An Architecture for Analysis

Joseph McMahan, Michael Christensen, and Lawton Nichols
University of California, Santa Barbara

Jared Roesch
University of Washington

Sung-Yee Guo
iRobot

Ben Hardekopf and Timothy Sherwood
University of California, Santa Barbara

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 and specification languages. This approach allows assembly-level verified versions of critical code to operate safely in tandem with arbitrary code without the need for large supporting trusted computing bases.

Computer architecture has a role to play in every computing device, from the largest datacenter to the smallest embedded sensor. While many are thinking about huge machine-learning farms and ultra-low-power approximate computing, it is important not to forget that even our most life-critical systems need architectures to get their computation done. The "hard" part of deploying such critical systems is not necessarily getting working code, or even achieving high performance, but rather convincing oneself that the program one developed is actually correct and secure. We explore a new approach to architecture, one where this act of analysis is elevated to a first-class design constraint, to see if and how the machines can be more easily and more completely analyzed.

When the ability to reason about and verify low-level life-critical software is paramount, it leads one to reconsider the role of the instruction set architecture (ISA). The machine we introduce, Zarf, takes a leap towards ease of analysis by executing code in a manner closely resembling the underlying computational model on which proof and reasoning systems are already built and specifications are already written. 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 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 for reasoning about behavior at the binary level.

Published by the IEEE Computer Society
0272-1732/18/$33.00 ©2018 IEEE

Application behaviors that are mission-critical can be hoisted piecemeal from the imperative to the functional world as needed.

To demonstrate the usefulness of this platform, we have developed, modeled, and tested an implantable cardio defibrillator (ICD)—an embedded medical device that 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.[1,2] We were able to formally verify the correctness of a low-level implementation of the core functions in the Coq Theorem Prover 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 critical functions. Furthermore, the functional abstraction built into 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, real-time deadlines are assured to be met, and C code can execute arbitrary auxiliary functions in parallel for monitoring. The high-level system architecture of the platform is shown in Figure 1.

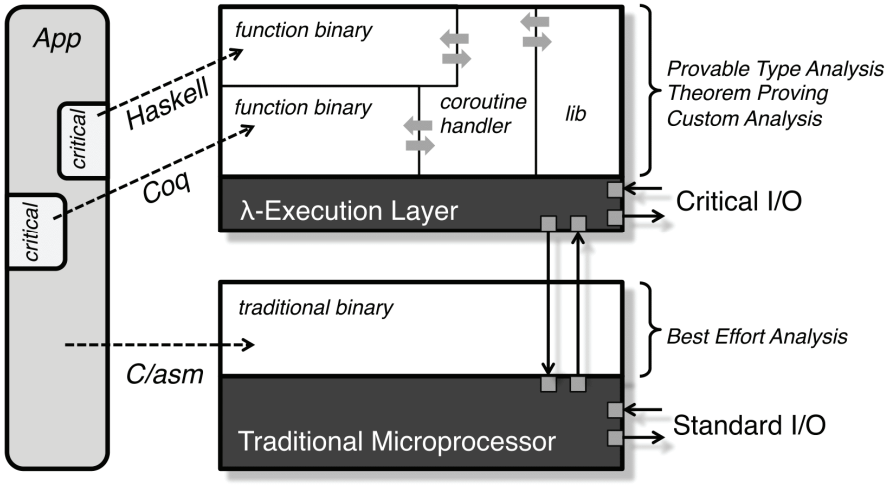

Figure 1. High-level system architecture. By pairing an imperative core with a Zarf processor, we can run legacy software and drivers while raising critical portions of software to a region where they can be reasoned about compositionally.

ISA

An ISA built for analysis must look fundamentally different from the "instruction-by-instruction" model of traditional machines. Global machine state, mutable state, large numbers of instructions and features, arbitrary control flow, unenforced function call conventions, and implicit instruction semantics all thoroughly complicate the process of modeling and reasoning about existing ISAs. 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. Instead of imperative instructions acting as the building blocks of a program, our basic unit is the function. 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, along with a precise and formal semantics of the ISA, allows us to efficiently reason about programs at a low level.

