View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Queensland University of Technology ePrints Archive

QUT Digital Repository:
http://eprints.qut.edu.au/

Fidge, Colin J. and Corney, Diane (2009) ***Integrating hardware and software information flow analyses.*** In: ACM SIGPLAN/SIGBED 2009 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2009), 19-20 June 2009, Trinity College, Dublin.

© Copyright 2009 Association for Computing Machinery

Integrating Hardware and Software Information Flow Analyses

Colin J. Fidge Diane Corney
Faculty of Science and Technology, Queensland University of Technology, Australia
{c.fidge,d.corney}@qut.edu.au

Abstract

Security-critical communications devices must be evaluated to the highest possible standards before they can be deployed. This process includes tracing potential information flow through the device's electronic circuitry, for each of the device's operating modes. Increasingly, however, security functionality is being entrusted to embedded software running on microprocessors within such devices, so new strategies are needed for integrating information flow analyses of embedded program code with hardware analyses. Here we show how standard compiler principles can augment high-integrity security evaluations to allow seamless tracing of information flow through both the hardware and software of embedded systems. This is done by unifying input/output statements in embedded program execution paths with the hardware pins they access, and by associating significant software states with corresponding operating modes of the surrounding electronic circuitry.

Categories and Subject Descriptors D.4.6 [*Security and protection*]: Information flow controls; C.3 [*Special-purpose and application-based systems*]: Real-time and embedded systems; F.3.2 [*Semantics of programming languages*]: Program analysis

General Terms Security, Verification

1. Introduction

Security-critical communications devices are used to protect data confidentiality in many government, military and industrial applications. In particular, *domain separation* devices aim to control the flow of information between classified and unclassified networks. Common examples of such devices are data diodes (which enforce unidirectional information flow), encryption devices (which allow classified data to be sent over insecure communications links), trusted filters (which constrict information flow), and keyboard-video-mouse switches (which allow a single workstation to access both high-security and low-security computers).

Before they can be deployed, such devices must be evaluated to a high degree of rigour [22]. This is usually the job of an experienced *information security evaluator*, skilled in electronic engineering, who tests the device and studies its design to identify security weaknesses. This process includes tracing all potential information flow pathways through the device, to detect unintended communications *channels*. Unintended information flow may occur due to design errors, or unanticipated behaviours when individual electronic components within the device fail or are deliberately attacked [5].

Increasingly, however, security functionality is now being entrusted to microprocessor-based embedded software, both to save costs and to provide greater flexibility. Consequently, previous hardware-oriented security evaluation procedures are no longer adequate, and new techniques are needed to integrate software analyses into information security evaluations of electronic devices.

Traditionally the process of evaluating information flow through electronic circuitry has involved laboriously tracing component connectivity through the device's schematic diagram [4]. To save effort, software tools can automate much of this tedious task [13], although the overall assessment still relies on the security evaluator's skill at interpreting the results. However, the entire procedure is stymied when embedded microprocessors are encountered. Such components typically have dozens of pins interconnecting them to their surrounding circuitry, and the only way to determine how information may flow through a microprocessor is to study the program code running on it. Whereas component connectivity is self-evident in circuitry schematics, information flow between program statements is not immediately obvious by mere inspection of the source code, making the job even harder.

Although security analysis of application-level software, to uncover program design flaws, has been well-explored in the academic literature [19, 17], little attention has been paid to *embedded* software. In particular, embedded programs interact directly with hardware components in their surrounding circuitry, thus creating potential information-flow pathways that traditional software analysis techniques [21] do not consider.

Here we present a high-integrity security evaluation process for tracing potential information flow through both hardware and embedded software. It integrates existing topological circuitry analysis principles [18] with standard program slicing technology [9], to allow seamless tracing of information flow through electronic devices. To make this possible, our strategy: unifies instances of input/output statements appearing in software execution paths with the hardware pins they access; associates software path conditions with the hardware operating modes within which such program states may occur; and takes into account changes to information flow due to program constructs that redirect (read or write) or transform (*downgrade* or *upgrade*) classified data.

2. Related and Previous Work

To combine hardware and software information flow analyses we build on existing techniques and tools from both fields.

2.1 Hardware Information Flow Analysis

Techniques for checking electronic circuitry diagrams for design errors are well established [4], but little has been published on hardware information flow analysis. Consideration has been given to the impact of deliberately-induced hardware faults on overall device security [5] and on cryptographic protocol strength [7], but this work does not address information flow directly. Also relevant is research into security models of embedded microprocessors [11], but our concern is with the software running on such microprocessors, rather than the microprocessors themselves.

Previous theoretical work on information flow through electronic circuitry schematics modelled circuits as directed graphs, with individual electronic components as vertices [18]. How each

electronic component connects its input pins to its outputs is then defined for each of the device's operating (or fault) modes. This model supports an end-to-end reachability analysis to identify modes in which a *high-security source* component (i.e., one receiving or generating classified data) may lead to a *low-security sink* component (i.e., one connected to a publicly-accessible device).

Tool support is essential to help a security evaluator perform this task for complex electronic devices. Topological analysis tools that check whether an idealised electronic circuit is consistent with a given one [12] can be used for this purpose, since topological reachability is related to *potential* information flow.

Foremost among such tools is SIFA (Secure Information Flow Analyser), an open-source software tool[1] for performing mode-specific reachability analyses of electronic circuits [13]. SIFA traces component connectivity and presents the user with a list of those paths through the circuit that connect selected data sources and sinks in particular modes. The security evaluator can then inspect each such path to determine whether or not it poses a security risk. Block diagrams can be entered into SIFA manually, or it can directly import circuitry schematics written in the VHDL hardware design language. Our research aims to be compatible with this tool.

2.2 Software Information Flow Analysis

Security analysis of application-level software, to identify program design vulnerabilities, has been well-explored [17], using either type theory [19], program semantics [10] or program logic [6]. However, little work has been done specific to embedded software.

Research into security analysis of portable Java bytecode programs [2] is relevant to our work, but the symbolic execution approach used there differs considerably from the static analyses performed by tools like SIFA. Also relevant is recent research into automated model checking of embedded microprocessor code [20], but that work aims to help debug embedded programs, rather than tracing information flow through them. Most relevant of all to our work is research into the security of programs that interact with their environment via input/output statements [16], but again the type inference theory developed there does not (directly) alert us to potentially dangerous information flows.

Instead, therefore, we revisited basic compilation principles in our work. Control flow graphs [1, §9.4] and program dependence graphs [8] are fundamental concepts for 'slicing' program code into potential execution and information flow pathways, respectively. As explained below, we make use of both kinds of graph.

