Software-based Gate-level Information Flow Security for IoT Systems

Hari Cherupalli
University of Minnesota
cheru007@umn.edu

Henry Duwe
Iowa State University
duwe@iastate.edu

Weidong Ye
University of Illinois
wye5@illinois.edu

Rakesh Kumar
University of Illinois
rakeshk@illinois.edu

John Sartori
University of Minnesota
jsartori@umn.edu

ABSTRACT

The growing movement to connect literally everything to the internet (*internet of things* or *IoT*) through ultra-low-power embedded microprocessors poses a critical challenge for information security. Gate-level tracking of information flows has been proposed to guarantee information flow security in computer systems. However, such solutions rely on non-commodity, secure-by-design processors. In this work, we observe that the need for secure-by-design processors arises because previous works on gate-level information flow tracking assume no knowledge of the application running in a system. Since IoT systems typically run a single application over and over for the lifetime of the system, we see a unique opportunity to provide application-specific gate-level information flow security for IoT systems. We develop a gate-level symbolic analysis framework that uses knowledge of the application running in a system to efficiently identify all possible information flow security vulnerabilities for the system. We leverage this information to provide security guarantees on commodity processors. We also show that security vulnerabilities identified by our analysis framework can be eliminated through software modifications at 15% energy overhead, on average, obviating the need for secure-by-design hardware. Our framework also allows us to identify and eliminate only the vulnerabilities that an application is prone to, reducing the cost of information flow security by 3.3× compared to a software-based approach that assumes no application knowledge.

CCS CONCEPTS

• **Computer systems organization** → **Special purpose systems**; **Embedded systems**;

KEYWORDS

ultra-low-power processors, security, information flow, hardware-software co-analysis, Internet of Things

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123955

ACM Reference format:
Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Software-based Gate-level Information Flow Security for IoT Systems. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages. https://doi.org/10.1145/3123939.3123955

1 INTRODUCTION

Wearables, sensors, and the internet of things (IoT) arguably represent the next frontier of computing. On one hand, they are characterized by extremely low power and cost requirements. On the other hand, they pose a dire security and privacy risk. As the internet of things progresses toward the internet of everything, where nearly everything is connected to the internet via an embedded ultra-low-power processor, higher connectedness implies more security attack vectors and a larger attack surface. Similarly, immersive usage models imply newer, more sinister consequences. The security and privacy concerns are not theoretical either. In the last couple of years, reported IoT attacks include compromising baby monitors to enable unauthorized live feeds [1], interconnected cars to control a car in motion [2], smartwatches and fitness trackers to steal private information and health data [3], power grids and steel mills to render them offline [4], and medical devices with detrimental, perhaps fatal, consequences on patients' health [5]. Consequently, security and privacy need to be first order design concerns for IoT systems.

Information-flow security is one of the most well-studied approaches to providing security and privacy in computer systems [6–17]. The goal is to track flows of information through a computer system and either detect or prevent illicit information flows between tainted (e.g., untrusted or secure) state and untainted (e.g., trusted or non-secure) state. Tracking and managing flows allows a computer system to support different information flow policies and provide information flow guarantees that security and privacy constructs and protocols can be built upon. An information flow security-based approach can be invaluable in context of IoT systems due to the above discussed security and privacy risks associated with such systems.

The vast majority of techniques for tracking and managing information flows operate at the level of the ISA or above. While these techniques allow tracking and management of explicit information channels, they are largely incapable of doing the same for implicit or covert channels (including timing channels) [16]. *Gate-level information flow security* approaches [16, 17] have been proposed to allow tracking and management of information flow channels at the finest-grained digital level – gates. These approaches typically augment hardware logic blocks with gate-level information flow tracking

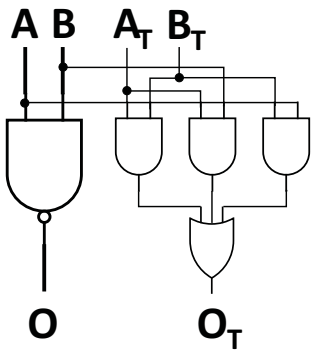
(GLIFT) logic to perform information tracking. They also specify a method for performing compositions of augmented logic blocks. A gate-level approach allows tracking of *all* information flows – implicit, explicit, and covert – allowing one to build secure-by-design systems [18, 19] with varying degrees of programmability, performance, and area.

While gate-level approaches are effective at providing information flow security, such approaches require hardware modifications. For example, required hardware support may include stacks of isolated timers that can reset the PC, new pipeline control hardware, support to manage memory bounds masking, and partitioned memory structures (caches, branch predictors, etc.) [19]. While these modifications may be acceptable for certain high-assurance systems [19], the ultra-low cost requirements of many IoT applications and the volume nature of their microcontrollers may prohibit such modifications.

We observe that many of the required architectural changes arise because prior works assume that all software besides the kernel is completely unknown. Since many emerging IoT applications run the same software again and again for the lifetime of the system, we argue that there is a unique opportunity to build low-overhead gate-level information flow techniques for these IoT systems. Many IoT systems – consider wearables, implantables, industrial controllers, sensor nodes, etc. – perform the same task (or a small set of tasks) repetitively. However, cost reasons dictate that these systems are implemented using a programmable microcontroller running application software instead of an ASIC. We observe that for such systems, it may be possible for a commodity microcontroller to guarantee gate-level information flow security for a *given* application, even if a guarantee cannot be provided for all applications.[1] Similarly, for some applications where gate-level information flow guarantees are not met, it may be possible to guarantee gate-level information flow security only through minimal changes to the application software, even if these changes will be inadequate at providing guarantees for all applications (or for other processors). The ability to guarantee gate-level information flow security for the applications of interest on commodity hardware, even if no guarantee is provided for all applications, allows trusted IoT execution without the programmability, performance, and monetary costs of specialized secure-by-design systems derived from previous gate-level approaches.

We rely on these observations to build a software tool that performs gate-level information flow tracking for a given application on a given processor design without any hardware design effort. The tool takes as input the processor's gate-level netlist, unmodified application binary, and information flow policies of interest, and performs symbolic (i.e., input-agnostic) gate-level simulation of the application binary on the netlist to determine if any of the information flow policies *could* be violated. If none of the information flow policies could be violated at the gate-level, the processor is declared to guarantee gate-level information flow security for the application. If an information flow policy could be violated, the tool reports the offending instruction(s) to the programmer as warnings or errors. This information can then be used by the programmer or the compiler to modify application software such that gate-level information flow guarantees are met for the application. This analysis can be applied to an arbitrary application and even for a commodity hardware design. Also, our approach can be used selectively for the applications limiting overheads only to certain applications that need software modification.

[1] In this paper, we refer to the *application* as the entire binary code loaded into a system's program memory. This includes all computational tasks as well as all system software.



A	A_T	B	B_T	O	O_T
0	0	0	0	1	0
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	1	0
0	1	0	1	1	1
0	1	1	0	1	1
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	1	0	1

Figure 1: Example truth table for gate-level information flow tracking of a NAND gate. A '1' in the taint value columns (shaded gray) represents a tainted value (e.g., untrusted or secret values). Taint is propagated through a gate based on the gate's input values (e.g., A, B) and their corresponding taints (e.g., A_T, B_T). If a tainted input can affect the output (e.g., when $A = 0$, $A_T = 1$, and $B = 1$), the output becomes tainted. However, a taint does not propagate to the output when a tainted input *cannot* affect the output (e.g., due to masking, as when $A = 1$, $A_T = 1$, $B = 0$, and $B_T = 0$).

This paper makes the following contributions:

- We present the first approach to track gate-level information on a per-application basis. Our approach is software-based and can track gate-level information even on a commodity hardware design.
- We show that feedback from our application-specific gate-level information flow tracking can be used to modify application software in a way that guarantees information flow security.
- We show that application-specific software modification to prevent only the insecure information flows that a system is vulnerable to can reduce overheads significantly (i.e., by 3.3×), on average, compared to an "always on" software-based approach that assumes no knowledge of the application running in a system.

2 BACKGROUND AND RELATED WORK

Information flow security aims to (1) determine if any information flows exist from one state element (e.g., a variable in a program) to another state element and to (2) prevent or warn users of such flows when a flow violates an *information flow policy*. Past work [6–11] has performed information-flow tracking at the software level and demonstrated its effectiveness at detecting a set of security vulnerabilities without modification of the hardware (i.e., applicable on commodity hardware). Other work [12–15] proposes hardware modifications for improved efficiency and accuracy of ISA-level information flow tracking. Unfortunately, these approaches not only require hardware modifications, but they may still miss information flows that crop up as a result of the low-level implementation details of a processor [16]. Our approach aims to achieve the advantages of both software-based and hardware-based information flow tracking – applicability to unmodified commodity hardware, accuracy in tracking information flows, and minimal runtime overhead – without the corresponding limitations.