Zarf's functional ISA is effectively a version of lambda calculus—specifically, it is an untyped, lambda-lifted, administrative normal form (ANF) lambda calculus. It is untyped to avoid the

complexity of a hardware type checker; it is lambda-lifted so that each function can be assigned a global address in memory; and finally, it is in ANF because nested sub-expressions cannot easily be encoded into a binary. One bit is used at runtime to distinguish the two forms of values in the system: integers and objects. This prevents malformed code from putting the machine an in invalid state at runtime. Instructions use De Bruijn indices to refer to data elements—this is akin to using the stack offset of each variable in the local frame; variables are put on a conceptual stack automatically, allowing an implicit and static indexing to be determined. Because references are localized and cannot refer to any global state, together with immutability, this enforces referential transparency.

Only three instructions are used to define the bodies of functions: let, case, and result. Unlike RISC instructions, let and case can be multiple words long, depending on the number of arguments and branches used (respectively); however, unlike most CISC instructions, each piece of the variable length instructions is word-aligned and trivial to decode. We describe the operation of each instruction below.

Because Zarf has no programmer-visible registers or memory addresses, instructions need some way to refer to data elements in the program. This is accomplished by using pairs of source and index values, where source is from a predefined set—such as local or arg, which serve purposes similar to stack offsets in a traditional machine—and the index specifies which value from the set to use. The actual addresses themselves are never visible, but one can still uniquely refer to elements that reside on the stack or in the heap of the machine.

Values in the system take the form of integers, closures, and constructors. Closures are unevaluated function calls, as discussed below. Constructors are stub functions that just hold data; formally, they can be used to make algebraic data structures. More informally, they are akin to structs in C.

Figure 2 shows how function and constructor declarations, and the three high-level instructions, end up mapping to binary programs. The assembly process is quite direct, as the figure illustrates: Variable-length instructions are broken up by word, and then each word is encoded in binary.

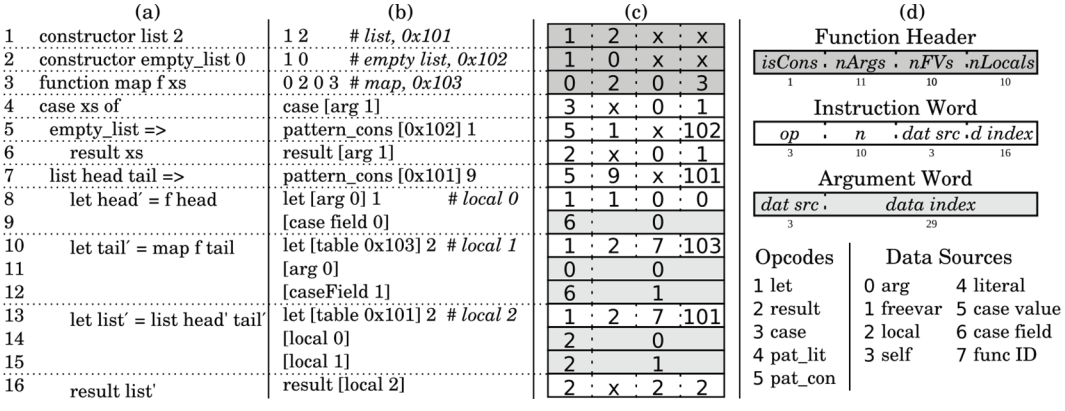

Figure 2. How high-level assembly instructions are compiled directly into a binary for the Zarf processor.

Instructions: Let, Case, and Result

Let

The let instruction is used for both object allocation and function application. Because our machine is a platform where one defines and uses functions, let operations are the bread-and-butter of any program, allowing one to apply arguments to functions defined in software or hardware.

Each let instruction is assigned an implicit local identifier, which is a sequential number that begins at 0 for each function. The first word in the let instruction indicates what function identifier (or closure object) is being used, and the number of arguments being applied. Then, that number of argument words follow, each one consisting of a source and an index, describing where the argument should be pulled from and what value to pull.

Unlike a traditional "call" instruction, let does not immediately change the control flow to force evaluation of the function call. Instead, it creates a new structure in memory—a closure—that ties a function identifier to the arguments. This object represents the unevaluated result of the function call. When finally needed, it can be evaluated, providing a final result of the call. This sort of evaluation semantics is known as "lazy evaluation."

