Automatic Derivation of Platform Noninterference Properties

Oliver Schwarz[1,2] and Mads Dam[2]

[1] SICS Swedish ICT, Kista, Sweden
[2] KTH Royal Institute of Technology, Stockholm, Sweden
{oschwarz,mfd}@kth.se

Legal Notice. This is the author version of the correspondent paper published in “Software Engineering and Formal Methods”, the proceedings of SEFM 2016 (editors: Rocco De Nicola, Eva Kühn), Springer LNCS 9763. The publisher is Springer International Publishing Switzerland. The final publication is available at Springer via
http://dx.doi.org/10.1007/978-3-319-41591-8_3.

Abstract. For the verification of system software, information flow properties of the instruction set architecture (ISA) are essential. They show how information propagates through the processor, including sometimes opaque control registers. Thus, they can be used to guarantee that user processes cannot infer the state of privileged system components, such as secure partitions. Formal ISA models - for example for the HOL4 theorem prover - have been available for a number of years. However, little work has been published on the formal analysis of these models. In this paper, we present a general framework for proving information flow properties of a number of ISAs automatically, for example for ARM. The analysis is represented in HOL4 using a direct semantical embedding of noninterference, and does not use an explicit type system, in order to (i) minimize the trusted computing base, and to (ii) support a large degree of context-sensitivity, which is needed for the analysis. The framework determines automatically which system components are accessible at a given privilege level, guaranteeing both soundness and accuracy.

Keywords: Instruction set architectures, ARM, MIPS, noninterference, information flow, theorem proving, HOL4

1 Introduction

From a security perspective, isolation of processes on lower privilege levels is one of the main tasks of system software. More and more vulnerabilities discovered in operating systems and hypervisors demonstrate that assurance of this isolation is far from given. That is why an increasing effort has been made to formally verify system software, with noticeable progress in recent years [10,14,16,6,1].

However, system software depends on hardware support to guarantee isolation. Usually, this involves at least the ability to execute code on different privilege levels and with basic memory protection. Kernels need to control access to their own code and data and to critical software, both in memory and as content of registers or other components. Moreover, they need to control the management of the access control itself. For the correct configuration of hardware, it is essential to understand how and under which circumstances information flows through the system. Hardware must comply to a contract that kernels can rely on. In practice, however, information flows can be indirect and hidden. For example, some processors automatically set control flags on context switches that can later be used by unprivileged code to see if neighbouring processes have been running or to establish a covert channel [19]. Such attacks can be addressed by the kernel, but to that end, kernel developers need machinery to identify the exact components available to unprivileged code, and specifications often fail to provide this information in a concise form. When analysing information flow, it is insufficient to focus on direct register and memory access. Confidentiality, in particular, can be broken in more subtle ways. Even if direct reads from a control flag are prevented by hardware, the flag can be set as an unintended side effect of an action by one process and later influence the behaviour of another process, allowing the latter to learn something about the control flow of the former.

In this paper we present a framework to automate information flow analysis of instruction set architectures (ISAs) and their operational semantics inside the interactive theorem prover HOL4 [11]. We employ the framework on ISA models developed by Fox et al. [7] and verify *noninterference*, that is, that secret (*high*) components can not influence public (*low*) components. Besides an ISA model, the input consists of desired conditions (such as a specific privilege mode) and a candidate labelling, specifying which system components are already to be considered as low (such as the program counter) and, implicitly, which components might possibly be high. The approach then iteratively refines the candidate labelling by downgrading new components from high to low until a proper noninterference labelling is obtained, reminiscent of [12]. The iteration may fail for decidability reasons. However, on successful termination, both soundness and accuracy are guaranteed unless a warning is given indicating that only an approximate, sound, but not necessarily accurate solution has been found.

What makes accurate ISA information flow analysis challenging is not only the size and complexity of modern instruction sets, but also particularities in semantics and representation of their models. For example, arithmetic operations (e.g., with bitmasks) can cancel out some information flows and data structures can contain a mix of high and low information. Modification of the models to suit the analysis is error prone and requires manual effort. Automatic, and provably correct, preprocessing of the specifications could overcome some, but not all, of those difficulties, but then the added value of standard approaches such as type systems over a direct implementation becomes questionable. By directly embedding noninterference into HOL4, we can make use of machinery to address the discussed difficulties and at the same time we are able to minimize the trusted computing base (TCB), since the models, the preprocessing and the actual reasoning are all implemented/represented in HOL4. Previous work on HOL4 noninterference proofs for ISA models [13] had to rely on some manual proofs, since its compositional approach suffered from the lack of sufficient context in some cases (e.g., the secrecy level of a register access in one step can depend on location lookups in earlier steps). In contrast, the approach suggested in the present paper analyses ISAs one instruction at a time, allowing for accuracy and automation at the same time. However, since many instructions involve a number of subroutines, this instruction-wide context introduces complexity challenges. We address those by unfolding definitions of transitions in such a way that their effects can be extracted in an efficient manner.

Our analysis is divided into three steps: (i) *rewriting* to unfold and simplify instruction definitions, (ii) the *actual proof attempt*, and (iii) automated counterexample-guided *refinement of the labelling* in cases where the proof fails. The framework can with minor adaptations be applied to arbitrary HOL4 ISA models. We present benchmarks for ARMv7 and MIPS. With a suitable labelling identified, the median verification time for one ARMv7 instruction is about 40 seconds. For MIPS, the complete analysis took slightly more than one hour and made configuration dependencies explicit that we had not been aware of before. We report on the following contributions: (i) a backward proof tactic to automatically verify noninterference of HOL4 state transition functions, as used in operational ISA semantics; (ii) the automated identification of sound and accurate labellings; (iii) benchmarks for the ISAs of ARMv7-A and MIPS, based on an SML-implementation of the approach.