In order to track all forms of digital information flow, Tiwari *et al.* [16] proposed gate-level information flow tracking (GLIFT). As shown in Figure 1, GLIFT augments each gate in a design with taint-tracking hardware. The taint of a gate's output is determined by the values and taints of its inputs. By propagating taint values through each gate, tainted data (e.g., untrusted or secret) can be tracked from input ports (or other marked data, including instructions in program memory) through the processor at the gate level to guarantee that no tainted data reaches an output port that should remain untainted (e.g., a trusted or non-secret output). When fabricated with the base design, GLIFT can dynamically track taints at a high degree of accuracy, albeit at up to a 3× overhead in hardware. More recently, GLIFT has been used to statically track information flows [19]. In this work, an analysis called *-logic is used to statically track taints for a microkernel with no non-determinism running on hardware designed to be easily verifiable. The focus was on performing gate-level information flow tracking for a specific, application-agnostic secure-by-design system. We focus, instead, on performing application-specific gate-level information flow tracking for arbitrary IoT applications on commodity hardware, including applications with control dependencies on unknown, tainted inputs. When analyzed with *-logic, such applications could unnecessarily taint all software-exercisable gates (Footnote 8).

Based on the insights and verification of GLIFT, several secure-by-design processors have been built. They range from a predication-based, non-Turing-complete processor [16] to processors that can handle arbitrary computations through hardware compartmentalization [18, 19]. While these processors can guarantee that any software that runs on them cannot violate a non-interference information security policy (i.e., no untrusted inputs can affect trusted outputs and no secret inputs can affect non-secret outputs), they can be limited in their programmability (e.g., [16] requires all loops to be statically bounded while [18] does not naturally support unbounded or variable-length operations) and require hardware modifications (e.g., partitioned memory structures and memory bounds checking hardware). The cost of any re-design of a commodity microcontroller may be prohibitive in the context of IoT systems, given the huge diversity of IoT systems being driven by commodity microcontrollers [20]. In this paper, we design full systems that ensure the same non-interference policy as [19] (i.e., no untrusted input can affect a trusted output and no secret input can affect a non-secret output), but on a per-application basis.

Recently, a body of work has emerged on developing hardware description languages and tools to design and verify information flow secure hardware [21–24]. While such works can prove that a hardware design meets an information flow security policy, even one that is commercial, such as ARM's Trustzone [24], these approaches cannot verify commodity hardware that does not already implement information flow security. Our approach targets commodity hardware, in addition to emerging hardware, and allows application developers to demonstrate the security of their applications at a fine-grained level.

3 MOTIVATION

In this section, we motivate an application-specific approach for gate-level information flow security in IoT systems through a series of examples. In the first example (Figure 2), we consider existing secure-by-design systems based on gate-level information flow tracking. These systems have been designed assuming that the application software running on the system is unknown [16, 18, 19]. Figure 2 depicts a processor running an unknown application (all Xs). In the figure, port P1 is an input port through which the processor may read tainted data. Also, a partition in the data memory is marked as containing tainted data. Since the application is unknown, we are forced to assume that the unknown instructions may read tainted data from all possible sources, propagate tainted data to all parts of the processor, and also write tainted data to all untainted ports and memory regions. I.e., we must assume that an unknown application has the potential to cause all possible information flow security violations. Faced with this possibility, the *only* way to guarantee information flow security is to design a secure-by-design system that includes hardware mechanisms to proactively prevent all possible insecure information flows. While this approach results in a system that is immune to all possible security violations that an arbitrary application may cause, such stringent security measures require modifications to processor hardware and may often be overly-conservative in an IoT system that runs a single, and often simple, application. For example, consider Figure 3, which shows the same processor running a known application. When the application is run on the processor, it never writes tainted data to untainted ports or memory partitions.[2] Therefore, it is possible to guarantee information flow security for this system *without making any changes to the hardware or software*. This example demonstrates that guaranteeing information flow security is possible, even for an application running on a commodity processor, when the application software is known. This is encouraging for security-critical IoT systems, which, due to economic considerations, more often than not rely on lightweight commodity processors.

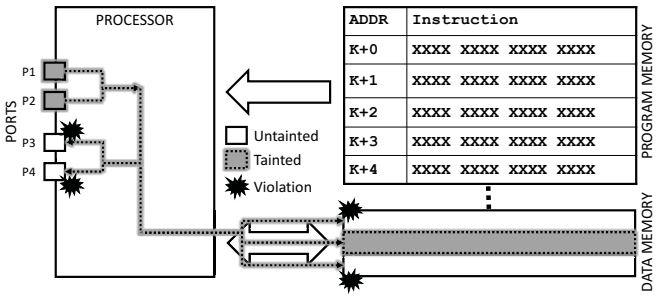


Figure 2: Assuming that an application is unknown means that the application may perform any action, including reading from all possible sources of tainted data, propagating tainted data to all parts of the processor, and writing tainted data through all untainted ports if hardware-based mechanisms are not put in place to prevent insecure information flows.

For the next example, consider Figure 4, which shows the same processor running a different known application that reads an input from a tainted port an uses it as a base pointer (offset) to access data memory. Since the input is tainted, it is possible that the memory address calculated from the offset maps to the untainted region of memory, allowing tainted data to propagate through the memory to an untainted output port. Thus, the application contains a potential information flow security violation that could be either exploited intentionally (e.g., an input supplied by a malicious attacker) or exposed

[2] Tracking of information flow violations is simplified for demonstrative purposes in these examples by assuming that the only flows that exist are the ones visible in the abstract processor representation; actual identification of tainted information flows requires gate-level tracking to ensure complete coverage [16] (see Section 4).

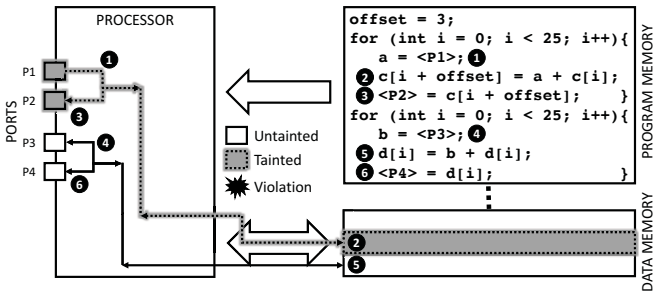


Figure 3: An IoT system that runs a single application may not be vulnerable to any insecure information flows, even when the application is run on a commodity processor. In this example, tainted/untainted code only uses tainted/untainted ports, and no insecure information flows are possible.

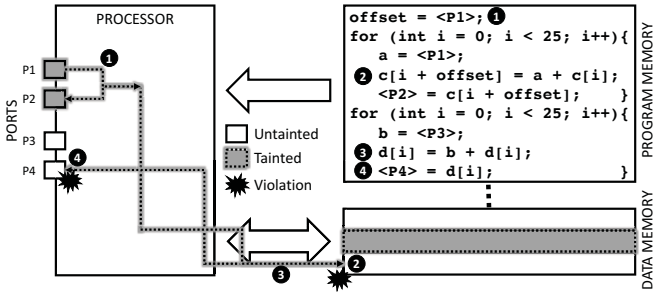


Figure 4: This application is vulnerable to information flows that could jeopardize system security. The application uses tainted input data to compute the address for a write operation. The write taints untainted memory, allowing a violation when tainted data are sent out of an untainted port.

unintentionally (e.g., an unfortunate input supplied by an unwitting user).

Although the application in Figure 4 is vulnerable to an insecure information flow, it does not necessarily mean that the application must be run on a secure-by-design system with hardware-based security mechanisms to ensure information flow security. Consider Figure 5, which shows a different, functionally-equivalent version of the same application running on the same commodity processor. In this version of the application, the base address (offset) read from the tainted port is filtered through a masking operation that sets certain bits in the address (e.g., the most significant bits) to ensure that addresses computed using the offset map only to the tainted region of memory. Since this software change prevents the possibility of propagating tainted data to an untainted output port, no information flow security violations are possible for the modified application. Thus, through knowledge of the application and its potential security exploits, it is possible in this case to prevent information flow security violations in a system only by making changes to the software running in the system.

The examples in this section show that (1) it is possible to guarantee information flow security on a commodity processor without the use of restrictive, hardware-based information flow control mechanisms, and (2) it is possible to eliminate information flow security violations in an embedded system simply by making software modifications. However, these possibilities can only be realized with (1) knowledge of the application running in the system, and (2) a means of identifying all possible insecure information flows to which the application is vulnerable.

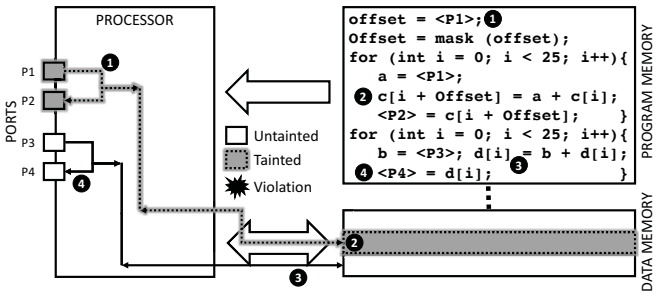


Figure 5: A simple change to the application in Figure 4 (masking the tainted memory address to limit its scope) renders the system immune to insecure information flows, demonstrating that it may be possible to provide information flow security on a commodity processor by changing the software that runs on the processor.

