An Architecture Supporting Formal and Compositional Binary Analysis

Joseph McMahan[†] Michael Christensen[†] Lawton Nichols[†] Jared Roesch[‡ *]
Sung-Yee Guo[†] Ben Hardekopf[†] Timothy Sherwood[†]
University of California, Santa Barbara[†]
University of Washington, Seattle[‡]
{ jmcmahan, mchristensen, lawtonnichols } @cs.ucsb.edu,
jroesch@cs.washington.edu, { sguo, benh, sherwood } @cs.ucsb.edu

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, including a proof of correctness of the assembly-level implementation of the core algorithm, the integrity of trusted data via a non-interference proof, and a guarantee that our prototype meets critical timing requirements.

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 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 no modern architecture has been 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 using a verified compiler, assuming other modules are correct, using only a subset of the ISA and assuming the rest

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

ASPLOS '17, April 08-12, 2017, Xi'an, China
© 2017 ACM. ISBN 978-1-4503-4465-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037733

is ununsed, 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 that proof and reasoning systems are already built upon. 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, observes the following properties:

1. The functional ISA, the "λ-execution layer," 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 the λ-execution layer 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, they, along with other embedded medical devices, have seen thousands of recalls due to dangerous software bugs [44; 59]. 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 Figure 1.

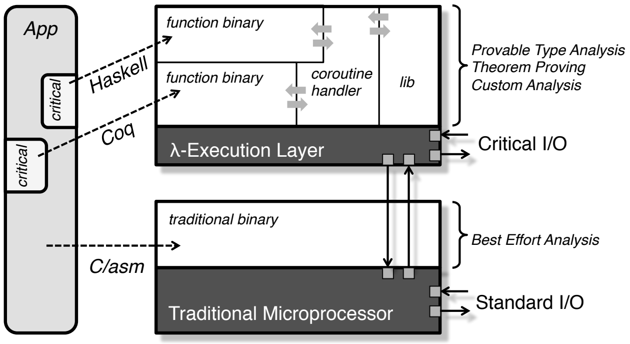


Figure 1. High-level Zarf system architecture: by dividing the system into two hardware realms — one that provides a precise, mathematical semantics for reasoning about program behavior, and the other a standard imperative core for legacy software — we can formally verify and otherwise reason about critical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

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, define the semantics precisely, 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 discussion of the verification of multiple properties of the critical sub-components of the ICD, covering correctness, timing, and non-interference (Section 5). 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 6), and conclude.

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 [57]. 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 [30] and its further development by Abadi et al. in their Core Calculus of Dependency [6]. It also draws inspiration from Volpano [66] 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 include Denning [20], Goguen [24], Pottier's information flow for ML [55], and Sabelfeld and Myers's survey on language-based information flow security [58].

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 [63] 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. [70] 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 [13], but limits available memory structures.

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 [10]; other examples include a formalization of the SPARC instruction set, including some of the more complex properties, such as branch delay slots [46]; and subsets of x86 [37]. One of the biggest efforts to date has been a formal model of the ARMv7 ISA using a monadic specification in HOL4 [23]. Moore developed Piton, a high level assembly language and a verified compiler for the FM8502 microprocessor [49], which complemented the verification work done on the FM8502 implementation [33]. 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 [5].

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 [52], where machine-checked proofs guarantee properties of a program [7]. Typed assemblies and intermediate representations have seen extensive use in the verification community [69; 13; 43; 9] and have been extended with dependent types [68], allowing for more expressive programs and proofs at the assembly level.

SeL4, a fully verified OS kernel [38], required several person-years of verification work. Verification is done at higher levels of abstraction to make the problem tractable, like modeling memory behavior at the C level [64].