Additionally, let instructions can be used to dynamically create new functions "on the fly." This can be accomplished through partial application—applying some, but not all, of a function's arguments. This creates a closure that still expects additional arguments, but already has some data values applied. For example, using the primitive "add" function, which expects two integers, one can execute "let f = add 1," which will create a new local variable (called "f" in the high-level code) for the closure, tying the add function to the integer value 1. Then, this closure can be used as a function any number of times. It expects one argument and adds 1 to that argument. This sort of behavior is called "currying" and is supported directly in the hardware to allow for expressive binary programs.

Case

The case instruction provides a mechanism for control flow, accomplished through pattern-matching. It takes a value (such as an argument or local identifier) and makes a set of equality comparisons, one for each "pattern" provided. When a matching pattern is found, that branch of the code is executed. Because these comparisons require an actual value, and not an unevaluated closure, this is the point in execution where evaluation takes place, reducing the structures created with let instructions down to values. Evaluation will be performed until the function returns either an integer or constructor. Reducing to this point allows the machine to have a value with which comparisons can be made.

The first word of the case, specifying the value to evaluate and match, is followed by a series of branches. Each branch contains a special branch head instruction that encodes something to match against. This can take the form of integer values, in the case that the function call reduced to an integer, or constructor identifiers, in the case that a constructor was found. The branch head also indicates how many words are in that particular branch, so on a failure, the machine knows how many instructions to skip to find the next branch head.

A pattern_literal instruction is used to match against a literal value. The match succeeds if and only if evaluation resulted in an integer and the value of the integer is equal to the value in the branch head. On a match, execution continues with the next instruction; on a failed match, execution skips to the next branch head. A pattern_cons instruction is used for matching constructors; here, the integer value encoded represents the function ID of the desired constructor to match against. The match succeeds if and only if evaluation resulted in a constructor object, not an integer, and the constructor ID exactly matches the instruction's. Finally, a pattern_else instruction is required in every case statement in the event that no matching value or constructor is found. Else branches are always taken and demarcate the end of the case instruction. Case/pattern sequences not adhering to the encoding described are malformed and invalid (for example, you cannot skip to the middle of a branch or have a case without an else branch). These properties can easily be checked in a single pass over the binary.

For example, if we are writing a function that operates on linked lists, we have two possible constructors: the list constructor and the empty list constructor. Writing a function that recursively follows the linked list to the end, we case on the current value and match against the two constructor varieties. If a list constructor was found, we recursively continue; if the end of the list is found, we have our base case.

Result

The result instruction is quite simple, taking up only a single word. It supplies a source and an index, indicating what data value should be returned from the current function. Every branch of every function must terminate with a result instruction. This significantly simplifies control-flow by disallowing re-convergent code; our simple pattern-skip mechanism is all that is required for control-flow, and checking that control-flow is well-formed is trivial and can be ensured with a single pass over a binary program. After a result instruction, either control flow passes to the case instruction where the function result was required (if an integer or constructor is being returned) or evaluation continues (if a closure object is being returned).

ICD SOFTWARE

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 who are at risk of 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 (VT), and administer pacing shocks, or anti-tachycardia pacing (ATP). These shocks help prevent the acceleration in heart rate leading to ventricular fibrillation, a form of cardiac arrest.

Unfortunately, devices such as these can be subject to dangerous software bugs. From 1990 to 2000, more than 200,000 ICDs and pacemakers were recalled due to software issues.[1] Between 2001 and 2015, more than 150,000 implanted medical devices were recalled by the US Food and Drug Administration (FDA) because of life-threatening software bugs.[2] However, the usefulness of the devices is clear; ICDs are credited with saving thousands of lives. For patients who have survived life-threatening arrhythmias, ICDs decrease mortality rates by 20 to 30 percent over medication.[3,4,5] Currently, around 10,000 new patients have an ICD implanted each month,[6] and around 800,000 people are living with ICDs.[7] This is a huge number of critical, life-saving devices that might contain catastrophic bugs.

