# Improving the Trustworthiness of Medical Device Software with Formal Verification Methods

Chunxiao Li, Anand Raghunathan, and Niraj K. Jha

***Abstract*—Wearable and implantable medical devices are commonly used for diagnosing, monitoring, and treating various medical conditions. Increasingly complex software and wireless connectivity have enabled great improvements in the quality of care and convenience for users of such devices. However, an unfortunate side-effect of these trends has been the emergence of security concerns. In this letter, we propose the use of formal verification techniques to verify temporal safety properties and improve the trustworthiness of medical device software. We demonstrate how to bridge the gap between traditional formal verification and the needs of medical device software. We apply the proposed approach to cardiac pacemaker software and demonstrate its ability to detect a range of software vulnerabilities that compromise security and safety.**

***Index Terms*—Formal verification, medical device software, safety, security.**

## I. Introduction

A GENERAL trend in personal healthcare systems is towards increased functional complexity, software programmability, and (wireless) network connectivity. An undesirable, yet inevitable, side-effect of these trends is that medical devices are becoming increasingly vulnerable to security attacks. Among all the security challenges posed by medical devices, software-related issues are of utmost importance. More than a fourth of the recalls of defective medical devices during the first half of 2010 were likely caused by software defects [1]. Furthermore, because of the increased deployment of powerful programmable processors in medical devices, software-related recalls are rapidly increasing [2].

Although the trustworthiness of software on medical devices is literally a life-or-death issue, few methods exist to address the problem. Internal code review and system testing may solve the problem, but are not efficient enough for device manufacturers to thoroughly adopt them, as shown by recent numerous recalls. Open-source medical device software has been suggested as a model, which is unlikely to be adopted by manufacturers due to competitive considerations.

Software verification is a technique that takes a particular program implementation and some specifications as inputs, analyzes all possible program execution traces implicitly, and outputs the traces, if any, that violate the given specifications (e.g., array bounds, pointer safety, and user-specified assertions) [3]. While software verification techniques have been used to verify general-purpose and embedded computing platforms, we identify two key limitations that must be overcome before they can be applied to medical devices. First, current software verification tools are targeted at verification of hardware-independent code written in high-level programming languages($\mathrm{C/C{+}{+}/Java}$) , which are not suitable for interrupt-driven programs written for medical devices. Second, and more importantly, comprehensive verification of security in medical devices requires the specification and verification of domain-specific properties that deal with the interaction of medical devices with the real world. Current software verification tools verify generic program properties such as buffer overflows. While this does provide some value, it is far from sufficient. For example, rather than merely verifying that a program is free of buffer overflows, we may be interested in verifying whether any execution path in the program leads to the cardiac pacing signal not being generated within a specified time window.

Our work makes two key contributions: 1) it demonstrates the use of formal software verification techniques to enhance security of medical devices, and 2) it proposes an approach to bridge the gaps described above so that relevant system-level properties may be verified. We bridge the first gap by transforming the program source code to reflect events such as interrupts, hardware-dependent behavior, and time. We bridge the second gap by transforming the real-world device safety properties into assertions in the program that can be verified.

## II. Background and Related Work

One formal software verification approach is model checking. The model checker exhaustively examines the state transitions in the program, and if a state or transition violates the property, an execution trace (counterexample) is produced. An example bounded model checker, which only explores states reachable within a bounded number of transitions, is CBMC [4]. CBMC can verify array bounds (buffer overflows), pointer safety, exceptions, and user-specified assertions. For example, for every use of an array element $a[i]$, an assertion may be verified that $i$ is less than or equal to the specified upper bound on the array index. CBMC explores all possible execution traces of the program and if there exists an execution trace in which $i$ is larger than the upper bound, the trace is reported.

Medical device software is usually developed from system specifications that document the required "safe" behavior of the medical device at its real-world interfaces. E.g., an example property in a pacemaker specification [5] is "the amplitude of the Pace-Now parameter shall have a value of $5.0\ \mathrm{V} \pm 0.5\ \mathrm{V}$."

Manuscript received March 18, 2013; accepted July 30, 2013. Date of publication August 02, 2013; date of current version August 26, 2013. This work was supported by the NSF under Grant CNS-0914787. This manuscript was recommended for publication by S. Ramesh.

C. Li and N. K. Jha are with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08540 USA (e-mail: chunxiao@princeton.edu; jha@princeton.edu).