3. An Integrated Methodology

In this section we describe our integrated hardware-software information-flow evaluation methodology. The process is illustrated via a detailed example in Sections 4 and 5.

3.1 Background and Assumptions

We assume that a given communications device is to be evaluated for information security using a process like that outlined in international security standards [22]. Such standards require devices to be examined from a variety of perspectives, but for our purposes we are interested in (overt) information flow only.

Our specific concern is communications devices intended for high-integrity situations such as government and military installations. The device itself is assumed to reside in a secure facility, i.e., inside a *high-security domain*, but some of its outputs are sent outside the facility, i.e., to a *low-security domain*. Adversaries are assumed to be located in the low-security domain and want to learn the value of classified data items in the high-security domain. The device is thus required to provide *domain separation*, meaning that only unclassified data may reach the low-security domain.

[1] `http://sifa.sourceforge.net/`

3.2 Problem Statement

Given a hardware schematic and software source code for a security-critical communications device, our goal is to help a security evaluator identify illegal information flow pathways from high-security data sources to low-security data sinks.

The current hardware-oriented evaluation process, using an analysis tool such as SIFA, involves several steps [13]. Firstly the device's block diagram or circuitry schematic is entered into the tool, which converts it into a directed graph representation. The security evaluator then defines the set of operating (or fault) modes of interest, based on the device's intended functionality. Next the evaluator defines how each kind of component within the device connects its inputs to its outputs in different modes. To perform the analysis the evaluator highlights 'source' and 'sink' components of interest and the tool automatically finds all paths that connect sources to sinks, and the mode(s) in which information flows along them. Finally, the security evaluator studies the paths generated to determine whether or not they are acceptable, in the context of the device's intended domain-separation properties.

This process works well when the circuit is composed solely of simple electronic components like logic gates, switches and flip flops. If a microprocessor is embedded in the circuitry, however, the security evaluator is faced with the problem of trying to determine how information flows through this complex component, which inevitably requires an understanding of the way the microprocessor's software works. Furthermore, significant *states* of the software need to be related to corresponding *modes* of the hardware. Our goal, therefore, is to devise a practical way of using compiler technology to fill this gap in the security evaluation process.

3.3 Identifying Information Flow Through Electronic Circuitry

For our purposes we assume the existence of a tool such as SIFA [13], capable of performing mode-specific reachability analyses of electronic circuits modelled as directed graphs [14].

A hardware circuit can be represented as a graph with its vertices P being the set of all physical pins (connectors) on the printed circuit board [14]. Distinct hardware components are represented by the subset of pins on their periphery. Then the 'inter-component' PCB tracks connecting electronic components can be modelled as a set $W \subseteq P \times P$ of pin-to-pin wires.[2]

To define the way information flows through each electronic component we need to associate each potential 'intra-component' connection through it with the set of modes in which information *may* flow along this pathway. Assuming that the set of operating modes identified for all electronic components is M, then intra-component connectivity can be modelled as a function $C \subseteq M \to \mathbf{P}(P \times P)$ which maps each mode to the set of intra-component connections it enables, for all components.[3]

Given a circuit modelled in this way, SIFA performs a reachability analysis between a selected high-security source pin h and a low-security sink pin ℓ, presenting the resulting set of paths to the security evaluator for inspection. In effect, the reachability analysis

[2] Let $S \times T$ be the cross product of sets S and T, i.e., the set of ordered pairs '$s \mapsto t$' for all elements $s \in S$ and $t \in T$.

[3] Let $\mathbf{P}\, S$ denote the powerset of set S, i.e., the set of all subsets, and $X \to Y$ denote the set of total functions from domain X to range Y.

returns the following set of paths.[4]

$$\{p : \text{iseq}\, P \mid \text{first}(p) = h \wedge \text{last}(p) = \ell \wedge \\ \forall i, j : \text{dom}\, p \bullet \\ i + 1 = j \Rightarrow p(i) \mapsto p(j) \in W \vee \\ \exists m : M \bullet p(i) \mapsto p(j) \in C(m)\}$$

Thus, the analysis returns each end-to-end path p that begins at source pin h, ends at sink pin ℓ, and for each step of which there is either a connection through inter-component circuit W, or a mode m in which an intra-component connection $C(m)$ is enabled.

Hardware operating modes may be assumed to be *global*, i.e., common to all components, or *local*, i.e., specific to each component, supporting top-down or bottom-up analyses, respectively [18]. SIFA assigns no semantics to modes—mode 'm' in one component is not necessarily related to an identically-named mode 'm' in another component—and it thus performs bottom-up analyses. However, we can easily interpret its analysis results as global, if desired, by considering only the subset of paths found in which all components share a common mode.

In this context, our challenge is thus to devise a way of populating intra-component connectivity function C with the pin-to-pin microprocessor connections enabled by the program code executing on a microprocessor chip in different operating modes.

3.4 Embedded Program Assumptions and Notations

Embedded programs are usually written in the C programming language or directly in assembly code. Here we merely assume that the program is written in a generic imperative language:

- **var** $v \bullet S$ — Declaration of a variable v whose scope is the following (compound) statement S;
- $v \leftarrow E$ — Assignment of the value of expression E to (type-compatible) variable v;
- S_1 **;** S_2 — Sequential composition of statements S_1 and S_2;
- **if** B **then** S_1 **else** S_2 **fi** — Choice between statements S_1 and S_2 depending on the value of Boolean guard expression B;
- **while** B **do** S **od** — Pre-tested iterative execution of statement S as long as Boolean guard expression B is true; and
- **repeat** S **until** B — Post-tested iterative execution of statement S until Boolean expression B becomes true.

To support expression of run-time paths through program code we also allow the following auxiliary statement to represent evaluation of 'guards' in conditional or iterative statements:

- **assert** B — Boolean expression B evaluates to 'true'.

Since we are interested in embedded code, we also need input and output statements for reading from and writing to hardware components. In practice a variety of processor- and compiler-dependent statements are used for reading and writing digital data, and for sampling or setting binary control signals. Furthermore, these statements differ depending on whether they are potentially blocking (synchronous) or non-blocking (asynchronous).

For simplicity here we assume that embedded programs can directly write values to, and read values from, physical pins on the microprocessor chip. We assume the existence of a special software variable 'pin_x' that gives access to hardware pin number x. Inputs and outputs are then represented as special-case assignments:

- $v \leftarrow \text{pin}_x$ — Read a value from microprocessor pin x into (type-compatible) software variable v; and
- $\text{pin}_x \leftarrow E$ — Write the value of expression E to microprocessor pin x.

We also allow terms of the form 'pin_x' to appear in expressions, under the assumption that the current value on, or accessible via, this hardware pin is sampled whenever the expression is evaluated.