Verified compilers are a popular topic in the verification community [12; 51; 19; 62], the most well-known example being CompCert [42], 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) [8] 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 [53; 56]. 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 [34], formal models of pacing systems in Z notation [26], quantitative and automated checking of the interaction of heart-pacemaker automata to verify pacemaker properties [11], and semi-formal verification by combining platform-dependent and independent model checking to exhaustively check the state space of an embedded system [17]. 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 [27], 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 [14] 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 [22; 40; 39]. 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 [39] describes a machine with instructions for Lisp primitives such as CAR and CADR, and also for complex operations like CALL and MOVE. While these 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 [45]. 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 [36] 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 [28; 25; 21]. The focus of our work is analysis and correctness properties, and we do not deal with security.

3. Hardware Architecture and ISA

Our system, Zarf, 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 the λ-execution layer: 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 the λ-execution layer as deemed necessary.

The following subsections describe the interface and construction of the λ-execution layer, 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 [53; 56]. 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 [23]).
4. Arbitrary control flow, which often requires complex and approximate analyses to soundly determine possible control flows [29].
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 type checker, the assembly is untyped; **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). One bit is attached to values at runtime to distinguish primitive integers from function objects; this prevents malformed code from placing the machine in an invalid state. Instructions use De Bruijn indices[1] to refer to data elements; since the references are localized and cannot refer to any global state, together with immutability it enforces referential transparency. The abstract syntax of the λ-execution layer assembly is given in Figure 2.

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

$$\begin{array}{rcl}
p \in \mathit{Program} & ::= & \overrightarrow{\mathit{decl}}\ \textbf{fun}\ \text{main}\ \textbf{=}\ e \\
\mathit{decl} \in \mathit{Declaration} & ::= & \mathit{cons} \mid \mathit{func} \\
\mathit{cons} \in \mathit{Constructor} & ::= & \textbf{con}\ \mathit{cn}\ \vec{x} \\
\mathit{func} \in \mathit{Function} & ::= & \textbf{fun}\ \mathit{fn}\ \vec{x}\ \textbf{=}\ e \\
e \in \mathit{Expression} & ::= & \mathit{let} \mid \mathit{case} \mid \mathit{res} \\
\mathit{let} \in \mathit{Let} & ::= & \textbf{let}\ x\ \textbf{=}\ \mathit{id}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \\
\mathit{case} \in \mathit{Case} & ::= & \textbf{case}\ \mathit{arg}\ \textbf{of}\ \overrightarrow{\mathit{br}}\ \textbf{else}\ e \\
\mathit{res} \in \mathit{Result} & ::= & \textbf{result}\ \mathit{arg} \\
\mathit{br} \in \mathit{Branch} & ::= & \mathit{cn}\ \vec{x}\ \Rightarrow\ e \mid n\ \Rightarrow\ e \\
\mathit{id} \in \mathit{Identifier} & ::= & x \mid \mathit{fn} \mid \mathit{cn} \mid \oplus \\
\mathit{arg} \in \mathit{Argument} & ::= & n \mid x
\end{array}$$

Figure 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. $\oplus$ 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.

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 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 starting at 0x100. 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

[1] This is effectively using the stack offset of each variable in the local frame; variables are placed on the "stack" automatically and static numbering is easily determined.

$$c \in \mathit{Constructor} = \mathit{Name} \times \overrightarrow{\mathit{Value}} \quad \mathit{clo} \in \mathit{Closure} = (\lambda\vec{x}.e) \times \overrightarrow{\mathit{Value}} \quad v \in \mathit{Value} = \mathbb{Z} \uplus \mathit{Constructor} \uplus \mathit{Closure} \quad \rho \in \mathit{Env} = \mathit{Variable} \rightarrow \mathit{Value}$$

$$\frac{\vdash e \Downarrow v}{\vdash \overrightarrow{\mathit{decl}}\ \textbf{fun}\ \text{main}\ \textbf{=}\ e \Downarrow v}\ (\textsc{program}) \qquad \frac{v = \rho(\mathit{arg})}{\rho \vdash \textbf{result}\ \mathit{arg} \Downarrow v}\ (\textsc{result}) \qquad \frac{\vec{v}_1 = \rho(\overrightarrow{\mathit{arg}}) \quad v_2 = \mathtt{applyCn}(\mathit{cn},\vec{v}_1) \quad \rho[x \mapsto v_2] \vdash e \Downarrow v_3}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ \mathit{cn}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \Downarrow v_3}\ (\textsc{let-con})$$