2 Processor Models

2.1 ISA Models

In the recent years, Fox et al. have created ISA models for x86-64, MIPS, several versions of ARM and other architectures [8,7]. The instruction sets are modelled based on official documentations and on the abstraction level of the programmer's view, thus being agnostic to internals like pipelines. The newest models are produced in the domain-specific language L3 [7] and can be exported to the interactive theorem prover HOL4. Our analysis targets those purely-functional HOL4 models for single-core systems. An ISA is formalized as a state transition system, with the machine state represented as record structure (on memory, registers, operational modes, control flags, etc.) and the operational semantics as functions (or *transitions*) on such states. The top-level transition `NEXT` processes the CPU by one instruction. While L3 also supports export to HOL4 definitions in monadic style, we focus our work on the standard functional representation based on let-expressions. States resulting from an *unpredictable* (i.e., underspecified) operation are tagged with an exception marker (see Section 7 for a discussion).

2.2 Notation

A *state* $s = \{C_1 := c_1, C_2 := c_2, \ldots\}$ is a record, where the fields $C_1, C_2, \ldots$ depend on the concrete ISA. As a naming convention, we use R_i for fields that are records themselves (such as control registers) and F_i for fields of a function/mapping type (such as general purpose register sets). The *components* of a state are all its fields and subfields (in arbitrary depth), as well as the single entries of the state's mappings. The value of field C in s is derived by $s.C$. An update of field C in s with value c is represented as $s[C := c]$. Similarly, function updates of F in location l by value v are written as $F[l := v]$. Conditionals and other case distinctions are written as $\mathbb{C}(b, a_1, a_2, \ldots, a_k)$, with b being the selector and $a_1, a_2, \ldots, a_k$ the alternatives. A transition Φ transforms a pre-state s into a return-value v and a post-state s', formally $\Phi s = (v, s')$. Usually, a transition contains subtransitions $\Phi_1, \Phi_2, \ldots, \Phi_n$, composed of some structure ϕ of abstractions, function applications, case distinctions, sequential compositions and other semantic operators, so that $\Phi s = \phi(\Phi_1, \Phi_2, \ldots, \Phi_n)s$. Transition definitions can be recursively unfolded: $\phi(\Phi_1, \ldots, \Phi_n)s = \phi(\phi_1(\Phi_{1,1}, \ldots, \Phi_{1,m}), \ldots, \Phi_n)s = \ldots = \vec{\phi}s$, where $\vec{\phi}$ is the completely unfolded transition, called the *evaluated form*. For the transitions of the considered instruction sets, unfolding always terminates. Note that '=' is used here for the equivalence of states, transitions or values, not for the syntactical equivalence of terms. Below we give the definition of the ARMv7-NOOP-instruction and its evaluated (and simplified) form:

$$\begin{aligned}
&\texttt{dfn}'\texttt{NoOperation}\ s\\
=\ &\texttt{BranchTo}(s.\texttt{REG RName\_PC}\ + \mathbb{C}(\texttt{FST}\ (\texttt{ThisInstrLength}\ ()\ s) = 16, 2, 4))\ s\\
=\ &((), s[\texttt{REG} := s.\texttt{REG}[\texttt{RName\_PC} := s.\texttt{REG RName\_PC}\ + \mathbb{C}(s.\texttt{Encoding} = \texttt{Thumb}, 2, 4)]])
\end{aligned}$$

NOOP branches to the current program counter (s.REG RName_PC) plus some offset. The offset depends on the current instruction length, which in turn depends on the current encoding. Here, FST selects the actual return value of the ThisInstrLength transition, ignoring its unchanged post-state.

2.3 Memory Management

For simplicity, our analysis focuses on core-internal flows (e.g., between registers) and abstracts away from the concrete behaviour of the memory subsystem (including address translation, memory protection, caching, peripherals, buses, etc.). Throughout the course of the - otherwise core internal - analysis, a contract on the memory subsystem is assumed that then allows the reasoning on global properties. The core can communicate with the memory subsystem through an interface, but never directly accesses its internal state. The interface expects inputs like the type of access (read, fetch, write, . . .), the virtual address, the privilege state of the processor, and other parameters. It updates the state of the memory subsystem and returns a success or error message along with possibly read data. While being agnostic about the concrete behaviour of the memory subsystems, we assume that there is a secure memory configuration $\mathcal{P}_m$, restricting unprivileged accesses, e.g., through page table settings. Furthermore, we assume the existence of a low-equivalence relation $\mathcal{R}_m$ on pairs of memory subsystems.

Typically, two memories in $\mathcal{R}_m$ would agree on memory content accessible in an unprivileged processor mode. When in unprivileged processor mode and starting from secure memory configurations, transitions on memory subsystems are assumed to maintain both the memory relation and secure configurations. Consider an update of state s assigning the sum of the values of register y and the memory at location a to register x, slightly simplified: $s[x := s.y + \mathtt{read}(a, s.\mathtt{mem})]$. Since `read` - as a function of the memory interface - satisfies the constraints above, for two pre-states s_1 and s_2 satisfying $\mathcal{P}_m s_1.\mathtt{mem} \wedge \mathcal{P}_m s_2.\mathtt{mem} \wedge \mathcal{R}_m(s_1.\mathtt{mem}, s_2.\mathtt{mem})$, we can infer that `read` will return the same value or error. Overall, with preconditions met, two states that agree on x, y, and the low parts of the memory before the computation, will also agree after the computation. That is, as long as `read` fulfils the contract, the analysis of the core (and in the end the global analysis) does not need to be concerned with details of the memory subsystem.