3.5 Extracting Hardware Component Connectivity From Embedded Software

In essence our goal is to extract information flow paths from an embedded program's source code. Control flow graphs and program dependence graphs are the traditional way of 'slicing' programs into separate paths [9], and both are needed in our approach.

In a *control flow graph* [1, §9.4] the vertices are atomic program statements. For our purposes these are assignments and Boolean 'guards' used in **if**, **while** and **repeat** statements. Vertices are connected by an edge in a CFG if the two statements/guards *may* be executed/evaluated consecutively at run time. Tracing a path from one vertex to another thus produces a possible *execution path*.

However, not all such paths are feasible at run time. To exclude infeasible paths we can calculate a *path condition* which characterises the initial states, if any, from which the path *will* be followed. A path condition is the conjunction of the Boolean guards appearing in the path, with appropriate substitutions to take account of assignments to variables appearing in the guards, conjoined with the condition required to reach the beginning of the path from the program's entry point.

Another important way of slicing programs, especially when considering information flow, is to construct a *program dependence graph* [8]. PDGs are derived from CFGs but have two main kinds of edge. A *control flow* edge links Boolean guards in conditional and iterative statements to those statements whose execution relies on the condition. A *data flow* edge connects assignments to immediately following uses of the target variable.

Both kinds of edge are essential for analysing information flow. Data flow edges handle the obvious case of data being explicitly transferred between program variables. More subtly, control flow edges model implicit information flow due to the value of a variable controlling execution of an assignment statement. An oft-cited security-related example of this is the following program, where h is a variable containing classified data and ℓ is a variable visible to the low-security domain, both of type natural number.

$\ell \leftarrow 0$ **;**
while $h > 0$ **do**
  $h \leftarrow h - 1$ **;**
  $\ell \leftarrow \ell + 1$
od

This program transfers the value of h to ℓ even though there is no assignment statement involving both variables. The corresponding program dependence graph reveals that information does indeed flow between these two variables by virtue of the control flow edge connecting guard '$h > 0$' to assignment '$\ell \leftarrow \ell + 1$' whose execution the guard controls. In general, information *may* flow between any two statements in a PDG connected by a sequence of control and/or data flow edges.

For our purposes we are interested in finding paths from high-security data sources to low-security data sinks, via an embedded microprocessor's software. The starting point for each of our paths is thus: an input statement that reads classified data from one of the microprocessor's pins; a statement that retrieves classified data from read-only memory within the microprocessor; a statement that creates new classified data, such as passwords or encryption

[4] Let '$\text{iseq}\, S$' be an injective sequence formed from items in set S, i.e., a sequence containing no repeated items. For such a sequence s, let $\text{first}(s)$ be its first element, $\text{last}(s)$ be its last element, $s(n)$ be its nth element, and '$\text{dom}\, s$' be its domain, i.e., its set of indices.

keys; or a statement that exposes previously-hidden classified data, usually via a call to a decryption function (sometimes referred to as a security class *upgrader*).

The ending point for a path of interest is usually an output statement that writes data to one of the microprocessor's pins. (Strictly speaking we are interested only in pins that may ultimately lead to the low-security domain but, given that it may not be immediately obvious which pins do this, especially if they feed into another microprocessor or even back into the same microprocessor, the safest approach is to explore *all* execution paths ending at output statements.) Furthermore, it's helpful to flag paths that we are confident *downgrade* high-security data [15], e.g., by encrypting it or by filtering out classified data fields.

To do this, the main steps in our approach for identifying potential information flow paths through embedded software are to:

1. extract all CFG execution paths that connect high-security data source statements to data sink statements in the program,
2. exclude those execution paths which have no end-to-end information flow according to the PDG, to reduce false positives, and
3. calculate the CFG path conditions for the remaining paths, to precisely characterise the states in which they may be executed, so that we can relate the software paths to corresponding hardware modes.

Each path remaining after this process is completed then becomes a pin-to-pin link through the microprocessor, before the entire circuit is subjected to topological analysis.

The second step eliminates execution paths that don't transfer information. For instance, if we are interested in potential information flow from microprocessor pin t to pin u then the execution path through program fragment '$v \leftarrow \text{pin_}t$ **;** $\text{pin_}u \leftarrow v$' would be retained because it transfers data directly between these two pins. However, the path through code fragment '$v \leftarrow \text{pin_}t$ **;** $\text{pin_}u \leftarrow w$' would be excluded because no information flows between the pins, even though the two statements are always executed consecutively. (This effect could also be achieved using program representations that integrate both CFG and PDG features, such as Gated Static Single Assignment form [3], but we have found it convenient during experimentation to keep the two kinds of graph separate.)

Although program dependence graphs are normally considered superior to control flow graphs for performing information flow analyses *within* program code, we favour CFG-derived execution paths because they can reveal important facts about information flow *outside* the program. Normally, PDGs ignore statements that don't contribute to end-to-end information flow within the program. For instance, assume we are interested in information flow from variable x to variable z in the code fragment '$y \leftarrow x$ **;** $a \leftarrow x$ **;** $z \leftarrow y$'. Using a CFG we would extract an execution path containing all three statements, because they all occur consecutively when the program is executed. However, the path between the first and third statements in a PDG bypasses the middle statement because it does not contribute to information flow between x and z [21].

While this may appear to be advantageous for our purposes—because it would reduce the security evaluator's workload by eliminating irrelevant statements—we have found that the situation for evaluating *embedded* program code is different because of the possibility that paths contain input/output statements other than those at the endpoints. Such statements create information flow relationships between the program and the electronic circuitry surrounding the microprocessor. As the example in Section 4.2 demonstrates, making the security evaluator aware of an output statement in the middle of an execution path, even though it does not contribute to information flow through the software, can help reveal the presence of significant security problems associated with the flow of information 'around' the program.

3.6 Linking Software States to Hardware Modes

Having identified the software execution paths that (may) allow information flow between microprocessor pins, and the software states from which they can be executed, the final requirement for integrating this knowledge into the hardware model is to associate the software states with corresponding hardware modes. Unfortunately, this is not a straightforward process because there is unlikely to be a simple mapping between those hardware modes identified as important by the security evaluator and the path conditions extracted automatically during analysis of the embedded program.

The combined CFG-PDG analysis process outlined in Section 3.5 produces a path condition for each potential information flow path through the microprocessor. Next we need to ensure that the information security evaluator can always identify corresponding hardware modes in which this software path may be followed.

Software states are characterised by predicates (Boolean-valued expressions) over the program's variables and constants and, in our case, values on hardware pins. Predicates form a lattice, ordered by logical implication, with 'true' at the top and 'false' at the bottom. However, previous work on hardware information flow analysis assumed that hardware modes were mutually exclusive, with the device or component residing in only one mode at a time [18]. This rigid structure means that it isn't always possible to neatly match a given software state with a single hardware mode.