$$\frac{\mathit{fn} \notin \{\textbf{getint}, \textbf{putint}\} \quad \textbf{fun}\ \mathit{fn}\ \vec{x}_2\ \textbf{=}\ e_2 \in \overrightarrow{\mathit{decl}} \quad \vec{v}_1 = \rho(\overrightarrow{\mathit{arg}}) \quad v_2 = \mathtt{applyFn}((\lambda\vec{x}_2.e_2, []),\vec{v}_1,\rho) \quad \rho[x_1 \mapsto v_2] \vdash e_1 \Downarrow v_3}{\rho \vdash \textbf{let}\ x_1\ \textbf{=}\ \mathit{fn}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e_1 \Downarrow v_3}\ (\textsc{let-fun})$$

$$\frac{v_1 = \rho(x_2) \quad \vec{v}_2 = \rho(\overrightarrow{\mathit{arg}}) \quad v_3 = \mathtt{applyFn}(v_1,\vec{v}_2,\rho) \quad \rho[x_1 \mapsto v_3] \vdash e \Downarrow v_4}{\rho \vdash \textbf{let}\ x_1\ \textbf{=}\ x_2\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \Downarrow v_4}\ (\textsc{let-var}) \qquad \frac{n_2 \text{ is input from port } n_1 \quad \rho[x \mapsto n_2] \vdash e \Downarrow v}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ \textbf{getint}\ n_1\ \textbf{in}\ e \Downarrow v}\ (\textsc{getint})$$

$$\frac{\vec{v}_1 = \rho(\overrightarrow{\mathit{arg}}) \quad v_2 = \mathtt{applyPrim}(\oplus, \vec{v}_1) \quad \rho[x \mapsto v_2] \vdash e \Downarrow v_3}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ \oplus\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \Downarrow v_3}\ (\textsc{let-prim}) \qquad \frac{n_2 = \rho(\mathit{arg}) \quad \rho[x \mapsto n_2] \vdash e \Downarrow v}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ \textbf{putint}\ n_1\ \mathit{arg}\ \textbf{in}\ e \Downarrow v}\ (\textsc{putint})$$

$$\frac{(\mathit{cn}, \vec{v}_1) = \rho(\mathit{arg}) \quad (\mathit{cn}\ \vec{x} \Rightarrow e_1) \in \overrightarrow{\mathit{br}} \quad \rho[\vec{x} \mapsto \vec{v}_1] \vdash e_1 \Downarrow v_2}{\rho \vdash \textbf{case}\ \mathit{arg}\ \textbf{of}\ \overrightarrow{\mathit{br}}\ \textbf{else}\ e_2 \Downarrow v_2}\ (\textsc{case-con}) \qquad \frac{n = \rho(\mathit{arg}) \quad (n \Rightarrow e_1) \in \overrightarrow{\mathit{br}} \quad \rho \vdash e_1 \Downarrow v}{\rho \vdash \textbf{case}\ \mathit{arg}\ \textbf{of}\ \overrightarrow{\mathit{br}}\ \textbf{else}\ e_2 \Downarrow v}\ (\textsc{case-lit})$$

$$\frac{(\mathit{cn}, \overrightarrow{v_1}) = \rho(\mathit{arg}) \quad (\mathit{cn}\ \vec{x} \Rightarrow e_1) \notin \overrightarrow{\mathit{br}} \quad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash \textbf{case}\ \mathit{arg}\ \textbf{of}\ \overrightarrow{\mathit{br}}\ \textbf{else}\ e_2 \Downarrow v_2}\ (\textsc{case-else}1) \qquad \frac{n = \rho(\mathit{arg}) \quad (n \Rightarrow e_1) \notin \overrightarrow{\mathit{br}} \quad \rho \vdash e_2 \Downarrow v_1}{\rho \vdash \textbf{case}\ \mathit{arg}\ \textbf{of}\ \overrightarrow{\mathit{br}}\ \textbf{else}\ e_2 \Downarrow v_1}\ (\textsc{case-else}2)$$