3 ISA Information Flow Analysis

3.1 Objectives

Consider an ISA model with an initial specification determining some preconditions (e.g., on the privilege mode) and some system components, typically only the program counter, that are to be regarded as observable (or *low*) by some given actor. If there is information flow from some other component (say, a control register) to some of these initially-low components, this other component must be regarded as observable too for noninterference to hold. The objective of the analysis is to identify all these other components that are observable due to their direct or indirect influence on the given low components.

A *labelling* $\mathcal{L}$ assigns to each atomic component (component without subcomponents) a label, high or low. [3] It is *sound* if it does not mark any component as high that can influence, and hence pass information to, a component marked as low. In the refinement order the labelling $\mathcal{L}'$ refines $\mathcal{L}$ ($\mathcal{L} \sqsubseteq \mathcal{L}'$), if low components in $\mathcal{L}$ are low also in $\mathcal{L}'$. The labelling $\mathcal{L}$ is *accurate*, if $\mathcal{L}$ is minimal in the refinement order such that $\mathcal{L}$ is sound and refines the initial labelling.

Determining whether a labelling is accurate is generally undecidable. Suppose $\mathbb{C}(P(x), s.C, 0)$ is assigned to a low component. Deciding whether C needs to be deemed low requires deciding whether there is some valid instantiation of x, such that $P(x)$ holds, which might not be decidable. However, it appears that in many cases, including those considered here, accurate labellings are feasible. In our approach we check the necessity of a label refinement by identifying an actual flow from the witness component to some low component. We cannot guarantee that this check always succeeds, for undecidability reasons. If it does not, the tool still tries to refine the low equivalence and a warning that the final relation may no longer be accurate is generated. For the considered case studies the tool always finds an accurate labelling, which is then by construction unique.

[3] We have not found a use for ISA security lattices of finer granularity.

Labellings correspond to low-equivalence relations on pairs of states, relations that agree on all low components including the memory relation $\mathcal{R}_m$ and leave all other components unrestricted. *Noninterference* holds if the only components affecting the state or any return value are themselves low. Formally, assume the two pre-states s_1 and s_2 agree on the low-labelled components, expressed by a low-equivalence relation $\mathcal{R}$ on those states. Then, for a given transition Φ and preconditions $\mathcal{P}$, noninterference $\mathcal{N}(\mathcal{R}, \mathcal{P}, \Phi)$ holds if after Φ the post-states are again in $\mathcal{R}$ and the resulting return values are equal:

$$\begin{aligned}&\mathcal{N}(\mathcal{R}, \mathcal{P}, \Phi) := \forall s_1, s_2, v_1, v_2, t_1, t_2 :\\ &\ ((v_1, t_1) = \Phi s_1) \wedge ((v_2, t_2) = \Phi s_2) \wedge \mathcal{R}(s_1, s_2) \wedge \mathcal{P} s_1 \wedge \mathcal{P} s_2\\ &\ \Rightarrow \mathcal{R}(t_1, t_2) \wedge (v_1 = v_2)\end{aligned}$$

Preconditions on the starting states can include architecture properties (version number, present extensions, etc.), a secure memory configuration and a specification of the privilege level. In our framework the user defines relevant preconditions and an initial low-equivalence relation $\mathcal{R}_0$ for an input ISA. The goal of the analysis is to statically and automatically find an accurate refinement of $\mathcal{R}_0$ so that noninterference holds for $\Phi = \texttt{NEXT}$. The analysis yields the final low-equivalence relation, the corresponding HOL4 noninterference theorem demonstrating the soundness of the relation, and a notification of whether the analysis succeeded to establish a guarantee on the relation's accuracy. The proof search is not guaranteed to terminate successfully, but we have found it robust enough to reliably produce accurate output on ISA models of considerable complexity (see Section 5). We do not treat timing and probabilistic channels and leave safety-properties about unmodified components for future work.

3.2 Challenges

Our goal is to perform the analysis from an initial, user-supplied labelling on a standard ISA with minimal user interaction. In particular, we wish to avoid user supplied label annotations and error-prone manual rewrites of the ISA specification, that a type-based approach might depend on to eliminate some of the complications specific to ISA models. Instead, we address those challenges with symbolic evaluation and the application of simplification theorems. Since both are available in HOL4, and so are the models, we verify noninterference in HOL4 directly. This also frees us from external preprocessing and soundness proofs, thus minimizing the TCB. Below, we give examples for common challenges.

Representation The functional models that we use represent register sets as mappings. Static type systems for (purely) functional languages [9,17] need to assign secrecy levels uniformly to all image values, even if a mapping has both public and secret entries. Adaptations of representation and type system might allow to type more accurately for lookups on constant locations. But common lookup patterns on locations represented by variables or complex terms would require a preprocessing that propagates constraints throughout large expressions.

Semantics Unprivileged ARMv7 processes can access the current state of the control register CPSR. The ISA specifies to (i) map all subcomponents of the control register to a 32-bit word and (ii) apply the resulting word to a bitmask. As a result, the returned value does actually not depend on all subcomponents of the CPSR, even though all of them were referred to in the first step. For accuracy, an actual understanding of the arithmetics is required.