A. Raghunathan is with the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA (e-mail: raghunathan@purdue.edu).

Color versions of one or more of the figures in this letter are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/LES.2013.2276434

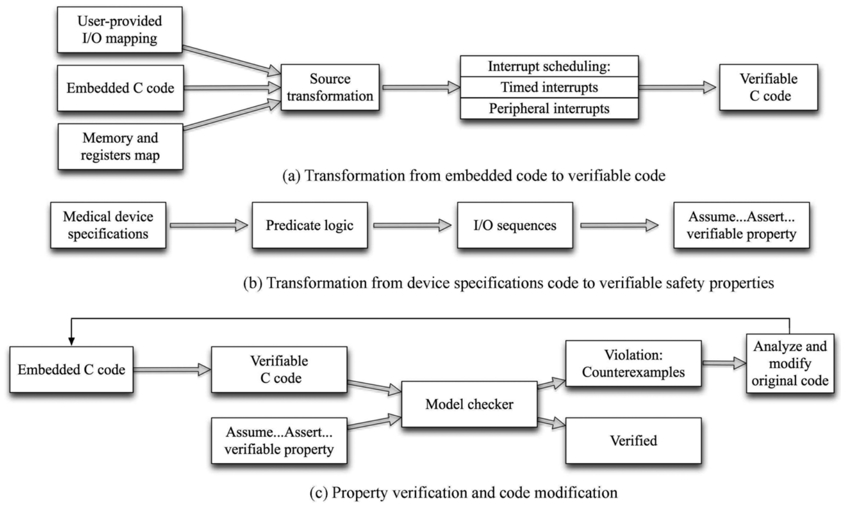
User-provided I/O mapping
Embedded C code
Memory and registers map
Source transformation
Interrupt scheduling:
Timed interrupts
Peripheral interrupts
Verifiable C code
(a) Transformation from embedded code to verifiable code
Medical device specifications
Predicate logic
I/O sequences
Assume...Assert... verifiable property
(b) Transformation from device specifications code to verifiable safety properties
Embedded C code
Verifiable C code
Assume...Assert... verifiable property
Model checker
Violation: Counterexamples
Analyze and modify original code
Verified
(c) Property verification and code modification

Fig. 1 Workflow for verifying safety properties of medical device software.

A survey article [6] reviews the security and privacy issues related to implantable medical devices. Current trends in the development and use of high-confidence medical cyber-physical systems are introduced in [7]. Researchers have also demonstrated possible attacks against specific medical devices, such as insulin pumps [8] and pacemakers [9].

Model-driven engineering is discussed in [10] and [11] to realize a safety-assured implementation of infusion pump software and real-time software, respectively. However, we believe our proposed approach works better in verifying legacy code, and the verified code may be more optimized than the model-synthesized code. In [12], researchers apply model checking to interrupt-driven software at the assembly level. Some articles [3], [13], [14] explore software verification targeted at sensor nodes, medical device systems or other embedded systems.

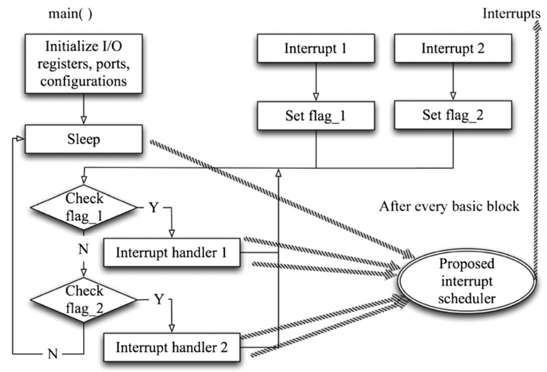

Fig. 2 Event-driven software architecture for medical device software.

## III. Methodology

In this section, we discuss our methodology for verifying safety properties of medical devices. As shown in Fig. 1, the embedded C code of the medical device is transformed into verifiable C code, which is then fed to a C code model checker, such as CBMC. Safety-related medical device specifications are then transformed to verifiable "assumptions and assertions," which are also fed to a C code model checker. The model checker verifies this transformed model of code against assertions, and reports whether it has been verified or a violation has been found.

### *A. Transforming Software to Enable Verification*

Medical device software is often written in hardware-specific microcontroller C code, with some features that are incompatible with verifiable ANSI C [15], e.g., direct hardware accesses, hardware-specific instructions, and interrupts. A user-provided I/O mapping profile is used to build the connections of real-world semantics and internal code conventions, and embedded platform-provided memory and register maps are important for transforming direct-hardware access in the code. Providing an exhaustive list of hardware-specific instruction transformations is beyond the scope of this letter–readers can refer to [3].