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

$$\mathtt{applyCn}(\mathit{cn}\ ,\vec{v}\ ) = \begin{cases} (\mathit{cn}, \vec{v}) & \text{if } (\textbf{con}\ \mathit{cn}\ \vec{x}) \in \overrightarrow{\mathit{decl}} \text{ and } |\vec{v}| = |\vec{x}| \\ (\lambda\vec{x}.\textbf{let}\ c\ \textbf{=}\ \mathit{cn}\ \vec{x}\ \textbf{in}\ \textbf{result}\ c, \vec{v}) & \text{if } (\textbf{con}\ \mathit{cn}\ \vec{x}) \in \overrightarrow{\mathit{decl}} \text{ and } |\vec{v}| < |\vec{x}| \end{cases} \quad \Bigg| \quad \rho(\mathit{arg}) = \begin{cases} n & \text{if } \mathit{arg} = n \\ v & \text{if } \mathit{arg} = x \text{ and } (x \mapsto v) \in \rho \end{cases}$$

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

Figure 3. Big-Step Semantics for Zarf's functional ISA. Our semantics is a ternary relation on an environment (a mapping from variables to values); a let, case, or return expression; and the value to which this expression evaluates. Program evaluation begins with the main function's body expression. `applyFn` (`applyCn`) performs function (constructor) application. Applying a helper function which accepts one argument to a list of arguments is shorthand for mapping that helper function over the list. $\vec{x} :+ y$ appends y to the end of $\vec{x}$, creating a new list. $\rho[x \mapsto v]$ returns an updated copy of the environment with x mapped to v. $|\vec{x}|$ 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. Although it appears that syntax is being created dynamically during the second cases of `applyCn` and `applyPrim`, we can treat each constructor or primitive operator as having a predefined body that evaluates to that constructor or primitive application, respectively, such that this transformation only occurs once and is thus static. **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.

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.

The λ-execution layer 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.

Figure 3 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; Figure 4 shows how the assembly maps one-to-one with the binary encoding, and Figure 6 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, which are sequential numbers that begin at 0 for each function. The first word in the `let` instruction indicates a function identifier or closure object and the number of argument words that follow. Each argument word consists of a source and an index, indicating where the argument value should be pulled from and which value to pull. 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.

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[2]. The first word of the instruction is followed by additional words encoding patterns against which to match the argument. A `pattern_literal` argument contains both an integer value to match against and a number of words n to skip if the match fails. If the case value exactly equals the literal value in the instruction word, then execution continues with the next instruction. If not equal, n instruction words are skipped, which brings execution to the next branch of the case. The `pattern_cons` takes a similar argument, but the integer value indicates a function identifier to match against. The match succeeds if and only if 1) the case instruction was attempting to match a constructor, not an integer, and 2) the constructor is the same as specified in the `pattern_cons`. Skips are handled in the same way. Finally, 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.