Therefore, we instead decided to treat the security evaluator's set of modes M as also forming a lattice, in this case ordered by mode inclusion, with 'any' at the top and 'none' at the bottom. This strategy means that we can always find a mode that embraces all software states characterised by any given execution path condition. In the worst case we can use the top of the mode lattice, but choosing the lowest possible element will produce the most precise result when the end-to-end reachability analysis is performed.

Recall from Section 3.3 that each mode $m \in M$ is assumed to be characterised by the set $C(m)$ of intra-component connections it enables. We therefore also need to ensure that the mode lattice obeys the following simple well-formedness conditions.

1. Modes form a lattice ordered by inclusion '$\leq$', with a distinguished top element 'Any' and bottom element 'None', such that
$$\forall m, n : M \bullet n \leq m \Rightarrow C(n) \subseteq C(m) \,.$$
In other words, any mode m in the lattice enables at least all information flow allowed by any inferior mode n.
2. The bottom element does not enable any information flow, i.e.,
$$C(\text{None}) = \{\} \,.$$

These constraints ensure that a composite mode high in the lattice subsumes all information flow allowed by its constituent modes lower in the lattice.

Importantly, a mode high in the lattice may introduce connections not allowed by *any* of its inferior modes. Superior modes are thus not merely a choice between their inferiors—they may model behaviours not possible by any one of the inferior modes alone. This lets a higher mode model situations where the device changes from one of the lower modes to another. The previous 'flat' model of modes made it impossible to analyse 'inter-mode' information flow, e.g., where classified data is stored in memory in one mode and retrieved in a subsequent one. Our hierarchical model allows composite modes high in the lattice to represent behaviours that straddle distinct modes lower in the lattice.

As long as the security evaluator can provide a mapping from each software path condition to a hardware mode, or modes, in a

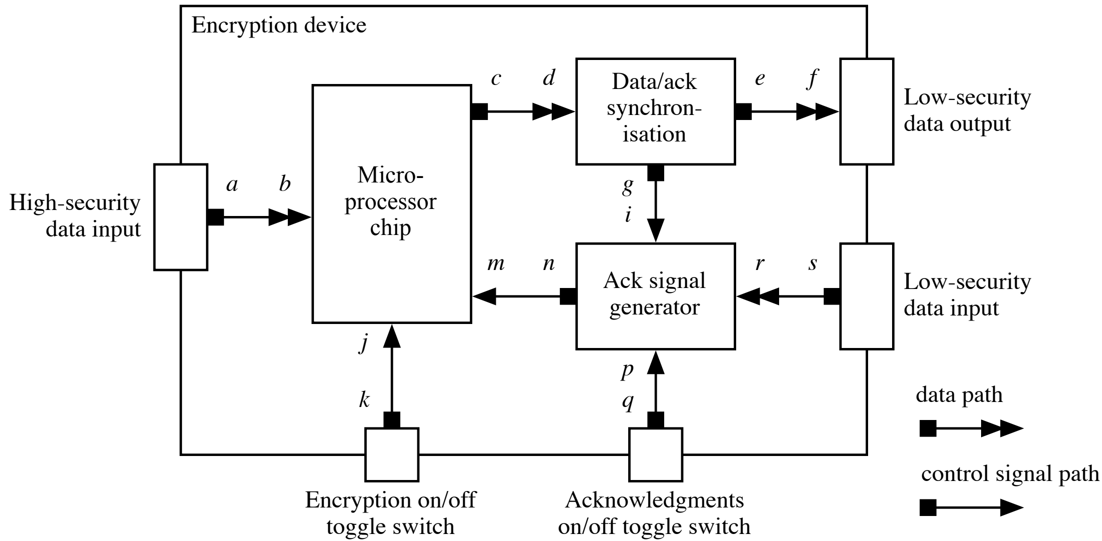

Figure 1. Block diagram for the encryption device.

lattice that obeys these well-formedness constraints then the corresponding microprocessor software execution paths can be integrated easily into the overall reachability analysis, like any other kind of intra-component connectivity, and we can use SIFA's existing reachability analysis mechanism without change (Section 5).

4. Case Study

In this section we present a small, but complete, example to illustrate the methodology outlined above.

Assume that an information security evaluator has been given the job of analysing an encryption device, designed to downgrade data being sent from a secure facility to a publicly-accessible network. The device's physical design is shown in Figure 1. Classified data enters via the high-security input on the left and exits via the low-security output on the right. The device also has an input from the low-security domain, used to receive acknowledgements that data packets sent to the low-security domain have been received successfully at the far end.

The device has two switches on its front panel. One switch allows the operator to choose whether encryption is turned on or off, because plaintext transmission is needed during establishment of the communications link, to allow configuration data to be sent to the receiving site. (An obvious procedural concern is the risk of the operator forgetting to switch the device to encryption mode after the link has been established—achieving absolute security is both a technological and procedural problem.) The second switch allows the operator to decide whether acknowledgement packets should be processed or not. If this switch is in the 'on' position the device will retransmit failed packets, but if the acknowledgement switch is in the 'off' position no retransmissions are performed, and any data received on the low-security input is ignored.

There are three main components within the device. On the left is a microprocessor which performs the buffering, encryption and retransmission tasks. At the top right is a simple timing circuit used to synchronise data packet outputs and subsequent acknowledgement packet inputs. It sends a timing signal to another circuit, at the bottom right, which processes acknowledgement packets received from the low-security domain. When a positive acknowledgement packet is received the acknowledgement circuit signals this event to the microprocessor. However, if the acknowledgement toggle switch is off the circuit permanently signals that a successful acknowledgement has occurred, irrespective of the packets received from the low-security input.

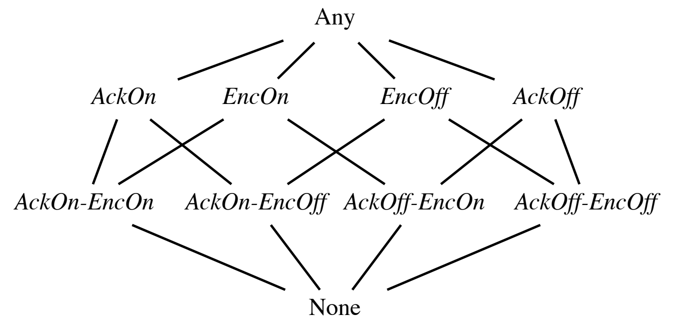

Figure 2. Operating mode lattice for the encryption device.

Thus, part of the overall device's functionality is implemented in hardware and part in software, so only a combined analysis of both can tell us about information flow from the high-security domain to the low-security one.