Based on these insights, we propose an application-specific approach to guaranteeing information flow security for IoT systems that identifies all information flow security violations that are possible for a system consisting of a commodity processor and application software and provides software-based techniques that can be used to prevent these information flow security violations.

4 APPLICATION-SPECIFIC GATE-LEVEL INFORMATION FLOW TRACKING

Section 3 motivates the potential benefits of a software-based *application-specific* approach to information flow security, but bringing the application into the picture presents several challenges for gate-level information flow tracking. While it does allow secure-by-design systems to be built on commodity hardware, it requires a means of identifying all possible insecure information flows that may occur in a system, for all possible executions of the system's software, for any possible inputs that may be applied to the system. In this section, we describe an automated technique that takes as input the hardware description (gate-level netlist) of a processor, the software that runs on the system, and labels identifying trusted / untrusted (or secure / insecure) inputs and outputs in the system and efficiently explores all possible application-specific execution states for the system to identify all possible insecure information flows in the system. The output from our automated framework can be used to verify the information flow security of a system as well as to guide and automate software modification to eliminate information flow security vulnerabilities in the system.

Figure 6 shows the process for verifying a security policy using application-specific gate-level information flow tracking. The first step performs offline input-independent gate-level taint tracking of an entire systems binary running on a gate-level description of a processor. The initial components that are tainted are specified by the information flow security policy (e.g., ports labeled as untrusted or memory locations labeled as secret). The result of taint tracking is a per-cycle representation of tainted state (both gates and memory bits). The second step performs information flow policy checking where the information flow checks specified by the information flow security

policy are verified on the per-cycle tainted state. The result is a list of possible violations of the information flow security policy.

4.1 Input-independent Gate-level Taint Tracking

Algorithm 1 describes our input-independent gate-level taint tracking. Initially, the values of all memory cells and gates are set as unknown values (i.e., Xs) and are marked as untainted. The system binary, consisting of both tainted and untainted partitions[3], is loaded into program memory. Our tool performs input-independent taint tracking based on symbolic simulation, where each bit of an input is set to an unknown value symbol, X. Additionally, inputs or state elements may be tainted according to the specified information flow security policy (e.g., the non-interference policy described in Section 2). Throughout simulation, logical values are propagated throughout the circuit as standard ternary logic. Taint values, which are dependent on both the taint values of inputs and their logical values, are propagated as described in [16] and exemplified in Figure 1.

A key difference between our input-independent gate-level taint tracking and prior analyses such as *-logic occurs when an unknown symbol propagates to the PC. For example, directly applying a *-logic analysis on commodity hardware to an application where the PC becomes unknown and tainted results in most of the gates in the hardware also becoming unknown and tainted, since most gates are impacted by the PC. However, in our analysis, if an X propagates to the PC, indicating input-dependent control flow, our simulator branches the execution tree and simulates execution for all possible branch paths (i.e., the abstract representation of the PC is made concrete while still retaining the taint values), following a depth-first ordering of the control flow graph. Since this naive simulation approach does not scale well for complex or infinite control structures which result in a large number of branches to explore, we employ a conservative approximation that allows our analysis to scale for arbitrarily-complex control structures while conservatively maintaining correctness in exploring possible execution states. Our scalable approach works by tracking the most conservative gate-level state that has been observed for each PC-changing instruction (e.g., conditional branch). The most conservative state is the one where the most variables are assumed to be unknown (X). When a branch is re-encountered while simulating on a control flow path, simulation down that path can be terminated if the symbolic state being simulated is a substate of the most conservative state previously observed at the branch (i.e., the states match or the more conservative state has Xs in all differing variables), since the state (or a more conservative version) has already been explored. If the simulated state is not a substate of the most conservative observed state, the two states are merged to create a new conservative symbolic state by replacing differing state variables with Xs, and simulation continues from the conservative state.

The result of the conservative approximation technique is a pruned execution tree that stores both the logical and taint values at each point. Once a state, such as S_2, is observed for a second time, there is no further exploration down that path since all further states have already been considered. This conservative approximation technique allows input-independent gate-level taint tracking to complete in a tractable amount of time, even for applications with an exponentially-large or infinite number of execution paths.[4]

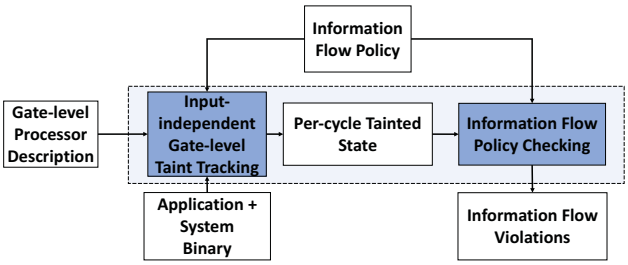


Figure 6: Application-specific gate-level information flow tracking evaluates specific information flow security policies across all possible executions of the entire system binary, producing a list of all possible violations.

Algorithm 1 Input-independent Gate-level Taint Tracking

```
1.  Procedure Taint Tracking(system_binary, design_netlist, security_policy)
2.  Initialize all memory cells and all gates in design_netlist to untainted X
3.  Mark tainted ports and gates according to security_policy
4.  Load system_binary into program memory
5.  Propagate reset signal
6.  s ← State at start of system_binary
7.  Table of previously observed symbolic states, T.insert(s)
8.  Symbolic execution tree, S.set_root(s)
9.  Stack of un-processed execution points, U.push(s)
10. mark_all_gates_untoggled(design_netlist)
11. while U != ∅ do
12.     e ← U.pop()
13.     while e.PC_next != X and !e.END do
14.         e.set_inputs_X() // set all peripheral port inputs to Xs
15.         e.set_taints(security_policy) // taint appropriate state according to security_policy
16.         e' ← propagate_gate_values(e) // simulate this cycle
17.         t ← propagate_taint_values(e',e) // determine taint values for e'
18.         S.add_simulation_point(e',t) // store logical and taint state
19.         if e'.modifies_PC then
20.             c ← T.get_conservative_state(e)
21.             if e' ⊄ c then
22.                 T.make_conservative_superstate(c,e')
23.             else
24.                 break
25.             end if
26.         end if
27.         e ← e' // advance cycle state
28.     end while
29.     if e.PC_next == X then
30.         c ← T.get_conservative_state(e)
31.         if e ⊄ c then
32.             e' ← T.make_conservative_superstate(c,e)
33.             for all a ∈ possible_PC_next_vals(e') do
34.                 e'' ← e.update_PC_next(a)
35.                 U.push(e'')
36.             end for
37.         end if
38.     end if
39. end while
```

4.2 Information Flow Checking

The result of input-independent gate-level taint tracking is a conservative symbolic execution that represents all possible executions of the entire system's binary. This symbolic execution tree is annotated with logical gate values and associated taint values. Using these taint values, information flow checking can be performed where the specific security policy is checked. An example information flow security

[3] Note that tainted and untainted code partitions do not indicate that the corresponding instructions are marked as tainted or untainted in the program memory, although our tool allows them to be.

[4] Some complex applications and processors might still require heuristics for exploration of a large number of execution paths [25, 26]; however, our approach is adequate for ultra-low-power systems, representative of an increasing number of future uses which tend to have simple processors and applications [27, 28]. For example, complete analysis of our most complex system takes 3 hours.

policy is defined by [19]: input and output ports are labeled as trusted or untrusted and, independently, as secret or non-secret (i.e., untrusted and secret are two taints that are analyzed separately). An attacker is assumed to have complete control over all untrusted inputs to the device and controls the initial implementation of untrusted code, which is known at analysis time. No untrusted information can flow out of a trusted port, and no secret information can flow out of a non-secret port.

4.3 Illustrative Example

This section illustrates how application-specific gate-level information flow tracking works. Figure 7 depicts application-specific gate-level information flow tracking on an example portion of a processor circuit using a symbolic execution tree that identifies all information flows in all execution paths of an application.