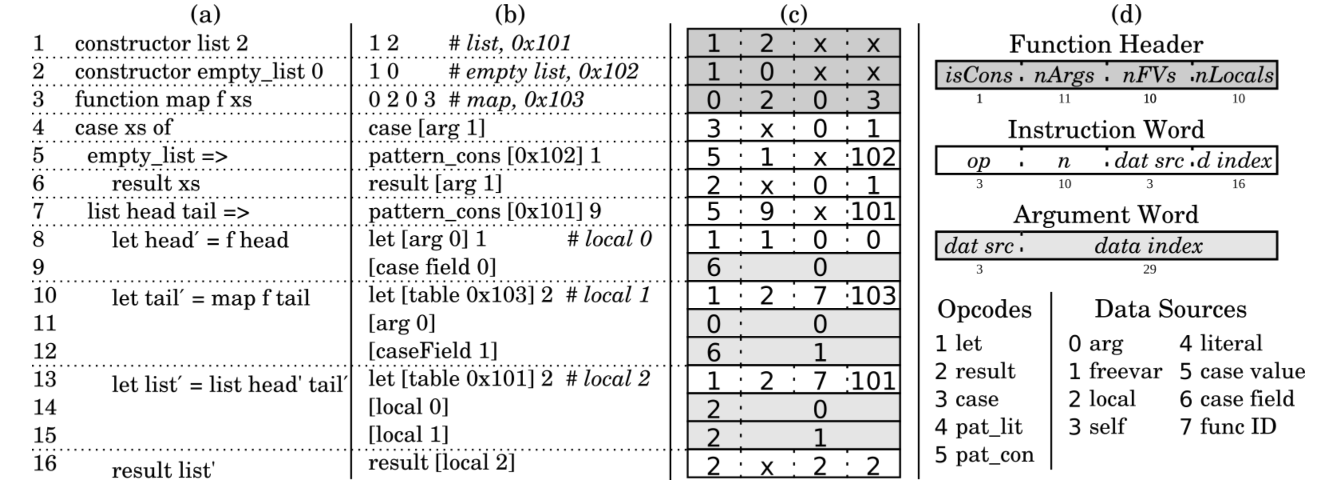
The `result` instructions are a single word, indicating a single piece of data that the current function should yield. Every branch of every function must terminate with a result instruction (disallowing re-convergent branches means the simple pattern-skip mechanism is all that is necessary for control 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 Figure 4 to help ground our descriptions in a concrete example. Figure 4 shows a small function, `map`, written in high-level assembly, machine assembly, and encoded as a binary.

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

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



	(a)	(b)	(c)
1	constructor list 2	1 2 # *list, 0x101*	1 : 2 : x : x
2	constructor empty_list 0	1 0 # *empty list, 0x102*	1 : 0 : x : x
3	function map f xs	0 2 0 3 # *map, 0x103*	0 : 2 : 0 : 3
4	case xs of	case [arg 1]	3 : x : 0 : 1
5	empty_list =>	pattern_cons [0x102] 1	5 : 1 : x :102
6	result xs	result [arg 1]	2 : x : 0 : 1
7	list head tail =>	pattern_cons [0x101] 9	5 : 9 : x :101
8	let head′ = f head	let [arg 0] 1 # *local 0*	1 : 1 : 0 : 0
9		[case field 0]	6 : 0
10	let tail′ = map f tail	let [table 0x103] 2 # *local 1*	1 : 2 : 7 :103
11		[arg 0]	0 : 0
12		[caseField 1]	6 : 1
13	let list′ = list head' tail′	let [table 0x101] 2 # *local 2*	1 : 2 : 7 :101
14		[local 0]	2 : 0
15		[local 1]	2 : 1
16	result list'	result [local 2]	2 : x : 2 : 2

(d)

Opcodes	Data Sources	
1 let	0 arg	4 literal
2 result	1 freevar	5 case value
3 case	2 local	6 case field
4 pat_lit	3 self	7 func ID
5 pat_con		

Figure 4. How the high-level assembly instructions are directly compiled into a Zarf binary for λ-execution layer 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.

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 environment by introducing artificial data dependencies; this is the principle underlying the I/O monad [48; 32], 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 [31; 47].

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 the λ-execution layer represents a trade-off of increased abstraction and verification capability for additional development effort and some decrease in performance. Section 6 provides some quantitative worst-case bounds for this trade-off.