Based on a study of the detailed circuitry schematics, the security evaluator must also identify the operating modes relevant to this device. In this simple example the modes are obvious and are defined by the positions of the 'encryption' and 'acknowledgements' toggle switches on the device's front panel. A complete global mode lattice for all combinations of these switches is shown in Figure 2. (This is not to suggest that security evaluators are obliged to draw such a lattice, but they need to recognise that any non-disjoint modes they identify are related by subsumption.)

In the context of the significant modes identified, the security evaluator then needs to define intra-component connectivity for the device's hardware components by studying their detailed circuitry schematics. Here we assume this analysis shows that the synchronisation circuit in Figure 1 connects pin d to pin e in all modes (excluding the lattice's bottom), since this is necessary for data packets exiting the microprocessor to be forwarded to the low-security output port. The circuit also allows information flow from pin d to

```
         var buffer•
1.       while true do
2.           buffer ← pin_b;
3.           if pin_j then
4.               pin_c ← encrypt(buffer)
             else
5.               pin_c ← buffer
             fi;
6.           sleep(T);
7.           while not pin_m do
8.               pin_c ← buffer;
9.               sleep(T)
             od
         od
```

Figure 3. An insecure program for the encryption device's microprocessor.

pin g in all modes, because the passage of a data packet through the circuit affects the timing signal.

We also assume that the acknowledgement signal circuit allows information to flow from pin r to pin n only in Figure 2's '*AckOn*' modes, because the information contained within packets received from the low-security domain is forwarded to the microprocessor only when the acknowledgement switch is in the 'on' position. Similarly, the timing signal received on pin i influences the timing of the signal sent to pin n in these modes. Arguably, the circuit also allows information flow from pin p to pin n in '*AckOn*' modes only, since a negative acknowledgement signal can be sent along the n–m path only when the acknowledgement switch is 'on'. (A continuous series of positive acknowledgements sent along this path does not reveal the position of the switch.)

The challenge now is find out how the microprocessor exchanges information between its input and output pins. To do this we need to examine its software.

4.1 An Obviously Insecure Program

As an initial example, assume that the program loaded into the microprocessor is the one in Figure 3. This program contains an obvious design flaw which can be revealed using our technique.

The program is poll-driven and consists of an infinite loop (line 1), each iteration of which processes one data packet from the high-security domain, forwarding the result to the low-security domain. It reads data from the high-security input (line 2), and then samples the signal generated by the encryption toggle switch to decide how to process it (line 3). If the signal is high ('true') it calls an encryption function to encipher the data before forwarding it to the low-security domain (line 4). Otherwise it forwards the data without change (line 5). After the data has been sent the program waits for an appropriate length of time T until the acknowledgement packet can be expected (line 6). The program then samples the signal generated by the acknowledgement circuit (line 7). If the signal is low ('false') we assume that the packet was not received successfully, so the data is retransmitted (line 8), and the program waits for another acknowledgement (line 9), until a successful one is received. (The program assumes the acknowledgement circuit generates 'high' when the acknowledgement switch is off.)

Unfortunately, this program contains a serious security flaw. To reveal it our evaluation begins by parsing the program to produce its control flow graph (Figure 4). From this we want to extract all paths from high-security sources to low-security sinks. From Figure 1 we can see that the main source of high-security data for the program is pin b, and the only output that may lead to the low-security domain is pin c.

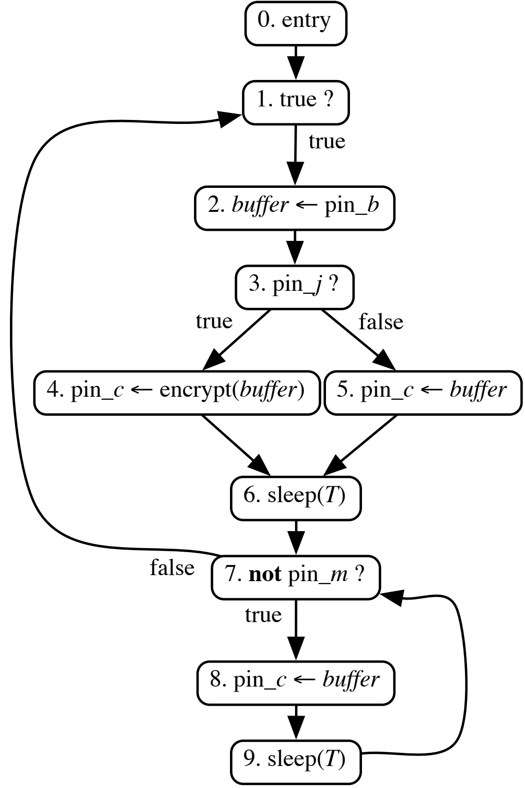

Figure 4. Control flow graph for the program in Figure 3.

We also produce the program dependence graph (Figure 5) so that we can tell which CFG paths involve end-to-end information flow. In Figure 5 we have distinguished two types of data dependence. As well as the usual explicit flow of information between two statements, we have highlighted 'downgraded' data dependences as those where the data transferred between pins is encrypted along the way. In this case the data flowing between pins b and c via the execution path beginning at statement 2 and ending at statement 4 is encrypted. Information flow along such paths is normally not a concern during security evaluations, provided we have faith in the strength of the encryption algorithm.

From Figure 4 we can extract a number of paths that link a statement that reads from pin b to one that writes to pin c. For instance, a short path is the following one from statement 2 to statement 5.

```
2.   buffer ← pin_b
3.   assert ¬pin_j
5.   pin_c ← buffer
```

The PDG in Figure 5 confirms that there is information flow between these statements, because statement 2 assigns to variable *buffer*, statement 5 accesses it, and there are no intervening assignments to the variable in the CFG [21].

Then a conventional CFG analysis tells us that the execution path condition is merely the predicate '$\neg$pin_j'. To relate this software condition to hardware modes we note that the signal on pin j can be low only when the device's encryption toggle switch is off. Thus, this path may be followed when the device is in any of the three '*EncOff*' modes in Figure 2. When the b–c path is added to the circuitry diagram it is therefore labelled with all of these modes.

