIST 2013 - International Conference on Health and Social Care Information Systems and Technologies, Proc. Technol. 9 (2013) 1159–1165, https://doi.org/10.1016/j.protcy.2013.12.129, http://www.sciencedirect.com/science/article/pii/S2212017313002831.

[66] Open source ECG analysis software, http://www.eplimited.com/confirmation.htm. (Accessed 24 October 2019).

[67] M.S. Wathen, P.J. DeGroot, M.O. Sweeney, A.J. Stark, M.F. Otterness, W.O. Adkisson, R.C. Canby, K. Khalighi, C. Machado, D.S. Rubenstein, K.J. Volosin, Prospective randomized multicenter trial of empirical antitachycardia pacing versus shocks for spontaneous rapid ventricular tachycardia in patients with implantable cardioverter-defibrillators, Circulation 110 (17) (2004) 2591–2596, https://doi.org/10.1161/01.CIR.0000145610.64014.E4, http://circ.ahajournals.org/content/110/17/2591.full.pdf, http://circ.ahajournals.org/content/110/17/2591.

[68] The Coq proof assistant, https://coq.inria.fr. (Accessed 24 October 2019).

[69] V. Kashyap, B. Wiedermann, B. Hardekopf, Timing- and termination-sensitive secure information flow: exploring a new approach, in: 2011 IEEE Symposium on Security and Privacy, 2011, pp. 413–428.

[70] V. Simonet, Fine-grained information flow analysis for a λ calculus with sum types, in: Proceedings of the 15th IEEE Workshop on Computer Security Foundations, CSFW '02, IEEE Computer Society, Washington, DC, USA, 2002, p. 223, http://dl.acm.org/citation.cfm?id=794201.795163.

[71] D. Ricketts, G. Malecha, M.M. Alvarez, V. Gowda, S. Lerner, Towards verification of hybrid systems in a foundational proof assistant, in: 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2015, pp. 248–257.