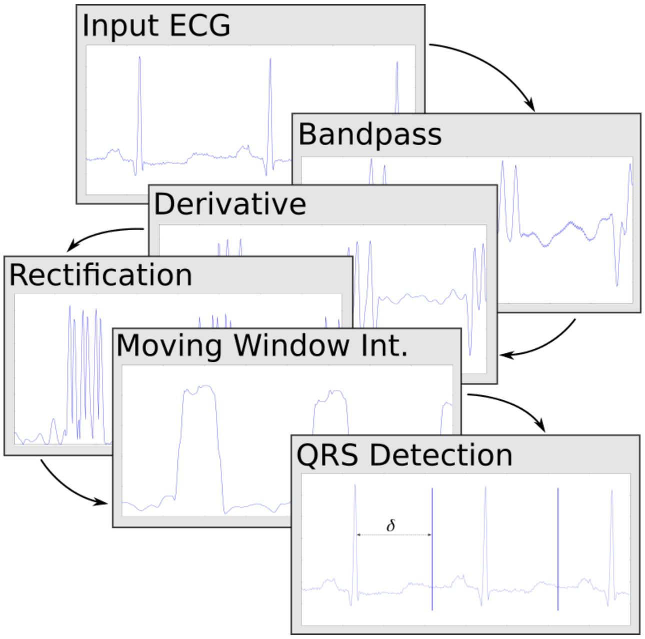


Figure 5. The ECG takes input signals sampled at 200 Hz and filters them multiple times, after which the peaks are classified and the rate of heartbeat determined. These values are fed to an ATP (antitachycardia pacing) procedure, which decides if ventricular tachycardia is occurring based on current and previous heart rate, and administers pacing shocks to prevent acceleration and ventricular fibrillation (a form of cardiac arrest).

The λ-execution layer runs a small microkernel based on cooperative coroutines [16; 50] 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 5.2) ensures each coroutine always returns control.

The λ-execution layer enables reasoning about these coroutines at the assembly and binary level. Section 5 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 the λ-execution layer. 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 the λ-execution layer.

In our sample application, three application coroutines are run on the λ-execution layer: 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 arrhthymias (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 [44]. Between 2001 and 2015, over 150,000 implanted medical devices were recalled by the FDA because of life-threatening software bugs [59]. However, ICDs are credited with saving thousands of lives; for patients who have survived life-threatening arrhthymia, ICDs decrease mortality rates by 20-30% over medication [15; 65; 60]. Currently, around 10,000 new patients have an ICD implanted each month [3], and around 800,000 people are living with ICDs [2].

The core of our ICD is an embedded, real-time ECG algorithm that performs QRS[3] detection on raw electrocardiogram data to determine the timing between heartbeats. We work off of an established real-time QRS detection algorithm [54], which has seen wide use and been the subject of studies examining its performance and efficacy [18]. An

[3] 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.

open-source update of several versions of the algorithm [4] 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 (Figure 5 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 5.2 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. 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.

5.1 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 [1], using this version as our specification of functional correctness. This specification operates on streams — a data type 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.

```
                          (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' => ...
```

Figure 6. 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 the Zarf λ-execution layer.

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 Figure 6). 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 the Zarf λ-execution layer is faithful to the high-level Coq specification. This proof of equivalence between the two Coq implementations is done by induction 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 λ-execution layer assembly keywords.

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 λ-execution layer 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.

5.2 Timing

With a knowledge of how the λ-execution layer 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 6 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.

5.3 Non-Interference

Because the ICD coroutine has been proven correct (Section 5.1), 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 ℓ in the security lattice $\mathcal{L}$" [35]. In a standard security lattice, $\mathbf{L}$ (low-security) $\sqsubseteq$ $\mathbf{H}$ (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, $\mathbf{T}$ (trusted) and $\mathbf{U}$ (untrusted), organized such that $\mathbf{T} \sqsubseteq \mathbf{U}$. Therefore, our integrity non-interference property is that untrusted values cannot affect trusted values [58].