The overall reachability analysis will then show that there is a direct information flow path through the encryption device, con-

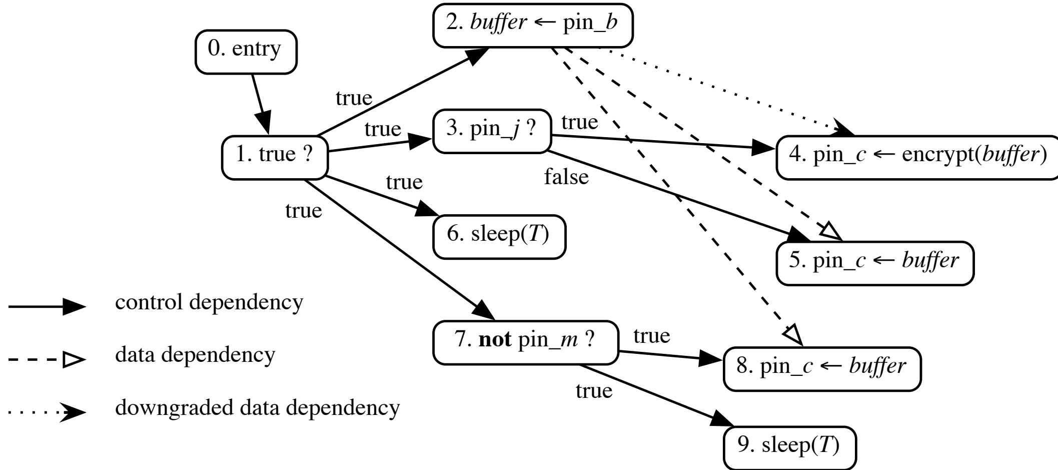

Figure 5. Program dependence graph for the program in Figure 3.

necting the high-security domain to the low-security domain, via pins a, b, c, d, e and f in all of the '*EncOff*' modes. However, this is not a security concern—it merely tells us that data flows freely through the device when it is *not* required to encrypt, as expected.

Of far greater concern is the following CFG-derived execution path, from statement 2 to statement 8.

2. *buffer* ← pin_b
3. **assert** pin_j
4. pin_c ← encrypt(*buffer*)
6. sleep(T)
7. **assert** ¬pin_m
8. pin_c ← *buffer*

Again the PDG confirms that there is a direct data dependence between the statements at this path's endpoints. However, the path condition is 'pin_j ∧ ¬pin_m'. Pin j is high when the encryption toggle switch is on, and pin m can be low only when the acknowledgement switch is on, because this switch setting allows negative acknowledgements from the low-security domain to be processed, so the only corresponding mode in the lattice is '*AckOn-EncOn*'. The combined hardware-software topological analysis will thus report that data may flow, *without change*, from pin a to pin f when the encryption device is meant to be encrypting data!

This discovery will alert the information security evaluator to the flaw in the program because there should not be any *non-downgraded* information flow from pin a to pin f when the device is in this mode. A closer study of the program in Figure 3 shows that the underlying problem is that when it retransmits an unacknowledged packet (line 8) it fails to check whether or not the retransmitted data is meant to be encrypted.

4.2 A Subtly Insecure Program

As another example, consider the program shown in Figure 6 executing in the same hardware environment. Like its predecessor this program is also flawed, but in a more subtle way.

Figure 6 shows that this program avoids the previous problem by checking to see whether or not data should be encrypted (line 3) every time it is sent to the low-security domain (lines 4 and 5). At first glance this would appear to solve the problem. Indeed, this program may run for a long time without any security issues arising. Nevertheless, it contains a flaw in the way it interacts with its hardware environment, which can be revealed by our approach.

```
    var buffer •
1.  while true do
2.      buffer ← pin_b;
        repeat
3.          if pin_j then
4.              pin_c ← encrypt(buffer)
            else
5.              pin_c ← buffer
            fi;
6.          sleep(T)
7.      until pin_m
    od
```

Figure 6. Another insecure program for the encryption device's microprocessor.

The program's control flow graph is shown in Figure 7 and its program dependence graph in Figure 8. Once again our process begins by extracting execution paths from the CFG that link high-security inputs (line 2 in Figure 6) to low-security outputs (lines 4 and 5).

For instance, the following short CFG-derived execution path from statement 2 to statement 5,

2. *buffer* ← pin_b
3. **assert** ¬pin_j
5. pin_c ← *buffer*

allows direct information flow from the high-security input to the low-security output according to the PDG, but does so only under path condition '¬pin_j', which the security evaluator would recognise means that the device is in one of the '*EncOff*' modes, in which case this behaviour is acceptable.

Similarly, the following execution path from statement 2 to statement 4,

2. *buffer* ← pin_b
3. **assert** pin_j
4. pin_c ← encrypt(*buffer*)

occurs only when path condition 'pin_j' is true, which means that the device must be in one of the '*EncOn*' modes, and the PDG confirms that there is only downgraded information flow through the device in this case. Again, this is an expected behaviour.

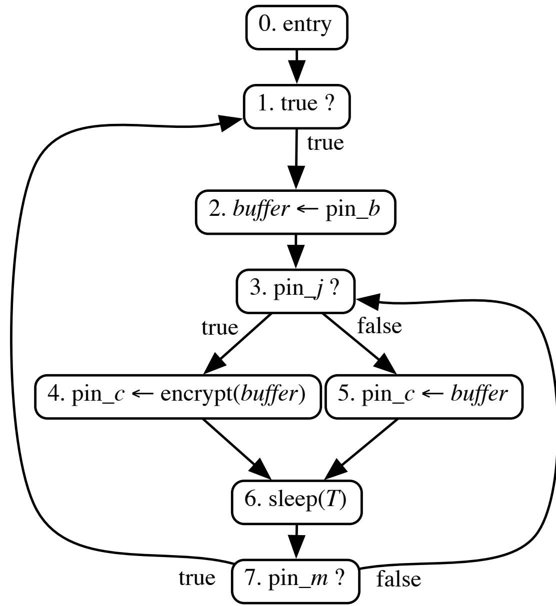

Figure 7. Control flow graph for the program in Figure 6.

However, a more interesting situation arises when we follow the loops around the control flow graph in Figure 7 to produce longer paths. In effect, this process involves unrolling the **repeat** loop in Figure 6. As in traditional test-case path coverage, this process is finite—we need only unroll loops enough times to ensure that all distinct control flow sequences through the loop body are generated. One such path begins at statement 2 and leads to statement 5, via statement 4.

2. *buffer* ← pin_b
3. **assert** pin_j
4. pin_c ← encrypt(*buffer*)
6. sleep(T)
7. **assert** ¬pin_m
3. **assert** ¬pin_j
5. pin_c ← *buffer*

The condition required to follow this execution path is 'pin_$j_1 \wedge$ ¬pin_$m \wedge$ ¬pin_j_2', where the subscripts are used to distinguish different signal samples read from pin j. It's important when analysing embedded code to recognise that consecutive inputs from the same microprocessor pin may return different data values. Unlike variables controlled by the program, signals on hardware pins can change value without the program doing anything. Therefore, although the path condition says that the value on pin j must be *both* high and low, this is not contradictory—it merely means that the hardware environment is assumed to have changed the control signal on this pin between samples.