Scheduling of interrupts poses a significant challenge to code transformation. For this purpose, interrupts can be categorized as timed or peripheral. Timed interrupts are triggered at a specific time, such as timer and ADC interrupt (if configured to sample at a fixed time interval). Peripheral interrupts, such as a universal synchronous/asynchronous receive/transmit (USART) port interrupt, can be triggered at any time.

For simplicity of exposition, we assume that the embedded code is written using an event-driven architecture similar to the one shown in Fig. 2. This is commonly used for power efficiency. We also assume that nesting of interrupts is not allowed, which is the default configuration for typical C compilers on MSP430. This means that the general interrupt enable (GIE) bit remains unset inside the interrupt service routine.

In order to enable the verification of temporal safety properties, we declare and maintain a global time variable in the embedded C code. After compiling the original embedded code, we conservatively compute the number of cycles ($c_i$) and the time ($t_i$) required for execution of each basic block $BB_i$ [estimated by the number of instructions and the million instructions per second (MIPS) rating of the microcontroller], and increment the global time variable ($T$) at the end of each basic block in the transformed code to note the time lapse.

The interrupt scheduler is called after the execution of each basic block or when the microcontroller goes to sleep. It does the following:

- updates the global time variable $T$;
- checks $T$ to schedule any timed interrupt that is due;

TABLE I
PACEMAKER OPERATION MODES

| | I | II | III |
|---|---|---|---|
| Category | Pacing | Sensing | Response |
| Letters | O-None<br>A-Atrium<br>V-Ventricle<br>D-Dual | O-None<br>A-Atrium<br>V-Ventricle<br>D-Dual | O-None<br>T-Triggered<br>I-Inhibited<br>D-Tracked |

- randomly decides to schedule a peripheral interrupt.

Both paths corresponding to whether the peripheral interrupt is called or not are examined and verified against the assertions in general. However, since we verify the model of C code, we cannot fully model the actual execution time with complexities that account for caching or pipelining mechanisms.

### *B. From Medical Device Specifications to Verifiable Properties*

Safety-related properties should be stated in the medical device specifications. For example, in the case of pacemakers, the specification may indicate when, how long, and at what voltage a pacing signal should be sent to the heart under any given device configuration. The translation is carried out over several steps. As shown in Fig. 1, we first transform device specifications given in plain English to predicate logic, then add I/O sequences, and finally transform them to model-checker-readable computer language. Examples are given in Section IV.

## IV. CASE STUDY: VERIFICATION OF PACEMAKER SOFTWARE

Next, we provide a case study to show how formal verification methods can be used to improve the trustworthiness of medical device software.

### *A. Pacemaker Basics*

Pacemaker is an implanted electronic device that regulates heart beats [16]. The two core functionalities of a pacemaker are sensing and pacing. Sensing in atrium (A) and ventricle (V) of the human heart are referred to as AS and VS. Pacing in these heart chambers are referred to as AP and VP. Pacemakers have different modes, as shown in Table I. In the response column, "T" implies that a sense in a chamber shall trigger an immediate pace in that chamber. "I" implies that a sense in a chamber shall inhibit a pending pace in that chamber. "D" implies that an AS shall result in a tracked VP after a programmed AV delay, unless a VS was detected beforehand. Thus, mode VVI implies that the pacemaker senses and paces in the ventricle, and a sense inhibits a pending pace in the ventricle. Similarly, mode DDD implies that the pacemaker senses and paces in both chambers, based on tracking with a programmed AV delay.

### *B. Example Pacemaker Embedded Program Written for the MSP430 Microcontroller*

Based on the pacemaker specifications, we wrote embedded C code to implement the logic and timing of pacemaker functions. The configurable parameters include:

- lower rate limit: the number of pacing pulses delivered per minute in the absence of sensed intrinsic activity;
- upper rate limit: maximum rate at which the paced ventricular rate tracks sensed atrial events;
- AV delay: time period from an atrial event (intrinsic or paced) to VP;
- Aatrial/ventricular (A/V) pacing amplitude, pulse width and sensitivity.

We implemented software for an example pacemaker, as shown in Fig. 3. Four interrupts are targeted: two for timers

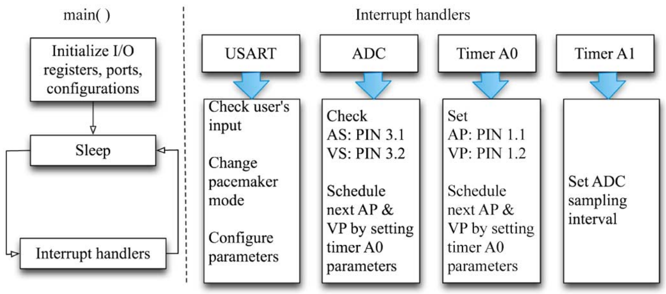

Fig. 3 Implemented pacemaker software.