To demonstrate the usefulness of the Zarf platform and to show that low-level analysis is, in fact, simplified, we implement a custom version of an ICD. The core of our ICD is an embedded, real-time electrocardiography (ECG) algorithm that perform QRS detection on raw electrocardiogram data to determine the timing between heartbeats. (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.) We work off of an established real-time QRS detection algorithm,[8] which has seen wide use and been the subject of studies examining its performance and efficacy.[9] An open-source update of several versions of the algorithm[10] is available; we use the results of this open-source work as the basis of our algorithm's specification. After the ECG algorithm detects the pacing between heartbeats, the ATP function checks for signs of VT and, if found, administers a series of pacing shocks. We implement a published VT test and ATP treatment procedure.[11]

Our software is structured as a microkernel that schedules and coordinates three cooperating coroutines. These consist of the core ICD coroutine, an I/O coroutine that handles both reading raw values from the heart and outputting pacing signals when requested, and a diagnostic coroutine that collects information as the system runs.

The I/O coroutine reads an input from the heart at a fixed frequency of 200 Hz. It passes this value to the microkernel, which then schedules the ICD coroutine, communicating the input value. The core ICD algorithm then takes place, consisting of a series of filter passes to detect the spacing between QRS complexes in the patient's heartbeat. These filter passes and the high-level software architecture are illustrated in Figure 3. Then, a screening algorithm examines a running window of the last 24 heartbeats; if 18 or more of them had periods of less than 360 ms, corresponding to a heart rate greater than 167 beats per minute, the ICD coroutine transitions to the treatment state. When in this state, it outputs a series of three sequences of eight pulses at 88 percent of the current heart rate, with a 20-ms decrement between sequences. This is designed to prevent continued acceleration and restore a safe heart rhythm.

These output values are fed to the microkernel, which passes them to the I/O coroutine when next it is scheduled. When requested, the I/O coroutine signals that a shock should be administered.

The microkernel, I/O coroutine, and ICD coroutine all run on the Zarf platform; the diagnostic coroutine runs on the imperative core, a soft-core Xilinx MicroBlaze. The diagnostic routine is fed the output of the ICD and catalogs the number of times treatment has occurred. This information is output when requested.

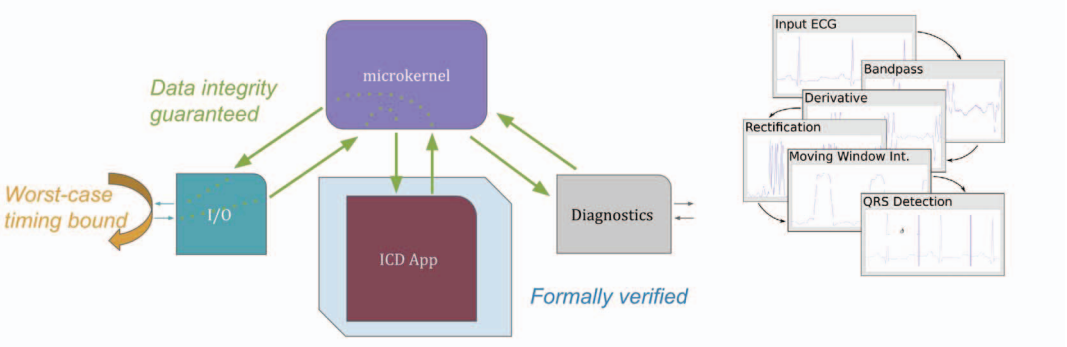

Figure 3. ICD application software architecture and summary of ECG algorithm. We perform three forms of analysis on the system at the assembly level, verifying correctness, timing, and data integrity properties.

BINARY ANALYSIS

The goal of our system is to create a platform where reasoning and analysis at the binary level is simplified. Analysis can take many forms, applied to many parts of the system. To demonstrate the ability to reason on the Zarf platform, we perform three different forms of low-level analysis on our ICD software, proving the correctness of the core ICD functionality, ensuring real-time timing deadlines are always met with a timing analysis, and showing that critical values in the system are never corrupted with an integrity type system.

Correctness

The first step in trying to assure the correctness of a system is to find or write a specification. For this, we implement a high-level version of the ICD's critical algorithms: the ECG filters and the ATP treatment procedure. These were written in Gallina, the specification language of the Coq Theorem Prover. The specification takes advantage of high-level language features, such as streams, allowing it to remain abstract and relatively simple. This allows us to be more confident that we have specified the algorithms correctly. (See Figure 4.)