Given this software path condition, the security evaluator is now obliged to find a corresponding hardware mode in which this situation may arise. We know that pin m can be low only when a negative acknowledgement has been received (or a timeout has occurred), so this tells us that the acknowledgements toggle switch must be in the 'on' position. However, there is no single hardware state in which pin j can be both low and high simultaneously. This means that this path occurs during a period when the value on this pin changes, i.e., when someone has moved the device's encryption toggle switch from 'on' to 'off'. Using the lattice in Figure 2 we can see that the only mode (apart from the lattice's top) consistent with this path condition is *AckOn*. Interestingly, neither of the inferior modes *AckOn-EncOn* and *AckOn-EncOff* can accommodate this path condition on their own.

Thus, when the potential *b*–*c* connection is added to the *AckOn* mode, the automated reachability analysis will report that there may be direct information flow through the encryption device, from pin a to pin f, when the acknowledgements switch is on, but without a guarantee that the encryption switch is off!

Faced with this undesirably liberal form of information flow, the security evaluator will inspect the hardware-software path generated to determine the problem's cause. In this case it is due to the fact that when a data packet is received it is always stored in unencrypted form (line 2 in Figure 6), even if the encryption toggle switch is on. Therefore, although the packet is forwarded to the low-security domain in encrypted form on the first occasion (line 4), this particular path shows that the packet may be retransmitted without encryption (line 5), if a negative acknowledgement is received and the encryption switch has subsequently been turned to the 'off' position. Thus data stored in the device, intended to be forwarded in encrypted form only, can be released to the low-security domain without encryption.

Importantly, this discovery would not have been made by using program dependence graphs alone. Conventional PDG analysis would assume that the assignments in the program code to pin c are unimportant, since no part of the program subsequently reads from this 'variable'. It is for this reason that we use control flow graph execution paths, which retain all statements executed. In embedded programs outputs to the hardware environment may have major significance for information flow *outside* the program.

4.3 A Secure Program

Finally, for completeness, Figure 9 presents a secure program for the encryption device's microprocessor. In this case the data is stored in the buffer in encrypted form if the encryption switch is on (line 4). Thus, if retransmission becomes necessary the data will always be resent (line 5) in a form that reflects the state of the encryption toggle switch when the data was first received from the high-security domain.

5. Implementation

Based on the principles outlined above we have implemented an embedded program parser that extracts information flow paths and associated path conditions for use in information security evaluations of communications devices. The embedded program is assumed to be written in Custom Computer Services' C dialect for Programmable Integrated Circuit microcontrollers.[5]

The tool itself is implemented in Microsoft C♯. Given an embedded C program it "inlines" function calls and unrolls loops enough times to expose all distinct control flow paths through the loop's body. It then constructs the control flow graph, converts the code to static single assignment form (including ϕ functions), and constructs the program dependence graph. During this process, any input-output pairs that are connected via data and/or control edges in the PDG are noted.

For every input in such a pair, all of the possible paths from the program's entry point to the input are then calculated, using the CFG. (The full path from the program's entry point is needed in order to calculate the path condition [21].) Next all possible paths between each input-output pair are calculated. By combining the paths from the entry point to the relevant input-output pair, a set of all paths from the start of the program to every output, via an input, is produced. The path condition is then calculated for each path.

[5] `http://www.ccsinfo.com/`

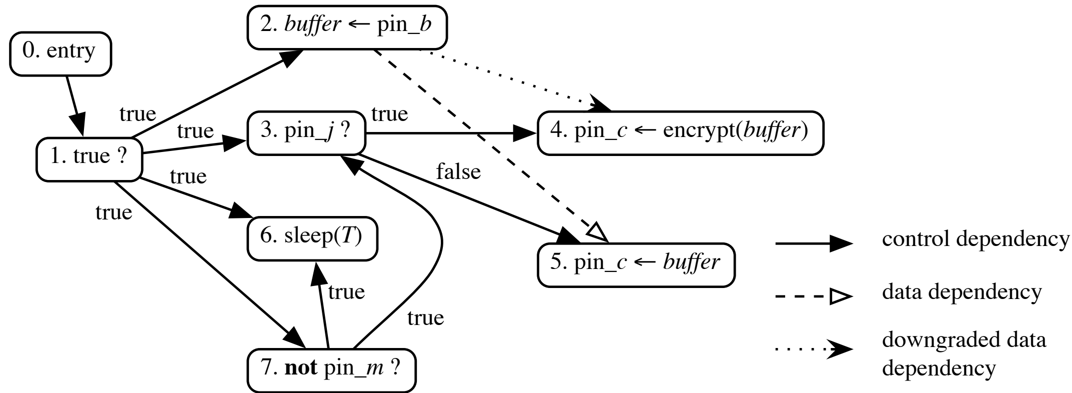

Figure 8. Program dependence graph for the program in Figure 6.

```
    var buffer •
1.  while true do
2.      buffer ← pin_b;
3.      if pin_j then
4.          buffer ← encrypt(buffer)
        fi;
        repeat
5.          pin_c ← buffer;
6.          sleep(T)
7.      until pin_m
    od
```

Figure 9. A secure program for the encryption device's microprocessor.

For instance, given an embedded C version of the program from Figure 6, the tool generated 15 distinct paths and path conditions, including those discussed in Section 4.2. (This many paths cannot be seen by mere inspection of Figure 6. The tool represents the program in an intermediate language form, to support planned extensions for modelling asynchronous interrupts, whose information flow effects are describable at a low level only [20]. Also, some generated paths are prefixes of longer paths that contain multiple output statements.)

Armed with this information, it is then a straightforward exercise to enter these paths as intra-processor connections in a SIFA [13] model of the entire encryption device. Furthermore, since SIFA allows free-form text to identify hardware modes, we can efficiently enter sets of modes from the lattice (Figure 2) through appropriate naming conventions. For this example we used 'Any mode' to denote all modes except None, 'M modes' plural to denote mode M and all of its non-None inferiors, and 'M mode' singular to denote mode M only. We also explicitly annotated paths that downgrade data.

The resulting SIFA model and analysis results are shown in Figure 10, assuming that the microprocessor is running the program from Figure 6. The tool successfully identifies the information flow channel through the device and the three hardware modes in which the embedded software may allow data to pass. These include the *EncOff* modes, and downgraded data flow in the *EncOn* modes, both of which are expected behaviours of the device. More importantly, the analysis reveals the undesired information flow allowed when the device is in *AckOn* mode, without requiring that the encryption switch is off.

Of course, this outcome is obvious for this small example. However, when dealing with circuit diagrams containing dozens of components, and embedded programs of hundreds of lines, such automated support is invaluable for information security evaluators.