Context-sensitivity Earlier work on ISA information flow [13] deals with ARM's complex operational semantics in a stepwise analysis, focusing on one subprocedure at a time. This allows for a systematic solution, but comes with the risk of insufficient context. For example, when reading from a register, usually two steps are involved: first, the concrete register identifier with respect to the current processor mode is looked up; second, the actual reading is performed. Analysing the reading operation in isolation is not accurate, since the lack of constraints on the register identifier would require to deem all registers low. In order to include restrictions from the context, [13] required a number of manual proofs. To avoid this, we analyse entire instructions at a time, using HOL4's machinery to propagate constraints.

4 Approach

We are not the first to study (semi-)automated hardware verification using theorem proving. As [5] points out for hardware refinement proofs, a large share of the proof obligations can be discharged by repeated unfolding (rewriting) of definitions, case splits and basic simplification. While easy to automate, these steps lead easily to an increase in complexity. The challenge, thus, is to find efficient and effective ways of rewriting and to minimize case splits throughout the proof. Our framework traverses the instruction set instruction by instruction, managing a task queue. For each instruction, three steps are performed: (i) rewriting/unfolding to obtain evaluated forms, (ii) attempting to prove noninterference for the instruction, (iii) on failure, using the identified counterexample to refine the low-equivalence relation. This section details those steps. After each refinement, the instructions verified so far are re-enqueued. The steps are repeated until the queue is empty and each instruction has successfully been verified with the most recent low-equivalence relation. Finally, noninterference is shown for `NEXT`, employing all instruction lemmas, as well as rewrite theorems for the fetch and decode transitions. Soundness is inherited from HOL4's machinery. Accuracy is tracked by the counterexample verification in step (iii).

4.1 Rewriting towards an Evaluated Form

The evaluated form of instructions is obtained through symbolic evaluation. Starting from the definition of a given transition, (i) let-expressions are eliminated, (ii) parameters of subtransitions are evaluated (in a call-by-value manner), (iii) the subtransitions are recursively unfolded by replacing them with their respective evaluated forms, (iv) the result is normalized, and (v) in a few cases substituted with an abstraction. Normalization and abstraction are described below. For the first three steps we reuse evaluation machinery from [7] and extend it, mainly to add support for automated subtransition identification and recursion. Preconditions, for example on the privilege level, allow to reduce rewriting time and the size of the result. Since they can become invalid during instruction execution, they have to be re-evaluated for each recursive invocation. Throughout the whole rewriting process, various simplifications are applied, for example on nested conditional expressions, case distinctions, words, and pairs, as well as conditional lifting, which we motivate below. For soundness, all steps produce equivalence theorems.

Step Library The ISA models are provided together with so-called *step libraries*, specific to every architecture [7]. They include a database of precomputed rewrite theorems, connecting transitions to their evaluated forms. Those theorems are computed in an automated manner, but are guided manually. Our tool is able to employ them as hints, as long as their preconditions are not too restrictive for the general security analysis. Otherwise, we compute the evaluated forms autonomously. Besides instruction specific theorems, we use some datatype specific theorems and general machinery from [7].

Conditional Lifting Throughout the rewriting process, the evaluated forms of two sequential subtransitions might be composed by passing the result of the first transition into the formal parameters of the second. This often leads to terms like $\gamma(s) := \mathbb{C}(b, s[C_1 := c_1], s[C_2 := c_2]).C_3$. However, in order to derive equality properties in the noninterference proof (e.g., $[s_1.C_3 = s_2.C_3] \vdash \gamma(s_1) = \gamma(s_2)$) or to check validity of premises (e.g., $\gamma(s) = 0$), conditional lifting is applied:

$$\begin{aligned}
\gamma(s) &= \mathbb{C}(b, s[C_1 := c_1], s[C_2 := c_2]).C_3 && \text{lifting}\\
&= \mathbb{C}(b, (s[C_1 := c_1]).C_3, (s[C_2 := c_2]).C_3) && \text{simplifying}\\
&= \mathbb{C}(b, s.C_3, s.C_3) && \text{merging}\\
&= s.C_3
\end{aligned}$$

To mitigate exponential blow-up, conditional lifting should only be applied where needed. For record field accesses we do this in a top-down manner, ignoring fields outside the current focus. For example, in $\gamma(s)$ there is no need to process c_1 at all, even in cases where c_1 itself is a conditional expression.

Normalization With record field accesses being so critical for performance, both rewriting and proof benefit from (intermediate) evaluated forms being normalized. A state term is *normalized* if it only consists of record field updates to a state variable s, that is, it has the form

$$s[C_1 := c_1, \ldots, C_n := c_n, R_1 := s.R_1[C_{1,1} := c_{1,1}, \ldots, C_{1,k} := c_{1,k}], \ldots].$$

For a state term τ updating state variable s in the fields $C_1, \ldots, C_n$ with the values $c_1, \ldots, c_n$, we verify the normalized form in a forward construction (omitting subcomponents here and below for readability; they are treated analogously):

$$\begin{aligned} \tau &= \tau[C_1 := \tau.C_1, \ldots, C_n := \tau.C_n] && (1)\\ &= s[C_1 := \tau.C_1, \ldots, C_n := \tau.C_n] && (2)\\ &= s[C_1 := c_1, \ldots, C_n := c_n] && (3) \end{aligned}$$

We significantly improve proof performance with the abstraction of complex expressions by showing (1) independently of the concrete τ and (2) independently of the values of the updates, both those inside τ and those applied to τ. We obtain $c_1, \ldots, c_n$ by similar means to those shown in the lifting example of γ above.