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

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

Specifically, following the spirit of Abadi et al. [6] and Simonet [61], 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. [66]. We show

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

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

that if an expression e has some specific type τ 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 casewise over the three types of expressions in our language, combining our evaluation semantics with our security typing rules.

6. Evaluation

To validate our designs, we download the λ-execution layer 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 the λ-execution layer 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 130nm, the combinational logic takes up .274 mm^2. Though larger than very simple embedded CPUs, the λ-execution layer 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 5.2 discusses the worst-case runtime of the λ-execution layer 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.

7. 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 the λ-execution layer 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 the λ-execution layer.

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.

Acknowledgments

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

We'd like to thank Nicholas Brown, Benjamin Campbell, Tristan Konologie, Bingbin 'Clara' Liu, and Benjamin Spitz for their work on related system infrastructure; Kyle Dewey, for his system and typesystem critiques; and the anonymous reviewers, for their invaluable feedback.

References

[1] The Coq proof assistant: https://coq.inria.fr.

[2] How many people have ICDs? http://asktheicd.com/tile/106/english-implantable-cardioverter-defibrillator-icd/how-many-people-have-icds/.

[3] 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.

[4] Open source ECG analysis software. http://www.eplimited.com/confirmation.htm.

[5] *Journal of Automated Reasoning*, 30(3-4), 2003.

[6] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160, New York, NY, USA, 1999. ACM.

[7] A. W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, pages 247–, Washington, DC, USA, 2001. IEEE Computer Society.

[8] 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, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and 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, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.

[10] R. S. Boyer and 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, pages 416–430, London, UK, UK, 1992. Springer-Verlag.

[11] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre. Quantitative verification of implantable cardiac pacemakers. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 263–272. IEEE, 2012.

[12] A. Chlipala. A verified compiler for an impure functional language. In *ACM Sigplan Notices*, volume 45, pages 93–106. ACM, 2010.

[13] 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, pages 234–245, New York, NY, USA, 2011. ACM.

[14] T. J. Clarke, P. J. Gladstone, C. D. MacLean, and A. C. Norman. Skim - the s, k, i reduction machine. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 128–135, New York, NY, USA, 1980. ACM.

[15] S. J. Connolly, M. Gent, R. S. Roberts, P. Dorian, D. Roy, R. S. Sheldon, L. B. Mitchell, M. S. Green, G. J. Klein, and B. O'Brien. Canadian implantable defibrillator study (cids). *Circulation*, 101(11):1297–1302, 2000.

[16] M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963.

[17] L. C. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In *ICESS*, 2009.

[18] M. M. Cruz-Cunha, J. Varaj£o, H. Krcmar, R. Martinho, R. A. lvarez, A. J. M. Penn, and X. A. V. Sobrino. 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 a comparison of three qrs detection algorithms over a public database. *Procedia Technology*, 9:1159 – 1165, 2013.

[19] P. Curzon and P. Curzon. A verified compiler for a structured assembly language. In *In proceedings of the 1991 international workshop on the HOL theorem Proving System and its applications. IEEE Computer*, 1991.

[20] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[21] T. Denning, K. Fu, and T. Kohno. Absence makes the heart grow fonder: New directions for implantable medical device security. In *HotSec*, 2008.

[22] L. P. Deutsch. A lisp machine with very compact programs. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 697–703. Morgan Kaufmann Publishers Inc., 1973.

[23] A. Fox and 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, pages 243–258, Berlin, Heidelberg, 2010. Springer-Verlag.

[24] J. A. Goguen and J. Meseguer. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on*, pages 11–11, April 1982.

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

[26] A. O. Gomes and M. V. M. Oliveira. *Formal Specification of a Cardiac Pacing System*, pages 692–707. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[27] B. Graham. Secd: Design issues. Technical report, University of Calgary, 1989.

[28] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and 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)*, pages 129–142. IEEE, 2008.

[29] B. Hardekopf and 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, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society.

[30] N. Heintze and 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, pages 365–377, New York, NY, USA, 1998. ACM.