Next, we refine the specification into an implementation layer, also written in Gallina. This version is more low-level than the specification; it operates on machine values rather than streams, isolates function applications to let expressions, and avoids use of the if-then-else construct (replacing it with equivalent logic and function calls). We formally prove, using Coq, that this version is functionally equivalent to the high-level spec.

From this layer, we take advantage of the narrow semantic gap from the formal specification language to our machine semantics to directly extract an executable version of the low-level specification. We do not need any expensive runtimes, complex translations and lowerings to imperative operations, or any compilation at all really; we simply replace the Coq syntax with the Zarf assembly syntax. The lowering from high-level spec to Zarf machine code is shown in Figure 4. This line-for-line extraction from specification to executable allows us to more or less directly run a version of the software that has been proven to be correct.

```
                    (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 4. Extraction of verified application component from high-level formal specifications.

Timing

Even though the Zarf architecture is very high-level, abstracting away memory and providing a clean functional abstraction, we are still able to bound worst-case execution of programs on the platform. This is accomplished in the standard way, by looking at the timing of each instruction, bounding worst-case time for each function, and composing those values into a program bound.

In addition, we have to bound the time for the garbage collector to run. We implement a semi-space trace-collector, which uses bump-pointer allocation and a relatively simple copy-collect algorithm for collection. We bound the worst-case collection time as a function of the memory usage, because collection is proportional to the live set of objects at collection time. Without making approximations of when different objects go out of scope and become dead, we use the ultra-conservative approach of assuming that all objects are still live at collection time, causing the worst possible runtime for the garbage collector. Using that, we can get a direct bound by simply counting up the number and size of let instructions in each function.

From the static analysis, we determine that the worst execution of the entire loop is 9,065 cycles to run one iteration of system, including garbage collection. This is 181.3 µs on our FPGA synthesized prototype running at 50 MHz, falling well within the real-time deadline of 5 ms.

Non-Interference

We have a nice, formal guarantee of the correctness of our ICD algorithms, but the remainder of the software in the system is potentially untrusted. To deal with this mismatch, we perform an analysis using an integrity type system to guarantee that critical values are never corrupted by any part of the software.

To prove this about Zarf, 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 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 prove soundness of the type system, or that if a program type-checks, it is guaranteed to preserve data integrity according to its security labels. For our program, the input and output of the ICD coroutine are considered critical, trusted values. Aside from the ICD coroutine itself, all other parts of the system are assumed to be untrusted. Once these type labels are applied, we perform type-checking to guarantee that those critical values are never corrupted by any part of the system.

EVALUATION

The hardware description of Zarf is more complex than a simple embedded CPU, with 66 total states of control logic (four 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 look-up tables (LUTs) when synthesized for an Artix-7 FPGA (less than 7 percent of the available logic resources). Estimated on 130 nm, the combinational logic takes up 0.274 mm^2, making Zarf quite a bit smaller than many common embedded microcontrollers.

Though potentially larger in hardware usage and slower than a traditional microcontroller, the Zarf ICD application is still able to operate more than 25X faster than necessary to meet its critical real-time deadlines. Invaluably, we are able to add guarantees about the correctness of the most critical application components, as well as assurance of non-interference between separate functions in the system, all performed at the assembly and binary level. This adds a new and interesting design point to the spectrum of embedded systems, where correctness and analysis are first-class citizens, but a universal system with practical performance is still possible.

CONCLUSION