In [7], both conditional lifting and normalization are based on the precomputation of datatype specific lifting and unlifting lemmas for updates. Our procedures are largely independent of record types and update patterns. However, because of the performance benefits of [7], we plan to generalize/automate their normalization machinery or combine both approaches in future work.

Abstracted Transitions Even with normalization, the specification of a transition grows quickly when unfolding complex subtransitions, especially for loops. We therefore choose to abstract selected subtransitions. To this end, we substitute their evaluated forms with terms that make potential flows explicit, but abstract away from concrete specifications. Let the normalized form of transition Φ be $\vec{\phi} s = (\beta(s), s[C_1 := \gamma_1(s), \ldots, C_n := \gamma_n(s)])$. The values of all primitive state updates $\gamma_1(s), \ldots, \gamma_n(s)$ on s and the return value $\beta(s)$ of Φ are substituted with new function constants $f_0, f_1, \ldots, f_n$ applied to relevant state components actually accessed instead of to the entire state:

$$\begin{aligned} \Phi s = \vec{\phi} s = (&f_0(s.C_{0,1}, \ldots, s.C_{0,k_0}),\\ &s[C_1 := f_1(s.C_{1,1}, \ldots, s.C_{1,k_1}), \ldots, C_n := f_n(s.C_{n,1}, \ldots, s.C_{n,k_n})]) \end{aligned}$$

Except for situations that suggest the need for a refinement of the low-equivalence relation, $f_0, \ldots, f_n$ do not need to be unfolded in the further processing of Φ. Low-equivalence of the post-states can be inferred trivially:

$$[(s_1.C_{1,1} = s_2.C_{1,1}) \wedge \ldots] \vdash f_1(s_1.C_{1,1}, s_1.C_{1,2}, \ldots) = f_1(s_2.C_{1,1}, s_2.C_{1,2}, \ldots))$$

To avoid accuracy losses in cases where $\vec{\phi}$ mentions components that neither return value nor low components actually depend on, we unfold abstractions as last resort before declaring a noninterference proof as failed.

4.2 Backward Proof Strategy

Having computed the evaluated form for an instruction Φ, we proceed with the verification attempt of $\mathcal{N}(\mathcal{R}, \mathcal{P}, \Phi)$ through a backward proof, for the user-provided preconditions $\mathcal{P}$ and the current low-equivalence relation $\mathcal{R}$. The sound backward proof employs a combination of the following steps:

- **Conditional Lifting:** Especially in order to resolve record field accesses on complex state expressions, we apply conditional lifting in various scopes (record accesses, operators, operands) and degrees of aggressiveness.
- **Equality of Subexpressions:** Let F be a functional component and n and m be two variables ranging over $\{0, 1, 2\}$. The equality

  $$\begin{aligned} &\mathbb{C}(n = 2, 0, s_1.F(\mathbb{C}(n, a, b, c))) + s_1.F(\mathbb{C}(m, a, b, a)) \\ = &\mathbb{C}(n = 2, 0, s_2.F(\mathbb{C}(n, a, b, c))) + s_2.F(\mathbb{C}(m, a, b, a)) \end{aligned}$$

  can be established from the premises $s_1.F(a) = s_2.F(a)$ and $s_1.F(b) = s_2.F(b)$ by lifting the distinctions on n and m outwards or - alternatively - by case splitting on n and m. Either way, equality should be established for each summand separately, in order to limit the number of considered cases to $3 + 3$ instead of 3×3. Doing so in explicit subgoals also helps in discarding unreachable cases, such as the one where c would be chosen. We identify relevant expressions via pre-defined and user-defined patterns.
- **Memory Reasoning:** Axioms and derived theorems on noninterference properties of the memory subsystem and maintained invariants are applied.
- **Simplifications:** Throughout the whole proof process, various simplifications take effect, for example on record field updates.
- **Case Splitting:** Usually the mentioned steps are sufficient. For a few harder instructions or if the low-equivalence relation requires refinement, we apply case splits, following the branching structure closely.
- **Evaluation:** After the case splitting, a number of more aggressive simplifications, evaluations, and automatic proof tactics are used to unfold remaining constants and to reason about words, bit operations, unusual forms of record accesses, and other corner cases.

4.3 Relation Refinement

Throughout the analysis, refinement of the low-equivalence relation is required whenever noninterference does not hold for the instruction currently considered. Counterexamples to noninterference enable the identification of new components to be downgraded to low. When managed carefully, failed backward proofs of noninterference allow to extract such counterexamples. However, backward proofs are not complete. Unsatisfiable subgoals might be introduced despite the goal being verifiable. For accuracy, we thus verify the necessity of downgrading a component C before the actual refinement of the relation. To that end, it is sufficient to identify two witness states that fulfil the preconditions $\mathcal{P}$, agree on all components except C, and lead to a violation of noninterference in respect to the analysed instruction Φ and the current (yet to be refined) relation $\mathcal{R}$. We refer to the existence of such witnesses as $\overline{\mathcal{N}}$:

$$\begin{aligned} &\overline{\mathcal{N}}(\mathcal{R}, \mathcal{P}, \Phi, C) := \exists s, x_1, x_2, v_1, v_2, t_1, t_2 : \\ &\;((v_1, t_1) = \Phi(s[C := x_1])) \wedge ((v_2, t_2) = \Phi(s[C := x_2])) \\ &\;\wedge \mathcal{P}(s[C := x_1]) \wedge \mathcal{P}(s[C := x_2]) \wedge (\neg\mathcal{R}(t_1, t_2) \vee (v_1 \neq v_2)) \end{aligned}$$