Consider a small portion of a processor represented by the simple state machine in the top left of Figure 7 and implemented by the circuit in the bottom left of the figure. During application-specific gate-level information flow tracking of the application binary, the gate-level circuit is symbolically simulated using logical 1s, 0s, and Xs (i.e., unknown value symbols). Along with the values of each gate, a taint value is associated with each gate and is propagated according to the gate type and input values of the gate (taint values are shown with a light gray background). The right side of Figure 7 contains an example (abbreviated) symbolic execution tree that tracks taint values through all possible execution paths of an application during application-specific gate-level information flow tracking. In cycle 0, the circuit starts out in an unknown, yet untainted state (i.e., both S and In are Xs while S_T and In_T are 0s). As a result of the untainted reset asserted in cycle 0, the circuit enters a known state, $S = 0$. Input In becomes an untainted 1 in cycle 1, resulting in the circuit transitioning to an untainted $S = 1$ state in cycle 2. After cycle 2, the PC (not shown) becomes an unknown value (X), so symbolic simulation is split into two paths. Since In is a tainted 0 in cycle 2 and propagates its taint to S', both branches start in a tainted state $S = 1$ in cycle 3. In cycle 3 of the left-hand path, In, which is unknown and untainted is XORed with S, which is tainted, and the circuit transitions into an unknown tainted state ($S = X, S_T = 1$). In cycle 4 of the left-hand path, a tainted reset is asserted, which puts the circuit into a known state, $S = 0$. However, since the reset signal was tainted, the output state of the flip-flop remains tainted ($S_T = 1$). This illustrates that a tainted reset signal will not untaint processor state elements. However, on the right-hand path, an *untainted* reset is asserted in cycle 4. This does reset the circuit into a known and untainted state ($S = 0, S_T = 0$).

After tracking taints through every execution of the application, the execution tree characterizes all possible information flows for the application and can be used to identify all possible information flow violations. The specific conditions that we check for violations are described in Section 5.1.

5 GUARANTEEING INFORMATION FLOW SECURITY FOR AN APPLICATION

In this section, we describe software-based techniques that eliminate information flow security vulnerabilities in applications. Section 5.1 establishes conditions that are sufficient to guarantee information flow security, and Section 5.2 describes software transformations that are designed to guarantee that an application that is vulnerable to insecure information flows will satisfy the sufficient conditions. In Section 5.3, we verify that the software transformations achieve information flow security when run on a commodity processor, and in Section 5.4, we prove that the transformations satisfy the sufficient conditions and ensure information flow security.

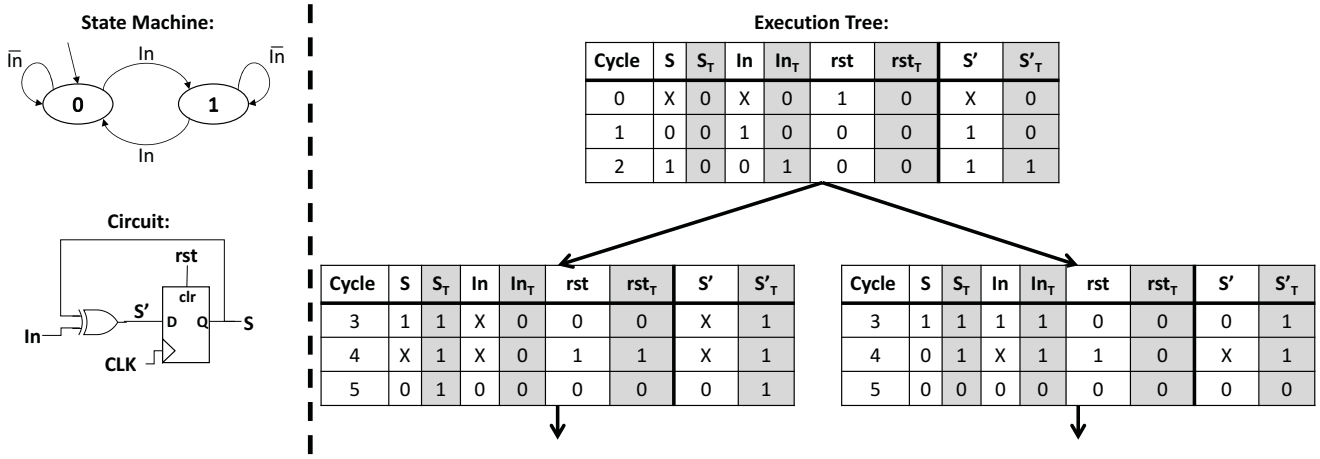
5.1 Sufficient Conditions for Guaranteeing Information Flow Security

In this section, we lay out a set of conditions that are sufficient for guaranteeing the non-interference information flow security policy described in Section 2. Later, we will show how our application-specific approach to information flow security satisfies these conditions.

(1) All processor state elements are untainted before untainted code (i.e., trusted or non-secret code) is executed.
(2) Tainted code does not taint an untainted memory partition used by untainted code.
(3) Untainted code does not load data from a tainted memory partition.
(4) Untainted code does not read from tainted input ports.
(5) Tainted code does not write to untainted output ports.

While the conditions above are not necessary for guaranteeing information flow security, they are sufficient; i.e., a system that maintains the conditions will not leak information. For an information leak of tainted data to occur, tainted data must be accessible to an untainted task in some state or memory element or through a port; a leak occurs when an untainted task propagates accessible tainted data to an untainted output that it has access to, or when a tainted task sends tainted data directly to an untainted output. The conditions above are sufficient to guarantee information flow security because they preclude all possible direct (through a port) or indirect (through state or memory) channels through which tainted information could leak. The first four conditions preclude all possible indirect information flows of tainted data, stating that if an untainted task executes in a taint-free processor, its memory partition remains taint-free, and it does not load tainted data from tainted memory or ports, its computations and outputs will remain untainted. The last condition precludes direct information flows of tainted data, stating that a tainted task is not allowed to write to untainted output ports.

Since the set of conditions above are sufficient, a system that meets the conditions guarantees non-interference. Secure-by-design processors use hardware-based information flow control mechanisms to guarantee that the above conditions are met for all possible applications that execute on the processor [16, 18, 19]. However, none of the conditions above are actually *necessary* to guarantee non-interference. For example, it is acceptable for state elements to be tainted when an untainted task executes (a violation of condition 1), as long as the computations performed by the task do not depend on any tainted state elements. Similarly, exceptions can be made for all the sufficient conditions (they are not necessary). Thus, as long as the original non-interference property (see Section 2) holds, any or all of the sufficient conditions described above may be relaxed. This insight has several interesting implications. (1) Since our symbolic analysis technique for input-independent gate-level taint tracking can check whether the non-interference property holds for all possible executions of a known application without forcing the application to meet the conditions above, it is possible to provide a security guarantee for any application that has no possible violations, even on a commodity processor



Cycle	S	S_T	In	In_T	rst	rst_T	S'	S'_T
0	X	0	X	0	1	0	X	0
1	0	0	1	0	0	0	1	0
2	1	0	0	1	0	0	1	1

Cycle	S	S_T	In	In_T	rst	rst_T	S'	S'_T
3	1	1	X	0	0	0	X	1
4	X	1	X	0	1	1	X	1
5	0	1	0	0	0	0	0	1

Cycle	S	S_T	In	In_T	rst	rst_T	S'	S'_T
3	1	1	1	1	0	0	0	1
4	0	1	X	1	1	0	X	1
5	0	0	0	0	0	0	0	0

Figure 7: Example of application-specific gate-level information flow tracking.

that is not secure by design. (2) Since symbolic input-independent gate-level taint tracking can identify all possible instances where an application causes the non-interference property to be violated for a system, it can be used to identify locations where an application must be modified to prevent insecure information flows, as well as to verify whether a modified application is secure. (3) Some applications have no possibility of violating one or more of the conditions above. Therefore, some security mechanisms applied by secure-by-design processors represent unnecessary overhead for those applications. On the other hand, if insecure information flows can be eliminated through software modifications, the modifications can specifically target only the insecure information flows to which an application is vulnerable, potentially reducing the overhead of providing security for the system and enhancing programmability (by imposing fewer restrictions on software).

5.2 Software Techniques to Eliminate Insecure Information Flows

When the sufficient conditions for information flow security described in the previous section are not satisfied, it is possible for tainted information to leak. For example, allowing an untainted task to read and operate on tainted data may result in tainting of a processor's control flow state, and subsequently the execution of an untainted task. Specifically, if a processor's program counter (PC) becomes tainted, then all subsequent instructions will be tainted. Therefore, the control flow of an untainted computational task can also become tainted if it executes after a tainted task that taints the processor's control flow state. In fact, once the PC is tainted by a tainted task, it is possible that control may never become untainted, even if control is returned to untainted code. Preventing information flows from tainted to untainted code must include prevention of all direct information flow (e.g., the tainted code cannot call a yield function to return to untainted execution) and all indirect information flow (i.e., there must exist a mechanism that deterministically bounds the execution time of the tainted code). To avoid information leaks through control flow, there must exist an untaintable, deterministic mechanism that recovers the PC to an untainted state that fetches code from an untainted code partition.

Another common way for information to leak in a commodity processor is through the memory. If code that is allowed to handle tainted information writes to data memory using a fully tainted address, then the entire data memory, including partitions belonging to untainted code, will become tainted. For example, if tainted code reads a value from a tainted input port and then uses the value as an index to write into an array, the tainted address causes the entire data memory to become tainted, not just the memory location pointed to by the address. To avoid such leaks, a mechanism is needed to guarantee that no possible execution of tainted code can write to an untainted data memory partition.