6. Conclusion

Analysing information flow through security-critical electronic devices is a challenging task, and their growing reliance on embedded software makes the job even harder. Here we have shown how traditional program compiler technology can be integrated with hardware analysis techniques to produce a seamless information flow evaluation methodology for embedded systems.

Future work will focus on ways of handling other programming constructs peculiar to embedded code. In particular, since embedded programs are usually weakly-typed, to allow bit- and byte-level manipulation of integers and other 'atomic' values, we are currently adapting analysis techniques for arrays to the problem of tracing information flow through compound data structures.

Acknowledgments

This research was funded by the Defence Signals Directorate and the Australian Research Council via ARC Linkage-Projects Grant LP0347620. We wish to thank Brian Palm, Vicky Briant and Peter Young for their ongoing support for this research, and the anonymous reviewers for their helpful comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, California, USA, 1986.

[2] M. Avvenuti, C. Bernardeschi, and N. De Francesco. Java bytecode verification for secure information flow. *ACM SIGPLAN Notices*, 38(12):20–27, December 2003.

[3] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90), New York, USA, June 20–22, 1990*, pages 257–271. ACM, 1990.

[4] K. Banks. Tips for checking schematics. *Embedded Systems*, 16(6):36–38, June 2003.

[5] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, February 2006.

[6] D. Clark, C. Hankin, and S. Hunt. Information flow for ALGOL-like languages. *Computer Languages*, 28(1):3–28, April 2002.

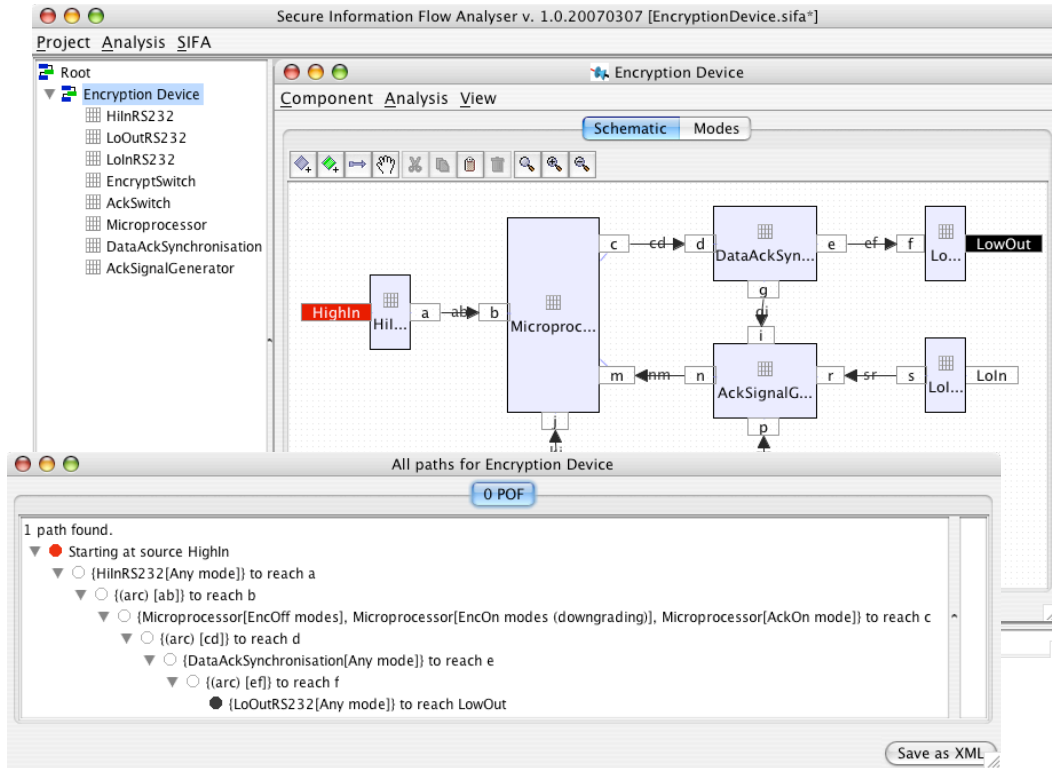

Figure 10. Automated analysis results for the encryption device, using paths from the program in Figure 6.

[7] A. W. Dent and J. Malone-Lee. The physically observable security of signature schemes. In N. P. Smart, editor, *Cryptography and Coding—Tenth IMA International Conference*, volume 3796 of *Lecture Notes in Computer Science*, pages 220–232, Cirencester, United Kingdom, 19–21 December 2005. Springer-Verlag, Berlin.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[9] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE'92)*, pages 392–411, Melbourne, Australia, 11–15 May 1992. ACM Press, New York.

[10] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, May 2000.

[11] V. Lotz, V. Kessler, and G. H. Walter. A formal security model for microprocessor hardware. *IEEE Transactions on Software Engineering*, 26(8):702–712, August 2000.

[12] A. Mahalingam, B. P. Butz, and M. Duarte. An intelligent circuit analysis module to analyze student queries in the Universal Virtual Laboratory. In W. Oakes, D. Voltmer, and C. Yokomoto, editors, *Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference (FIE'05)*, pages F4E–1–F4E–6, Indianapolis, USA, 19–22 October 2005. Institute of Electrical and Electronics Engineers, New Jersey, USA.

[13] T. McComb and L. P. Wildman. SIFA: A tool for evaluation of high-grade security devices. In C. Boyd and J. Nieto, editors, *Proceedings of the Tenth Australasian Conference on Information Security and Privacy (ACISP 2005)*, volume 3574 of *Lecture Notes in Computer Science*, pages 230–241, Brisbane, Australia, 4–6 July 2005. Springer-Verlag, Berlin.

[14] T. McComb and L. P. Wildman. Verifying abstract information flow properties in fault tolerant security devices. In Z. Liu and J. He, editors, *Proceedings of the Eighth International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *Lecture Notes in Computer Science*, pages 621–638, Macao, China, 1–3 November 2006. Springer-Verlag, Berlin.

[15] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[16] K. R. O'Neill, M. R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 190–201, Venice, Italy, 5–7 July 2006. IEEE Computer Society, Washington, DC, USA.

[17] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.

[18] A. J. Rae and C. J. Fidge. Information flow analysis for fail-secure devices. *The Computer Journal*, 48(1):17–26, January 2005.

[19] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):1–15, January 2003.

[20] B. Schlich, M. Rohrbach, M. Weber, and S. Kowalewski. Model checking software for microcontrollers. Technical Report AIB-2006-11, Department of Computer Science, RWTH Aachen University, Germany, 2006.

[21] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology*, 15(5):410–457, October 2006.

[22] The Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology Security Evaluation*. International Organization for Standardization, Geneva, August 1999.