If such witnesses exist, any sound relation $\mathcal{R}'$ refining $\mathcal{R}$ will have to contain some restriction on C. With the chosen granularity, that translates to $\forall s_1, s_2 : \mathcal{R}'(s_1, s_2) \Rightarrow (\mathcal{R}(s_1, s_2) \wedge s_1.C = s_2.C)$. We proceed with the weakest such relation, i.e., $\mathcal{R}'(s_1, s_2) := (\mathcal{R}(s_1, s_2) \wedge s_1.C = s_2.C)$. As discussed in Section 3.1, it can be undecidable whether the current relation needs refinement. However, for the models that we analyzed, our framework was always able to verify the existence of suitable witnesses. The identification and verification of new low components consists of three steps:

1. **Identification of a new low component.** We transform subgoal G on top of the goal stack into a subgoal `false` with premises extended by $\neg G$. In this updated list of premises for the pre-states s_1 and s_2, we identify a premise on s_1 which would solve the transformed subgoal by contradiction when assumed for s_2 as well. Intuitively, we suspect that noninterference is prevented by the disagreement on components in the identified premise. We arbitrarily pick one such component as candidate for downgrading.
2. **Existential verification of the scenario.** To ensure that the extended premises alone are not already in contradiction, we prove the existence of a scenario in which all of them hold. We furthermore introduce the additional premise that the two pre-states disagree on the chosen candidate, but agree on all other components. An instantiation satisfying this existential statement is a promising suspect for the set of witnesses for $\overline{\mathcal{N}}$. The existential proof in HOL4 refines existentially quantified variables with patterns, e.g., symbolic states for state variables, bit vectors for words, and mappings with abstract updates for function variables (allowing to reduce $\exists f : P(f(n))$ to $\exists x : P(x)$). If possible, existential goals are split. Further simplifications include HOL4 tactics particular to existential reasoning, the application of type-specific existential inequality theorems, and simplifications on word and bit operations. If after those steps and automatic reasoning existential subgoals remain, the tool attempts to finish the proof with different combinations of standard values for the remaining existentially quantified variables.
3. **Witness verification.** We use the anonymous witnesses of the existential statement in the previous step as witnesses for $\overline{\mathcal{N}}$. After initialisation, the core parts of the proof strategy from the failed noninterference proof are repeated until the violation of noninterference has been demonstrated.

In order to keep the analysis focused, it is important to handle case splits before entering the refinement stage. At the same time, persistent case splits can be expensive on a non-provable goal. Therefore, we implemented a depth first proof tactical, which introduces hardly any performance overhead on successful proofs, but fails early in cases where the proof strategy does not succeed. Furthermore, whenever case splits become necessary in the proof attempt, the framework strives to diverge early, prioritizing case splits on state components.

5 Evaluation

We applied our framework to analyse information flows on ARMv7-A and MIPS-III (64-bit RS4000). For ARM, we focus on user mode execution without security or virtualization extension. Since unprivileged ARM code is able to switch between several instructions sets (ARM, Thumb, Thumb2, ThumbEE), the information flow analysis has to be performed for all of them. For MIPS, we consider all three privilege modes (user, kernel, and supervisor). The single-core model does not include floating point operations or memory management instructions.

ISA	mode	initial relation	final relation
ARMv7-A	user mode	program counter	user registers; control register CPSR (all flags); floating point registers of FP.REG and FP.FSPCR; TEEHBR register (coprocessor 14); Encoding ghost component; system control register SCTLR (coprocessor 15, flags: EE, TE, V, A, U, DZ)
MIPS-III	user *or* kernel *or* supervisor mode	program counter; BranchTo; BranchDelay; CP0.Count; exception marker; CP0.Status.KSU; CP0.Status.EXL; CP0.Status.ERL	all modelled system components
MIPS-III	*restricted* user mode		general purpose register set; LLbit; lo; hi; CP0.Config.BE; CP0.Status.RE; CP0.Status.BEV; exceptionSignalled

Table 1. Identified flows (model components might deviate from physical systems)

Table 1 shows the initial and accurate final low-equivalence relations for the two ISAs with different configurations. All relations refine the memory relation. The *final relation* column only lists components not already restricted by the corresponding initial relations. For simplicity, the initial relation for MIPS restricts three components accessed on the highest level of NEXT. The corresponding table cell also lists components already restricted by the preconditions. Initially unaware of the privilege management in MIPS, we were surprised that our tool first yielded the same results for all MIPS processor modes and that even user processes can read the entire state of system coprocessor CP0, which is responsible for privileged operations such as the management of interrupts, exceptions, or contexts. To restrict user privileges, the CU0 status flag must be cleared (see last line of the table). While ARMv7-processes in user mode can not read from banked registers of privileged modes, they can infer the state of various control registers. Alignment control register flags (CP15.SCTLR.A/U in ARMv7) are a good example for implicit flows in CPUs. Depending on their values, an unaligned address will either be accessed as is, forcibly aligned, or cause an alignment fault. Table 2 shows the time that rewriting, instruction proofs (including relation refinement), and the composing proof for NEXT took on a single Xeon® X3470 core. The first benchmark for MIPS refers to unrestricted user mode (with similar times as for kernel and supervisor mode), the second one to restricted user mode. Even though we borrowed a few data type theorems and some basic machinery from the step library, we did not use instruction specific theorems for the MIPS verification. Both ISAs have around 130 modelled instructions, but with 9238 lines