For cases where an application violates the sufficient conditions and is vulnerable to insecure information flows, we propose two software transformations, analogous to the hardware mechanisms presented in [18], that target and prevent insecure information flows.

Untainted Timer Reset: An untainted timer can be used to reset the PC to an untainted location after a deterministic execution time of running tainted code, thus guaranteeing that tainted code cannot affect the execution of untainted code. However, on a commodity processor (e.g., openMSP430), generating such a timer is challenging for two reasons. First, common mechanisms for setting the PC, such as interrupts, still depend on the current, possibly tainted state of the pipeline to determine when the PC is reset. Second, the timer must not become tainted. As an example, on the openMSP430, a timer could be directly tainted by tainted code writing to its memory-mapped control register. To overcome these challenges, we propose using the watchdog timer that is common to many microcontrollers to reset the entire processor after a deterministic-length period of tainted execution. We use our symbolic simulation-based analysis to guarantee that the watchdog remains untainted.

Figure 8 shows our proposed watchdog timer reset. During the execution of a context switch in an untainted system code partition, the watchdog timer is set to a predetermined value for the computational task that is being switched in. The untainted system code then transfers execution to the tainted computational task. This tainted task can make full use of the processor, except writing to the watchdog or an untainted memory space partition or port, possibly propagating taints throughout the pipeline. When the untainted watchdog expires,

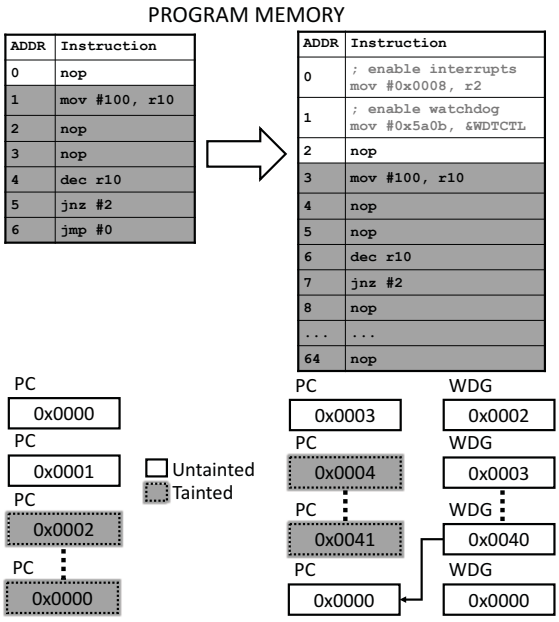


Figure 8: Untainted timer reset example: In the left-hand code listing, all instructions after address 0 are marked as tainted. Once the first tainted instruction is loaded, the PC quickly becomes tainted and the jump back to the untainted instruction at address 0 does not reset the PC to being untainted. However, by setting the watchdog timer during the untainted portion of the code (the right-hand code listing) and padding the tainted portion with nops, the PC can be reset to an untainted value in the untainted partition of code.

it resets the entire pipeline with a power-on reset (POR).[5] Since this reset is untainted, the state within the pipeline will be reset to untainted values, including the PC.

While using the watchdog timer flushes tainted data from the processor, the subsequent reset state is only untainted if the watchdog timer itself remains untainted. Since applications are known during analysis, the symbolic simulation used during input-independent gate-level taint tracking allows us to identify whether or not any tainted code can write to the control register of the watchdog timer during any possible execution of the tainted code. If there is no possibility of tainted code writing to the control register of the watchdog timer, the write enable input for the control register is verified to be untainted. The only information this can leak is the fact that the tainted code does not access the watchdog timer – a known requirement for guaranteeing information flow security using our approach.

Note that this mechanism works naturally in multi-programming and task switching environments that are common in realtime embedded systems. Before context switching to a tainted computational task, the untainted system code simply sets the watchdog timer to the appropriate interval for the task – either the maximum length of the task or the length of an OS time slice, depending on the usage scenario. Expiration of the timer resets the processor to an untainted state, as usual, which also resets the PC. The code at PC=0 either contains or vectors to the system routine for switching in the next context.

[5]We assume that the POR does not reset memory. This is a reasonable assumption, since many microcontrollers have non-volatile memory, including TI's MSP430FRXX series.

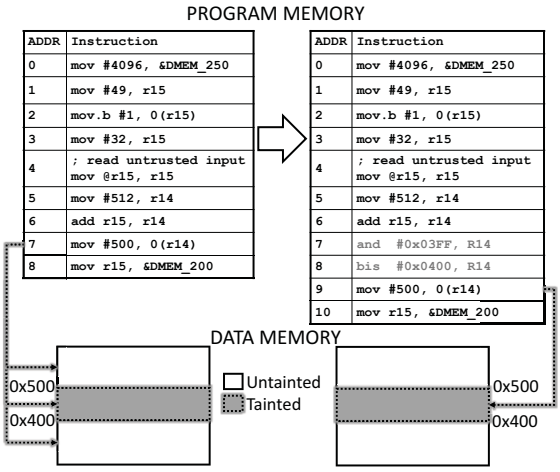


Figure 9: Memory mask example: In the left-hand code listing, the instruction at address 4 reads a tainted and unknown input. At address 6, the tainted input is used to calculate the address for a store at address 7. Since the address is both unknown and tainted, that store ends up tainting the whole data memory space. By adding two instructions at addresses 7 and 8 (see the right-hand code listing) that mask the address to only use the tainted task's memory partition, no untainted memory locations become tainted.

If a tainted computational task wants to use the watchdog timer, it may not be possible to certify the system as *secure* unless a) it is impossible for the tainted task to cause a control flow violation or b) an alternative, functionally-equivalent (or otherwise acceptable) option can be used in place of the watchdog timer. Microprocessors typically provide several hardware timers, and it may be possible to emulate the functionality desired by the tainted task using a different timer. If it is not possible to use another available timer, software optimizations such as prediction may be used to eliminate the possibility of control flow violations.

Software Masked Addressing:

Figure 9 shows our proposed memory bounds masking. The left side shows the original assembly code where a tainted address is used to store data, tainting the entire data memory. On the right side, the assembly code is modified to mask the memory address to guarantee that it falls within the region of data memory to which tainted code is allowed to write. Input-independent taint tracking can then verify that no taint is propagated to memory regions that are untainted. While simple masking solves the memory address taint problem for the case where the PC remains untainted, masking alone cannot guarantee information flow security when the PC becomes tainted. In this case, the tainted PC taints the masking instructions themselves. However, during application-specific gate-level information flow tracking, the program, including the added masking instructions, is known. In this case, our information flow tracking analysis can verify that no possible execution of the tainted code can generate an address outside of the regions of data memory that are allowed to be tainted. If there is no possibility of being able to write outside of allowed memory regions, there is no possibility of information flow, either explicit or implicit, between the allowed and disallowed memory regions. The only information flow that can leak is the information that the tainted

application does not write outside of its allowed memory region – a known condition for guaranteeing information flow security.

5.3 Verification of Software Techniques

Here, we verify that our software techniques for guaranteeing information flow security indeed work using the micro-benchmarks presented in Figure 8 and Figure 9 and an unmodified openMSP430 processor. Consider the left-hand code listing in Figure 8. We initialize the input-independent gate-level taint tracking such that the instructions shaded gray are tainted. During any possible execution of the application, once the PC becomes tainted, it never becomes untainted again. However, if the watchdog timer is set using untainted code (see the right-hand code listing in Figure 8), each execution of the untainted code section has a trusted PC. Now consider the right-hand code listing in Figure 9. Here, the code itself is not marked as tainted, but the code reads data from a tainted port and uses it to index into an array. During input-independent taint tracking, each input that is read from the tainted port is tainted. We observe during information flow tracking that the entire memory space becomes tainted, due to the propagation of tainted data to a memory address calculation. When instructions are inserted that guarantee that the unknown address is bounded to the tainted task's region in data memory, then the result of information flow tracking indicates that no untainted memory locations can be tainted.

5.4 Proving Information Flow Security

Theorem: For a system $\mathcal{S}$ consisting of a processor $\mathcal{P}$ and application $\mathcal{A}$, if application-specific gate-level information flow tracking $\mathcal{T}_\mathcal{S}$ of $\mathcal{S}$ reports that $\mathcal{S}$ is secure (i.e., satisfies the non-interference property), tainted data in $\mathcal{P}$ will never influence the execution of a trusted computational task $\mathcal{A}_\mathcal{I}$ in $\mathcal{S}$, and $\mathcal{P}$ will never propagate tainted data through an untainted output.

Proof: For tainted data to influence the execution of $\mathcal{A}_\mathcal{I}$, a taint must propagate from a tainted input of $\mathcal{S}$ to an untainted output written by $\mathcal{A}_\mathcal{I}$ either through a state element of $\mathcal{P}$, through the memory, or directly from a port.