As computing continues to automate and improve the control of life-critical systems, new techniques that 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 runtimes and compilers in the trusted code base. We take a holistic approach to the evaluation of this idea, not only demonstrating its practicality through an FPGA-implemented prototype but also showing the successful application of three forms of static analysis at the assembly level.

As we move to increasingly diverse SoCs, 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 percent 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 of the design process, rather than treated as an afterthought.

ACKNOWLEDGMENTS

This material is based on work supported by the NSF under grants 1740352, 1730309, 1717779, 1563935, 1444481, and 1341058, as well as a gift from Cisco Systems.

REFERENCES

1. R. Mangharam et al., “Three Challenges in Cyber-Physical Systems,” *8th International Conference on Communication Systems and Networks* (COMSNETS), 2016.
2. S. Shuja et al., “A Formal Verification Methodology for DDD Mode Pacemaker Control Programs,” *Journal of Electrical and Computer Engineering*, 2015.
3. S. Connolly et al., “Canadian Implantable Defibrillator Study (CIDS),” *American Heart Association Journals*, 2000, pp. 1297–1302.

4. "A Comparison of Antiarrhythmic-Drug Therapy with Implantable Defibrillators in Patients Resuscitated from Near-Fatal Ventricular Arrhythmias," *New England Journal of Medicine*, 1997, pp. 1576–1584.
5. J. Siebels and K. Kuck, "Implantable Cardioverter Defibrillator Compared with Antiarrhythmic Drug Treatment in Cardiac Arrest Survivors (the Cardiac Arrest Study Hamburg)," *American Heart Journal*, vol. 127, no. 4, 1994, pp. 1139–1144.
6. "Living With Your Implantable Cardioverter Defibrillator (ICD)"; www.heart.org/HEARTORG/Conditions/Arrhythmia/PreventionTreatmentofArrhythmia/Living-With-Your-Implantable-Cardioverter-Defibrillator-ICD_UCM_448462_Article.jsp.
7. "How Many People Have ICDs?"; http://asktheicd.com/tile/106/english-implantable-cardioverter-defibrillator-icd/how-many-people-have-icds/.
8. J. Pan and W. Tompkins, "A Real-Time QRS Detection Algorithm," *IEEE Transactions on Biomedical Engineering*, vol. BME-32, 1985, pp. 230–236.
9. M. Cruz-Cunha et al., "A Comparison of Three QRS Detection Algorithms Over a Public Database," *Procedia Technology*, vol. 9, 2013, pp. 1159–1165.
10. "Open-Source ECG Analysis Software"; www.eplimited.com/confirmation.htm.
11. M. Wathen et al., "Prospective Randomized Multicenter Trial of Empirical Antitachycardia Pacing Versus Shocks for Spontaneous Rapid Ventricular Tachycardia in Patients With Implantable Cardioverter Defibrillators," *American Heart Association Journals*, vol. 110, no. 17, 2004, pp. 2591–2596.

ABOUT THE AUTHORS

Joseph McMahan is a PhD student in computer architecture at the University of California, Santa Barbara. His research deals mainly with the intersection of architecture and formal methods, building hardware and analysis techniques to support correctness, and security of critical and embedded systems. He previously studied physics at Princeton University, and he is a student member of the ACM and IEEE. Contact him at jmcmahan@cs.ucsb.edu.

Michael Christensen is a PhD student in the Programming Languages Lab at the University of California, Santa Barbara. He is interested in the co-design of programming languages and safe low-level systems. Contact him at mchristensen@cs.ucsb.edu.

Lawton Nichols is a computer science PhD student at the University of California, Santa Barbara. His research interests include static program analysis and program verification. Contact him at lawtonnichols@cs.ucsb.edu.

Jared Roesch is a PhD student in the Paul G. Allen School of Computer Science & Engineering at the University of Washington. His research focuses on the application of techniques from programming languages and automated reasoning to domains including architecture, systems, security, and machine learning. Contact him at jroesch@cs.washington.edu.

Sung-Yee Guo is a robotics engineer at iRobot. Previously, he was a master's student at the University of California, Santa Barbara, working under Timothy Sherwood in the computer architecture lab. Contact him at sguo@umail.ucsb.edu.

Ben Hardekopf is a professor in the Department of Computer Science at the University of California, Santa Barbara. His research focuses on programming languages. Hardekopf has a PhD in computer science from the University of Texas at Austin. Contact him at benh@cs.ucsb.edu.

Timothy Sherwood is a professor in the Department of Computer Science at the University of California, Santa Barbara. His research focuses on the development of processors exploiting novel technologies, designed for provable properties, and/or implementing hardware-aware algorithms. Sherwood has a PhD in computer science from the University of California, San Diego. He is a senior member of the IEEE and an ACM Distinguished Scientist. Contact him at sherwood@cs.ucsb.edu.