ISA	rewrite	instr.	NEXT	total
ARMv7	29,829	46,146	2,171	78,146 (21 h, 42 min)
MIPS (1)	537	1,790	1,594	3,921 (1 h, 5 min)
MIPS (2)	537	1,216	562	2,315 (38 min)

Table 2. Proof performance (in seconds)

step	min	median	mean	max
rewrite	1	25	167	2,384
instr. (success)	1	15	96	3,605
instr. (fail)	3	26	72	1,544
refinement	7	50	89	1,326

Table 3. Performance ARMv7 proof

of L3 compared to 2080 lines [7], the specifications of the ARMv7 instructions are both larger and more complex. Consequently, we observed a remarkable difference in performance. However, as Table 3 shows, minimum, median, and mean processing times (given in seconds) for the ARM instructions are actually moderate throughout all steps (rewriting, successful and failed noninterference proofs, and relation refinement). Merely a few complex outliers are responsible for the high verification time of the ARM ISA. While we believe that optimizations and parallelization could significantly improve performance, those outliers still demonstrate the limits of analyzing entire instructions as a whole. Combining our approach with compositional solutions such as [13] could overcome this remaining challenge. We leave this for future work.

6 Related Work

While most work on processor verification focuses on functional correctness [4,5,21] and ignores information flow, we survey hardware noninterference, both for special separation hardware and for general purpose hardware.

Noninterference Verification for Separation Hardware Wilding et al. [24] verify noninterference for the partitioning system of the AAMP7G microprocessor. The processor can be seen as a separation kernel in hardware, but lacks for example user-visible registers. Security is first shown for an abstract model, which is later refined to a more concrete model of the system, comprising about 3000 lines of ACL2. The proof appears to be performed semi-automatically.

SAFE is a computer system with hardware operating on tagged data [2]. Noninterference is first proven for a more abstract machine model and then transferred to the concrete machine by refinement. The proof in Coq does not seem to involve much automation.

Sinha et al. [20] verify confidentiality of x86 programs that use Intel's Software Guard Extensions (SGX) in order to execute critical code inside an SGX enclave, a hardware-isolated execution environment. They formalize the extended ISA axiomatically and model execution as interleaving between enclave and environment actions. A type system then checks that the enclave does not contain insecure code that leaks sensitive data to non-enclave memory. At the same time, accompanying theorems guarantee some protection from the environment, in particular that an adversary can not influence the enclave by any instruction other than a write to input memory. However, [20] assumes that SGX management data structures are not shared and that there are no register contents that survive an enclave exit and are readable by the environment. Once L3/HOL4 models of x86 with SGX are available, our machinery would allow to validate those assumptions in an automated manner, even for a realistic x86 ISA model. Such a verification would demonstrate that instructions executed by the environment do not leak enclave data from shared resources (like non-mediated registers) to components observable by the adversary.

Noninterference Verification for General Purpose Hardware Information flow analysis below ISA level is discussed in [18] and [15]. Procter et al. [18] present a functional hardware description language suitable for formal verification, while the language in [15] can be typed with information flow labels to allow for static verification of noninterference. Described hardware can be compiled into VHDL and Verilog, respectively. Both papers demonstrate how their approaches can be used to verify information flow properties of hardware executing both trusted and untrusted code. We are not aware of the application of either approach to information flow analysis of complex commodity processors such as ARM.

Tiwari et al. [23] augment gate level designs with information flow labels, allowing simulators to statically verify information flow policies. Signals from outside the TCB are modelled as *unknown*. Logical gates are automatically replaced with label propagating gates that operate on both known and unknown values. The authors employ the machinery to verify the security of a combination of a processor, I/O, and a microkernel with a small TCB. It is unclear to us how the approach would scale to commodity processors with a more complex TCB. From our own experience on ISA-level, the bottleneck is mainly constituted by the preprocessing to obtain the model's evaluated form and by the identification of a suitable labelling. The actual verification is comparatively fast.

In earlier work [13] we described a HOL4 proof for the noninterference (and other isolation properties) of a monadic ARMv7-model. A compositional approach based on proof rules was used to support a semi-automatic analysis. However, due to insufficient context, a number of transitions had to be verified manually or with the support of context-enhancing proof rules. In the present work, we overcome this issue by analysing entire instructions. Furthermore, our new analysis exhibits the low-equivalence relation automatically, while [13] provides it as fixed input. Finally, the framework described in the present paper is less dependent of the analysed architecture.

Verification of Binaries Fox's ARM model is also used to automatically verify security properties of binary code. Balliu et al. [3] does this for noninterference, Tan et al. [22] for safety-properties. Despite the seeming similarities, ISA analysis and binary code analysis differ in many respects. While binary verification considers concrete assembly instructions for (partly) known parameters, ISA analysis has to consider all possible assembly instructions for all possible parameters. On the other hand, it is sufficient for an ISA analysis to do this for each instruction in isolation, while binary verification usually reasons on a sequence (or a tree of) instructions. In effect, that makes the verification of a binary program an analysis on imperative code. In contrast, ISA analysis (in our setting) is really concerned with functional code, namely the operational semantics that describe the different steps of single instructions. In either case, to enable full automation, both analyses have to include a broader context when the local context is not sufficient to verify the desired property for a single step in isolation. As discussed above, we choose an instruction-wide context from the beginning. Both [3] and [22] employ a more local reasoning. In [22] a Hoare-style logic is used and context is provided by selective synchronisation of pre- and postconditions between neighbouring code blocks. In [3] a forward symbolic analysis carries the context in a path condition when advancing from instruction to instruction. SMT solvers then allow to discard symbolic states with non-satisfiable paths.