Case 1 – taint propagation through a state element: For taintedness to influence $\mathcal{A}_\mathcal{I}$ through a state element $\mathcal{E}$, $\mathcal{E}$ must be tainted by a tainted computational task $\mathcal{A}_\mathcal{J}$ and remain tainted while $\mathcal{A}_\mathcal{I}$ is executing on $\mathcal{P}$. However, in any case where $\mathcal{T}_\mathcal{S}$ identifies a possible tainted information flow from $\mathcal{A}_\mathcal{J}$ to $\mathcal{A}_\mathcal{I}$, $\mathcal{A}$ is modified to invoke the watchdog timer mechanism to reset all state elements in the processor after the execution of $\mathcal{A}_\mathcal{J}$ and before the execution of $\mathcal{A}_\mathcal{I}$. Therefore, taint propagation through a state element is impossible, as long as $\mathcal{A}_\mathcal{J}$ does not interfere with the untainted operation of the watchdog timer. Since $\mathcal{T}_\mathcal{S}$ checks all possible execution states of $\mathcal{A}$ on $\mathcal{P}$ and also reports that $\mathcal{A}$ is insecure if a taint propagates to the watchdog timer in any possible state, assurance of security from $\mathcal{T}_\mathcal{S}$ means that it is impossible for tainted data to propagate through a state element and influence the execution of $\mathcal{A}_\mathcal{I}$.

Case 2 – taint propagation through memory: For taintedness to influence $\mathcal{A}_\mathcal{I}$ through the memory, a tainted computational task $\mathcal{A}_\mathcal{J}$ must write to some memory location $\mathcal{M}$ outside its tainted memory partition, and $\mathcal{A}_\mathcal{I}$ must read from that memory location while it is executing on $\mathcal{P}$. However, in any case where $\mathcal{T}_\mathcal{S}$ identifies that $\mathcal{A}_\mathcal{J}$ could write outside of its memory partition, $\mathcal{A}$ is modified such that masking instructions are inserted to ensure that $\mathcal{A}_\mathcal{J}$ can only write inside its own memory partition. Furthermore, $\mathcal{T}_\mathcal{S}$ checks all possible execution states of $\mathcal{A}$ on $\mathcal{P}$ and reports that $\mathcal{A}$ is insecure if a tainted write is performed to untainted memory or a read is performed from tainted memory by any untainted computational task. Therefore, assurance of security from $\mathcal{T}_\mathcal{S}$ means that it is impossible for tainted data to propagate through memory and influence the execution of $\mathcal{A}_\mathcal{I}$.

Case 3 – taint propagation through a port: For taintedness to propagate to an output through a port, either some $\mathcal{A}_\mathcal{I}$ must read from a tainted port or some $\mathcal{A}_\mathcal{J}$ must write directly to an untainted port. Both cases are reported as insecure by $\mathcal{T}_\mathcal{S}$ as it evaluates all possible execution states of $\mathcal{A}$. Therefore, assurance of security from $\mathcal{T}_\mathcal{S}$ means that it is impossible for tainted data to influence the execution of $\mathcal{A}_\mathcal{I}$ or propagate to an untainted output from a port. ■

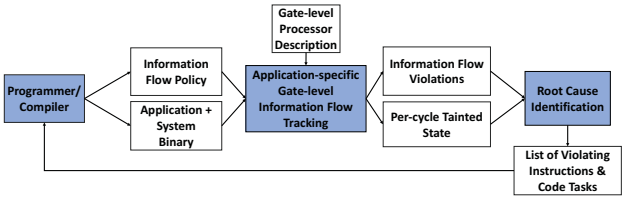


Figure 10: Software refactoring tool flow: Based on the results of application-specific gate-level information flow tracking (Section 4), root cause identification generates a list of instructions that violate the untainted memory partitions condition and code tasks (e.g., functions) that violate the untainted control flow condition. These can then be used by a programmer or compiler to apply software-based fixes for the information flow violations.

6 A TOOLFLOW FOR SOFTWARE-BASED GATE-LEVEL INFORMATION FLOW SECURITY

We have developed an end-to-end toolflow, depicted in Figure 10, for developing systems that guarantee information flow security on commodity processors. The first stage in the toolflow checks whether an application conforms to a given information flow security policy. This stage takes as input the application software, including application code, library code, and system code, as well as the gate-level description of the processor, and performs application-specific gate-level information flow tracking (Section 4) on the system for a developer-defined information flow security policy that provides tainted / untainted labels for hardware and software (e.g., ports, code partitions, data partitions). The output of information flow tracking is a list of all possible information flow violations that may be generated by the application, along with cycle-accurate tainted state for each type of information flow.

To guarantee information flow security for the system, all identified violations must be eliminated by modifying the application software. To this end, the next stage of the toolflow reports potential information flow security violations to the developer at instruction-level granularity. This stage identifies the root cause of each potential gate-level violation – i.e., the instructions that lead to violations. For violations where the PC becomes tainted during execution of a tainted code partition, our root cause identification tool marks the tainted partition as having tainted control flow, requiring the watchdog mechanism to be invoked. In cases where a store instruction in a tainted program partition can potentially write to an untainted memory partition, the

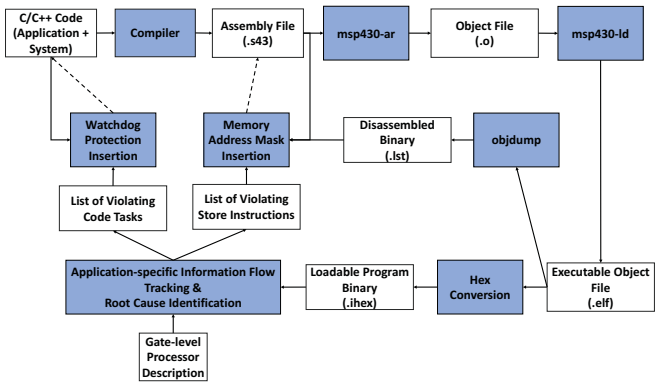


Figure 11: Automated software modification to eliminate information flow security violations for MSP430 involves C/C++ source code compilation and assembly into an object file that is linked with libraries, including a runtime support library to generate an executable object file, which is converted into hex that can be loaded into program memory. Application-specific information flow tracking and root-cause analysis use the final hex (program memory contents) to identify code tasks and store instructions that can cause violations. If a watchdog timer is needed, it is enabled in the system software via a `#define`. Any necessary mask insertions are performed in the assembly file at the specific addresses (instructions) identified by root cause analysis (if the watchdog was needed, analysis must be performed again on the new assembly file prior to mask insertion). The new, modified assembly file is run through the remainder of the toolflow to produce a new hex file to be loaded into the program memory.

static instruction (identified by its address in the program memory) is marked as needing masking.

The final stage of our toolflow refactors the software of the application in order to guarantee information flow security. The necessary software modifications identified by root cause analysis can be applied either manually or automatically by the compiler (Figure 11).[6] For each instance where the compiler applies a modification to the software to eliminate a possible insecure information flow, it also reports a compile error or warning (depending on the severity of the violation) to the developer, indicating the line of code that caused the violation and the change that was made to fix the violation. Errors are reported for direct information leaks of tainted data that are not allowed (e.g., tainted code writes to an untainted output port), and warnings are reported for violations that may indirectly lead to information leaks if not fixed (e.g., a store from tainted code can write to an untainted memory partition). Reviewing the list of compile errors and warnings can be informative, since some violations and fixes are unavoidable (e.g., tainted control flow resulting from a control instruction that depends on a tainted input, which is fixed using the watchdog timer), while other violations may be caused by unintended software bugs (e.g., a store that is vulnerable to buffer overflow, which is fixed by masking). In the case of unintended software bugs, changing the program code may avoid the need for automated software modification to eliminate violations (e.g., fixing the buffer overflow problem avoids the need to mask the store).

After software has been modified to eliminate all possible information leaks of tainted data, application-specific gate-level information flow tracking can be used to verify that it is now impossible for the system to violate the specified information flow policy, i.e., the system now guarantees information flow security.

The feedback provided by our toolflow potentially represents another benefit of application-specific information flow tracking over secure-by-design processors. Our toolflow identifies and reports all possible causes of insecure information flows. Thus, security vulnerabilities are brought to the developer's attention and can be addressed appropriately, resulting in an application that is secure. In a secure-by-design processor, hardware mechanisms are used to alter the functionality of the application silently, so an application's security vulnerabilities may never be remedied, or even known. Also, violations corrected silently in hardware may manifest as runtime errors. For example, address masking performed by hardware can fix a buffer overflow problem, but the result is probably to map the store to some erroneous location inside the buffer, resulting in an erroneous execution / output for the application.