[31] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[32] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[33] W. A. Hunt Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[34] Z. Jiang, M. Pajic, and R. Mangharam. Cyber-physical modeling of implantable cardiac medical devices. *Proceedings of the IEEE*, 100(1):122–137, Jan 2012.

[35] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *2011 IEEE Symposium on Security and Privacy*, pages 413–428, May 2011.

[36] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: Virtualized cloud infrastructure without the virtualization. *SIGARCH Comput. Archit. News*, 38(3):350–361, June 2010.

[37] A. Kennedy, N. Benton, J. B. Jensen, and 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, pages 13–24, New York, NY, USA, 2013. ACM.

[38] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[39] T. F. Knight. *Implementation of a list processing machine*. PhD thesis, Massachusetts Institute of Technology, 1979.

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

[41] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, Jan. 1964.

[42] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[43] T. Maeda and 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, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

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

[45] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev.*, 42(4):315–328, Apr. 2008.

[46] N. G. Michael and A. W. Appel. Machine instruction syntax and semantics in higher order logic. In *Proceedings of the 17th International Conference on Automated Deduction*, CADE-17, pages 7–24, London, UK, UK, 2000. Springer-Verlag.

[47] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[48] E. Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[49] J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[50] A. L. D. Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):6:1–6:31, Feb. 2009.

[51] G. C. Necula. Translation validation for an optimizing compiler. In *ACM sigplan notices*, volume 35, pages 83–94. ACM, 2000.

[52] G. C. Necula. *Proof-carrying code. design and implementation*. Springer, 2002.

[53] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and 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, pages 166–178, New York, NY, USA, 2015. ACM.

[54] J. Pan and W. J. Tompkins. A real-time qrs detection algorithm. *IEEE Transactions on Biomedical Engineering*, BME-32(3):230–236, March 1985.

[55] F. Pottier and V. Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, Jan. 2003.

[56] T. Ramananandro, Z. Shao, S.-C. Weng, J. Koenig, and Y. Fu. A compositional semantics for verified separate compilation and linking. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 3–14, New York, NY, USA, 2015. ACM.

[57] 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*, pages 352–367, London, UK, UK, 1982. Springer-Verlag.

[58] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, Sept. 2006.

[59] S. Shuja, S. K. Srinivasan, S. Jabeen, and D. Nawarathna. A formal verification methodology for ddd mode pacemaker control programs. *Journal of Electrical and Computer Engineering*, 2015.

[60] J. Siebels, K.-H. Kuck, and C. Investigators. Implantable cardioverter defibrillator compared with antiarrhythmic drug treatment in cardiac arrest survivors (the cardiac arrest study hamburg). *American Heart Journal*, 127:1139–1144, April 1994.

[61] 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, pages 223–, Washington, DC, USA, 2002. IEEE Computer Society.

[62] M. Strecker. Formal verification of a java compiler in isabelle. In *Automated DeductionCADE-18*, pages 63–77. Springer, 2002.

[63] D. Terei, S. Marlow, S. Peyton Jones, and D. Mazières. Safe haskell. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 137–148, New York, NY, USA, 2012. ACM.

[64] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editor, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, jan 2007. ACM.

[65] T. A. versus Implantable Defibrillators (AVID) Investigators. A comparison of antiarrhythmic-drug therapy with implantable defibrillators in patients resuscitated from near-fatal ventricular arrhythmias. *New England Journal of Medicine*, 337(22):1576–1584, 1997. PMID: 9411221.

[66] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.

[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, and 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):2591–2596, 2004.

[68] H. Xi and R. Harper. A dependently typed assembly language. In *ACM SIGPLAN Notices*, volume 36, pages 169–180. ACM, 2001.

[69] J. Yang and 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, pages 99–110, New York, NY, USA, 2010. ACM.

[70] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for pcc: dynamic storage allocation. In *Proceedings of the 12th European conference on Programming*, pages 363–379. Springer-Verlag, 2003.