A0 and A1 on the MSP430, one for the ADC module used for sensing, and one for the user (doctor) configuration interface (USART port). The software is implemented using the coding style suggested in Fig. 2, based on the pacemaker specifications in [5] and [16]. Since it only simulates part of the pacemaker functionality, it is not as rigorous as a realistic implementation. However, we have tested the normal user cases discussed in [5] before performing formal verification. As shown later, the bugs/vulnerabilities we found cannot easily be found in a testing scenario.

### *C. Transformation of Code to Verifiable Code*

To transform the embedded code to verifiable serialized C code, we carry out four steps, as detailed next.

Step 1) We declare and maintain a global time variable in the embedded C code. After compiling the original embedded code, we compute the number of instructions in each basic block and update the time in the transformed code at the end of each basic block.

Step 2) For direct hardware accesses, we transform memory addresses to variables and make consistent changes in the following code.

Step 3) For hardware-specific instructions, we make changes according to the hardware specifications.

Step 4) We add a global interrupt scheduler to the code. The two timer interrupts and the ADC interrupt occur at a specific time. We just check the global time variable and decide whether to call the corresponding interrupt service routine. We randomly decide whether to call the USART interrupt each time we update the global time variable.

### *D. Transformation of Specifications to Verifiable Properties: Three Example Properties*

Note that although it is not a complete list of all safety properties, the parameters we chose to verify are typical and most security-critical because they directly control the timing, amplitude, and duration of pacemaker pacing.

*Property 1:* The device shall output pulses (atrial and ventricular) that provide electrical stimulation to the heart for pacing.

Consider mode VVI. Application of the property to the VVI mode can be expressed in another way: "with inputs that set the device to the VVI mode, output VP can occur and output AP should not." We translate this property into the following code:

```
__CPROVER_assume(mode_state==VVI);
assert((P1OUT&0x08)==0); // assert_1.
assert((P1OUT&0x04)==0); // assert_2
```

"__CPROVER_assume" is a keyword in CBMC, with which the verification tool only checks those execution paths in which the assumption holds. Therefore, it can be used for specifying input sequences. P1OUT&0x08 is the output PIN for AP and P1OUT&0x04 is the output PIN for VP. The expected behavior should be: for assert_1, the code should be verified without violation; for assert_2, CBMC should find a violation and report the violating execution path. Therefore, we can verify that in the VVI mode, AP never occurs and VP can occur.

*Property 2:* The device shall output ventricular pulses with programmable width $\pm 0.2$ ms.

This property can be expressed in another way: "with inputs that set VP width to $X$ ms, the duration of output VP should be within the $[X–0.2, X+0.2]$ ms range. We translate this property into the following code:

```
__CPROVER_assume(vp_w < 19 && vp_w > 1);
if (time_v_pace!=0 && time_v_reset!=0)
  assert(time_v_reset-time_v_pace < (vp_w+2));
  assert(time_v_reset-time_v_pace > (vp_w-2));
```

The setting of the $vp\_w$ value is triggered by the input command directed at configuring the VP width, which should be between 0.1 ms and 1.9 ms. $time\_v\_pace$ and $time\_v\_reset$ refer to the time recorded on the rising edge and falling edge of the output VP signal, respectively. The expected behavior is that there should be no violations of either assertion.

*Property 3:* In the DDD mode, the time interval between two V events cannot be larger than the lower rate interval (LRI), which is programmed by the doctor.

We just record times at which every V event occurs and assert that the time interval between two successive ones should be less than or equal to LRI. In practice, this simple verification step can detect many programming errors.

## *E. Vulnerabilities/Bugs Found by the Verification Process*

In the process of verifying the above three safety properties, we discovered several software vulnerabilities and bugs:

*Example 1:* Interrupt preemption vulnerabilities.

In Fig. 4, a timer interrupt and a port interrupt (used for receiving doctor's configuration command) are shown. If we use the execution order shown in the figure, Property 3 will fail. The reason is that there exists an execution path in which, during the execution of the port interrupt handler, another port interrupt occurs. In this case, the program will forever execute the port interrupt handler and miss the timer interrupt handler, which will delay the A/V pacing time. Thus, this violates the LRI constraints in Property 3. This error may not be easy to find in a testing environment because repeated interruptions from the USART port at a very high rate is not easily realized or may not even be targeted for testing. Although the high-rate interrupts may not be feasible during normal operation, this error may still occur because of potential bugs in the hardware/software of the doctor's programming device or, more importantly, malicious attacks that exploit this vulnerability.