[6] Two specific cases require programmer attention. First, if an untainted task directly accesses a tainted memory location or input port or a tainted task directly accesses an untainted output port, there is a fundamental violation in the software. In this case, an error is reported and the programmer must either change the software to eliminate the illegal access or redefine the information flow security labels. Second, if a tainted task originally uses the watchdog and also requires the watchdog for information flow guarantees, the programmer must either avoid using the watchdog or refactor the program to avoid tainting control flow (see Section 5.2 for details).

Table 1: Benchmarks

Embedded Sensor Benchmarks [34]
mult, binSearch, tea8, intFilt, tHold, div, inSort, rle, intAVG
EEMBC Embedded Benchmarks [35]
Autocorr, FFT, ConvEn, Viterbi

7 RESULTS

Processor and Benchmarks: We perform evaluations on a silicon-proven processor – openMSP430 [29], an open-source version of one of the most popular ultra-low-power processors [30, 31]. The processor is synthesized, placed, and routed in TSMC 65GP technology (65nm) for an operating point of 1V and 100 MHz using Synopsys Design Compiler [32] and Cadence EDI System [33]. Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using a custom gate-level simulator that implements application-specific gate-level information flow tracking (Section 4). We show results for all benchmarks from [34] and all EEMBC benchmarks [35] that fit in the program memory of the processor (Table 1). Benchmarks are chosen to be representative of emerging ultra-low-power application domains such as wearables, internet of things, and sensor networks [34]. Benchmark performance (IPC) on our processor varies from 1.25 to 1.39.

We evaluate the information flow security of each benchmark running as a tainted computational task on the system (ports it uses are labeled tainted). System code is an untainted task consisting of the instructions needed to restart the benchmark after each execution.

7.1 Information Flow Violations

Application-specific gate-level information flow tracking reports all possible information flow violations for an application. Table 2 shows which of the unmodified benchmarks violated the sufficient conditions

Table 2: Benchmarks that violate sufficient conditions 1 and 2 for information flow security (see Section 5.1) before and after modification.

Benchmark	Unmodified		Modified	
	C1	C2	C1	C2
binSearch	✓	✓	-	-
div	✓	✓	-	-
inSort	✓	✓	-	-
intAVG	✓	✓	-	-
intFilt	-	-	-	-
mult	-	-	-	-
rle	-	-	-	-
tHold	✓	✓	-	-
tea8	-	-	-	-
FFT	-	-	-	-
Viterbi	✓	✓	-	-
ConvEn	-	-	-	-
autocorr	-	-	-	-

Table 3: Performance overhead (%) for watchdog timer reset and memory address masking applied with and without application-specific analysis.

Benchmark	Without Analysis	With Analysis
binSearch	34.63	34.63
div	33.16	33.16
inSort	37.92	10.00
intAVG	45.56	11.90
intFilt	19.58	0
mult	150.9	0
rle	45.61	0
tHold	106.2	106.2
tea8	93.89	0
FFT	17.63	0
Viterbi	1.029	1.029
ConvEn	19.69	0
autocorr	42.15	0

described in Section 5.1. Seven benchmarks do not violate any of the conditions. Effectively, our analysis shows that these benchmarks cannot violate our information flow security policy on this processor. However, six benchmarks violate sufficient conditions 1 and 2.[7] These benchmarks require the techniques described in Section 5.2 to guarantee information flow security. After performing software modifications identified by our toolflow, all condition violations are eliminated.[8] Thus, symbolic gate-level information flow tracking in conjunction with software modification is able to guarantee information flow security for these applications on a commodity processor without hardware-based information flow control mechanisms.

7.2 Runtime Overheads of Software-based Gate-level Information Flow Security

Since we eliminate possible tainted information flows through software modification, guaranteeing information flow security in our approach incurs performance and energy overheads whenever an application has potential violations to eliminate. The right column of Table 3 (*With Analysis*) shows the performance overhead of using the watchdog timer and memory masking to eliminate information flow security vulnerabilities in our benchmark applications. Since application-specific gate-level information flow tracking is able to identify and eliminate only the tainted information flows that an application is vulnerable to, applications that are not vulnerable to tainted information flows require no modifications and incur no overhead. For applications where modifications are necessary, masking is applied to store instructions that may be tainted, and the watchdog timer is used to deterministically bound the execution time of tainted computational tasks.

[7]None of our benchmarks violate sufficient conditions 3, 4, or 5; however, this is not surprising for well-written code, since the conditions preclude scenarios like reading memory out of bounds or illegal port accesses.

[8]When *-logic analysis was used to verify information flow security on the six applications with information flow violations, it identified that the condition violations were not removed. This is because these applications have control dependences on an unknown, tainted input, which causes *-logic to taint the PC and make it unknown, resulting in 70% of the gates in MSP430 becoming unknown and tainted, even those required by the software techniques to remain untainted (e.g., the watchdog timer). Therefore, a direct application of *-logic analysis would not allow the software-based techniques to be verified on commodity hardware.

Since the MSP430 watchdog has a maximum interval length of 32768 cycles, which may not be long enough to bound the longest execution time of a computational task, we evaluate a system that implements time-slicing (e.g., as an RTOS might schedule one computational task across multiple time slices). Also, since the execution time of a task may not be an even multiple of one of the available watchdog timer intervals (64, 512, 8192, and 32768 cycles), an infinite idle loop is added at the end of each benchmark to fill the remainder of the final time slice. The number and duration of time slices are selected to minimize overhead, based on the available watchdog timer intervals and the overhead of state checkpointing and recovery (context switching) for time slicing.[9] Intuitively, using fewer, longer time slices for a given task duration incurs less overhead for context switching but may incur more idling overhead in waiting for the final watchdog interval to complete. Our toolflow accounts for the overheads of context switching and scheduling the watchdog timer, along with the maximum duration of a computational task, to select the number and duration of watchdog intervals that minimize overhead while providing a deterministic bound on execution time.

Since application-specific gate-level information flow analysis indicates precisely which computational tasks need to be protected by a watchdog timer and which store instructions need to be protected by address masking, the techniques can be applied only where necessary. On the other hand, guaranteeing information flow security for an unknown application requires masking of every store and time bounding of every tainted task using a deterministic timer, since all sufficient conditions must be satisfied to guarantee non-interference, even though they may not be necessary for a particular application (Section 5.1). Without the ability to identify all possible tainted information flows for all possible executions of an application on a commodity processor using input-independent gate-level information flow tracking, an "always-on" approach for information flow control would be required.

The left column of Table 3 (*Without Analysis*) shows the performance overhead of using masking for all stores and time bounding for all tainted tasks, representing a case where application analysis is not available and all sufficient conditions must be enforced. In this case, performance overhead is 3.3× higher than in the case where application analysis is able to target only the possible insecure information flows for an application. Even considering only the applications that have possible information flow security violations, applying software-based techniques only where necessary reduces performance overhead by 24%. Overall, application-specific information flow analysis can minimize the overhead of providing information flow security guarantees on a commodity processor using software-based techniques.

7.3 Information Flow Secure Scheduling: A System-level Use Case

In this section, we show that we can use the techniques developed in this work to guarantee information flow security at the system level; we focus on an IoT system that performs scheduling between multiple tasks. Specifically, we show that without any modifications to the processor, we can guarantee that (1) there are no insecure

[9]For openMSP430, the overhead of saving and restoring a task's state is 20 cycles, and watchdog timer initialization and reset takes 10 cycles.

information flows across scheduled tasks, and (2) no task can affect the scheduling performed by the system software. In order to demonstrate these properties, we construct an IoT system in which FreeRTOS [36] performs task scheduling between two tasks – `div` and `binSearch` – where `binSearch` is an untrusted task (its input and output ports are marked as untrusted), and FreeRTOS and `div` are both trusted.

The control flow of `binSearch` depends on an untrusted input value. Thus, in the baseline case, after `binSearch` is scheduled on the processor, the processor's control flow becomes tainted. Among the consequences of this tainting are that (1) the trusted task `div` becomes untrusted the next time it is scheduled, and (2) the scheduling of FreeRTOS itself is compromised, since it too becomes untrusted as a result of the tainted task.

To provide information flow security for this system, we use our toolflow to modify the system's application, consisting of FreeRTOS and the two computational tasks. 330 store instructions in `binSearch` are identified as potential security violations, and our toolflow applies memory masking to these instructions. Also, our toolflow invokes the watchdog timer mechanism around the untrusted task. This modification is performed in FreeRTOS system code. The value of the reset interrupt vector is set to a location in the middle of FreeRTOS's scheduler interrupt. On a watchdog-invoked reset, scheduling is performed as usual with the exception that the watchdog timer is also reset with the scheduling timer prior to restoring the context of the next task from its own stack. After modification, application-specific information flow tracking verifies that the application runs successfully without any tainting of the trusted task or the RTOS.