*Example 2:* Parameter range check.

In the early versions of the program, some parameter constraints were not enforced. *E.g.*, upper rate interval (URI) and LRI are two parameters that can be programmed independently by doctors. In practice, if the parameters are programmed in such a way that URI is larger than LRI, then Properties 2 and 3 fail. The verification tool analyzes all possible combinations

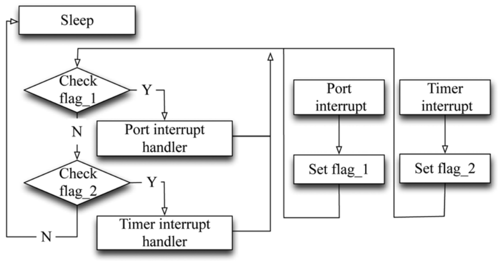

Fig. 4 Example code flow.

of programmed parameters, and reports a violation of properties with specifically configured URI and LRI.

## V. Conclusion

In this letter, a medical device software verification methodology was proposed. Rather than just verifying program properties, such as pointer safety and array bound, the code and assertions are fed to the tool to verify real-world device safety properties. The methodology was illustrated using pacemaker software.

## References

[1] K. Sandler, L. Ohrstrom, L. Moy, and R. McVay, "Killed by code: Software transparency in implantable medical devices," *Software Freedom Law Center Tech. Rep.*, 2010.

[2] D. R. Wallace and D. R. Kuhn, "Failure modes in medical device software: An analysis of 15 years of recall data," in *Proc. ACS/IEEE Int. Conf. Comput. Syst. Appl.*, Jun. 2001, pp. 301–311.

[3] D. Bucur and M. Kwiatkowska, "On software verification for sensor nodes," *J. Syst. Software*, vol. 84, no. 10, pp. 1693–1707, 2011.

[4] Bounded model checker for ANSI-C and C + + programs [Online]. Available: http://www.cprover.org/cbmc/.

[5] Pacemaker system specification [Online]. Available: http://sqrl.mcmaster.ca/_SQRLDocuments/PACEMAKER.pdf

[6] D. Halperin *et al.*, "Security and privacy for implantable medical devices," *IEEE Pervasive Comput.*, vol. 7, pp. 30–39, Jan. 2008.

[7] I. Lee and O. Sokolsky, "Medical cyber physical systems," in *Proc. Design Autom. Conf.*, Jun. 2010, pp. 743–748.

[8] C. Li, A. Raghunathan, and N. K. Jha, "Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system," in *Proc. IEEE Int. Conf. Health Netw. Appl. Services*, Jun. 2011, pp. 150–156.

[9] D. Halperin *et al.*, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," in *Proc. IEEE Symp. Security Privacy*, May 2008, pp. 129–142.

[10] B. Kim, A. Ayoub, O. Sokolsky, I. Lee, P. Jones, Y. Zhang, and R. Jetley, "Safety-assured development of the GPCA infusion pump software," in *Proc. ACM Int. Conf. Embed. Software*, 2011, pp. 155–164.

[11] E. Jee, S. Wang, J.-K. Kim, J. Lee, O. Sokolsky, and I. Lee, "A safety-assured development approach for real-time software," in *Proc. IEEE Int. Conf. Embed. Real-time Comput. Syst. Appl.*, Aug. 2010, pp. 133–142.

[12] C. Fidge and P. Cook, "Model checking interrupt-dependent software," in *Proc. IEEE Asia-Pacific Software Eng. Conf.*, Dec. 2005, pp. 51–58.

[13] L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva, "Semiformal verification of embedded software in medical devices considering stringent hardware constraints," in *Proc. Int. Conf. Embed. Software Syst.*, May 2009, pp. 396–403.

[14] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam, "Modeling and verification of a dual chamber implantable pacemaker," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, Mar. 2012, pp. 188–203.

[15] B. Schlich and S. Kowalewski, "Model checking C source code for embedded systems," *Int. J. Software Tools Technol. Transfer*, vol. 11, pp. 187–202, 2009.

[16] S. S. Barold, R. Stroobandt, and A. F. Sinnaeve, *Cardiac Pacemakers Step by Step: An Illustrated Guide*. New York, NY, USA: Wiley-Blackwell, 2004.