We measure the performance overhead of our modification using input-based gate-level simulations; runtime is measured from when the first task is scheduled to when both tasks have completed. The total performance overhead of adding the watchdog timer reset and memory masking is only 0.83%. The overhead is low since only `binSearch` requires memory masking and the modifications required to add the watchdog timer to FreeRTOS's system code are small (e.g., FreeRTOS already requires context saving and restoring).

The above example shows that we can guarantee gate-level information flow security with low-overhead software modifications for an application built on a commodity RTOS. More broadly, this shows that our techniques are applicable at the system-level and can be used to verify complex and system-level security properties.

8 GENERALITY AND LIMITATIONS

We target application-specific information flow security for IoT applications with ultra-low area and power constraints. Low-power processors are already the most widely-used type of processor and are also expected to power a large number of emerging applications [37–41]. Such processors also tend to be simple, run relatively simple applications, and often do not support non-determinism (no branch prediction and caching; for example, see Table 4). This makes our symbolic simulation-based technique a good fit for such processors. Below, we discuss how our technique may scale for complex processors and applications, if necessary.

More complex processors contain more performance-enhancing features such as caches, prediction or speculation mechanisms, and out-of-order execution, that introduce non-determinism into the instruction stream. Symbolic co-analysis is capable of handling this added non-determinism at the expense of analysis tool runtime. For example, by injecting an X as the result of a tag check, both the cache hit and miss paths will be explored in the memory hierarchy. Similarly, since co-analysis already explores taken and not-taken paths for input-dependent branches, it can be adapted to handle branch prediction. In an out-of-order processor, instruction ordering is based on the dependence pattern between instructions. While instructions may execute in different orders depending on the state of pipelines and schedulers, a processor that starts from a known reset state and executes the same piece of code will transition through the same sequence of states each time. Thus, modifying input-independent CFG exploration to perform input-independent exploration of the data flow graph (DFG) may allow analysis to be extended to out-of-order execution.

Table 4: Microarchitectural features in recent embedded processors.

Processor	Branch Predictor	Cache
ARM Cortex-M0	no	no
ARM Cortex-M3	yes	no
Atmel ATxmega128A4	no	no
Freescale/NXP MC13224v	no	no
Intel Quark-D1000	yes	yes
Jennic/NXP JN5169	no	no
SiLab Si2012	no	no
TI MSP430	no	no

For complex applications, CFG complexity increases. This may not be an issue for simple in-order processors (e.g., the ultra-low-power processors studied in this paper), since the number of possible execution states that must be evaluated is naturally limited based on the number of instructions that can be resident in the processor pipeline at once. However, for complex applications running on complex processors, heuristic techniques may have to be used to improve scalability; a large number of such heuristics have been proposed [25, 26].

In a multi-programmed setting (including systems that support dynamic linking), we consider the union of all application code (e.g., caller, callee, and relevant OS code in case of dynamic linking) to identify all possible execution states. Similarly, for self-modifying code, the set of exercisable states is determined considering all code versions. In case of fine-grained execution, any state that is not maintained as part of a thread's context is assumed to have a value of X when symbolic execution is performed for an instruction belonging to the thread. This leads to a conservative coverage of execution states for the thread, irrespective of the behavior of the other threads.

9 CONCLUSION

IoT applications present a stress test for information flow security. The rapidly increasing quantity and variety of IoT devices also increases the quantity and variety of data handled by the devices, while driving down the power and area constraints, leaving little budget for security in this security-critical domain. In this work, we showed how knowledge of the application that runs on an IoT device can be leveraged to identify all possible information flow security vulnerabilities in the system, modify application software to eliminate vulnerabilities, and provide a guarantee that a system is information flow secure, even for systems built upon commodity ultra-low-power processors commonly used in IoT applications. Since our analysis framework identifies and eliminates only the information flows that a particular application is vulnerable to, the cost of eliminating all insecure information flows with our application-specific approach to information flow security is 3.3× lower than a software-based approach that assumes no application knowledge.

REFERENCES

[1] M. Stanislav and T. Beardsley, "Hacking iot: A case study on baby monitor exposures and vulnerabilities," 2015.

[2] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, vol. 2015, 2015.

[3] H. Wang, T. T.-T. Lai, and R. Roy Choudhury, "Mole: Motion leaks through smartwatch sensors," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, (New York, NY, USA), pp. 155–166, ACM, 2015.

[4] S. Sridhar, A. Hahn, and M. Govindarasu, "Cyber-physical system security for the electric power grid," *Proceedings of the IEEE*, vol. 100, pp. 210–224, Jan 2012.

[5] J. Sametinger, J. Rozenblit, R. Lysecky, and P. Ott, "Security challenges for medical devices," *Commun. ACM*, vol. 58, pp. 74–82, Mar. 2015.

[6] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, (New York, NY, USA), pp. 196–206, ACM, 2007.

[7] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, (Washington, DC, USA), pp. 135–148, IEEE Computer Society, 2006.

[8] J. Newsome, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, (New York, NY, USA), pp. 133–147, ACM, 2005.

[10] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, (Berkeley, CA, USA), USENIX Association, 2006.

[11] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07*, 2007.

[12] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, (New York, NY, USA), pp. 482–493, ACM, 2007.

[13] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," in *2008 International Symposium on Computer Architecture*, pp. 377–388, June 2008.

[14] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 173–184, Feb 2008.

[15] H. Chen, X. Wu, L. Yuan, B. Zang, P. c. Yew, and F. T. Chong, "From speculation to security: Practical and efficient information flow tracking using speculative hardware," in *2008 International Symposium on Computer Architecture*, pp. 401–412, June 2008.

[16] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," *SIGPLAN Not.*, vol. 44, pp. 109–120, Mar. 2009.

[17] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "On the complexity of generating gate level information flow tracking logic," *IEEE Transactions on Information Forensics and Security*, vol. 7, pp. 1067–1080, June 2012.

[18] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 493–504, Dec 2009.

[19] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 189–200, ACM, 2011.

[20] "Products with an MSP430." http://43oh.com/2012/03/winner-products-using-the-msp430/.

[21] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: A hardware description language for secure information flow," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 109–120, ACM, 2011.

[22] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 97–112, ACM, 2014.

[23] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 503–516, ACM, 2015.

[24] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a practical hardware security architecture through static information flow analysis," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), pp. 555–568, ACM, 2017.

[25] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, pp. 82–90, Feb. 2013.

[26] K. Hamaguchi, "Symbolic simulation heuristics for high-level design descriptions with uninterpreted functions," in *High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International*, pp. 25–30, 2001.

[27] "International Technology Roadmap for Semiconductors 2.0 2015 Edition Executive Report." http://www.semiconductors.org/main/2015_international_technology_-roadmap_for_semiconductors_itrs/.

[28] G. Press, "Internet of Things By The Numbers: Market Estimates And Forecasts," *Forbes*, 2014.

[29] O. Girard, "OpenMSP430 project," *available at opencores.org*, 2013.

[30] Wikipedia, "List of wireless sensor nodes," 2016. [Online; accessed 7-April-2016].

[31] J. Borgeson, "Ultra-low-power pioneers: TI slashes total MCU power by 50 percent with new "Wolverine" MCU platform," *Texas Instruments White Paper*, 2012.

[32] Synopsys, *Design Compiler User Guide*.

[33] Cadence, *Encounter Digital Implementation User Guide*.

[34] B. Zhai, S. Pant, L. Nazhandali, S. Hanson, J. Olson, A. Reeves, M. Minuth, R. Helfand, T. Austin, D. Sylvester, *et al.*, "Energy-efficient subthreshold processor design," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 8, pp. 1127–1137, 2009.

[35] "EEMBC, Embedded Microprocessor Benchmark Consortium." http://www.eembc.org.

[36] "The FreeRTOS website." http://www.freertos.org/.

[37] M. Magno, L. Benini, C. Spagnol, and E. Popovici, "Wearable low power dry surface wireless sensor node for healthcare monitoring application," in *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*, pp. 189–195, IEEE, 2013.

[38] R. Yu and T. Watteyne, "Reliable, Low Power Wireless Sensor Networks for the Internet of Things: Making Wireless Sensors as Accessible as Web Servers," *Linear Technology*, 2013.

[39] A. Dunkels, J. Eriksson, N. Finne, F. Osterlind, N. Tsiftes, J. Abeillé, and M. Durvy, "Low-Power IPv6 for the internet of things," in *Networked Sensing Systems (INSS), 2012 Ninth International Conference on*, pp. 1–6, IEEE, 2012.

[40] R. Tessier, D. Jasinski, A. Maheshwari, A. Natarajan, W. Xu, and W. Burleson, "An energy-aware active smart card," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, no. 10, pp. 1190–1199, 2005.

[41] C. Park, P. H. Chou, Y. Bai, R. Matthews, and A. Hibbs, "An ultra-wearable, wireless, low power ECG monitoring system," in *Biomedical Circuits and Systems Conference, 2006. BioCAS 2006. IEEE*, pp. 241–244, IEEE, 2006.