7 Discussion on Unpredictable Behaviour

ISA specifications usually target actors responsible for code production, like programmers or compiler developers. Consequently, they are often based on the assumption that executed code will be composed from a set of well-defined instructions and sound conditions, so that no one relies on combinations of instructions, parameters and configurations not fully covered by the specification. This allows to keep instructions partly underspecified and leave room for optimizations on the manufacturer's side. However, this practice comes at the cost of actors who have to trust the execution of unknown and potentially malicious third-party code. For example, an OS has an interest in maintaining confidentiality between processes. To that end, it has different means such as clearing visible registers on context switches. But if the specification is incomplete on which registers actually are visible to an instruction with uncommon parameters, then there is no guarantee that malicious code can not use underspecified instructions (i.e., instructions resulting in unpredictable states) to learn about otherwise secret components. ARM attempts to address this by specifying that "*unpredictable* behaviour must not perform any function that cannot be performed at the current or lower level of privilege using instructions that are not *unpredictable*". [4] While this might indeed remedy integrity concerns, it is still problematic for noninterference. An underspecified instruction can be implemented by two different "safe" behaviours, with the choice of the behaviour depending on an otherwise secret component. The models by Fox et al. mark the post-states of underspecified operations as unpredictable by assigning an exception marker to those states. In addition, newer versions still model a reasonable behaviour for such cases, but there is no guarantee that the manufacturer chooses the same behaviour. A physical implementation might include flows from more components than the model does, or vice versa. A more conservative analysis like ours takes state changes after model exceptions into account, but can still miss flows simply not specified. To the rescue might come statements from processor designers like ARM that "*unpredictable* behaviour must not represent security holes". [5] In one

[4] ARMv7-A architecture reference manual, issue C: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c`

[5] ARMv7-A architecture reference manual, issue B

interpretation, flows not occurring elsewhere can be excluded in underspecified instructions. The need to rely on this interpretation can be reduced (but not entirely removed) when the exception marker itself is considered low in the initial labelling. As an example, consider an instruction that is well-defined when system component C_1 is 0, but underspecified when it is 1. The manufacturer might choose different behaviours for both cases, thus possibly introducing a flow from C_1 to low components. At the same time, the creator of the formal model might implement both cases in the same way, so that the analysis could miss the flow. But with a low exception marker, C_1 would also be labelled low, since it influences the marker. However, an additional undocumented dependency on another component C_2 that only exists when C_1 is 1 can still be missed.

8 Conclusions and Future Work

We presented a sound and accurate approach to automatically and statically verify noninterference on instruction set architectures, including the automatic identification of a least restrictive low-equivalence relation. Besides applying our framework to more models such as the one of ARMv8, we intend to improve robustness and performance, and to cover integrity properties as well.

Integrity Properties We plan to enhance the framework by safety-properties such as nonexfiltration [10,13] and mode switch properties [13]. Nonexfiltration asserts that certain components do not change throughout (unprivileged) execution. Mode switch properties make guarantees on how components change when transiting to higher privilege levels, for example that the program counter will point to a well-defined entry point of the kernel code. We believe that both properties can be derived relatively easily from the normalized forms of the instructions.

Performance Optimization While our benchmarks have demonstrated that ISA information flow analysis on an instruction by instruction basis allows for a large degree of automation, they also have shown that this approach introduces severe performance penalties for more complex instructions. To increase scalability and at the same time maintain automation, we plan to investigate how to combine the compositional approach of [13] with the more global reasoning demonstrated here. Furthermore, there is potential for improvements in the performance of individual steps. E.g., our normalization could be combined with the one of [7].

Acknowledgments. Work supported by the Swedish Foundation for Strategic Research, by VINNOVA's HASPOC-project, and by the Swedish Civil Contingencies Agency project CERCES. Thanks to Anthony C. J. Fox, Roberto Guanciale, Nicolae Paladi, and the anonymous reviewers for their helpful comments.

References

1. E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. Automated verification of a small hypervisor. In *VSTTE*, pages 40–54, 2010.
2. A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *Principles of Programming Languages*, POPL, pages 165–178, 2014.
3. M. Balliu, M. Dam, and R. Guanciale. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 1080–1091, 2014.
4. S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. Putting it all together – formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4):411–430, 2006.
5. D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design*, pages 203–222, 1994.
6. M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Computer and Communications Security*, CCS, pages 223–234, 2013.
7. A. C. J. Fox. Improved tool support for machine-code decompilation in HOL4. In *Interactive Theorem Proving (ITP)*, pages 187–202, 2015.
8. A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving (ITP)*, pages 243–258, 2010.
9. N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Principles of Programming Languages*, POPL, pages 365–377, 1998.
10. C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Applying formal methods to a certifiably secure software system. *IEEE Trans. Softw. Eng.*, 34(1):82–98, 2008.
11. HOL4 project. `http://hol.sourceforge.net/`.
12. S. Hunt and D. Sands. On flow-sensitive security types. In *Principles of Programming Languages*, POPL, pages 79–90, 2006.
13. N. Khakpour, O. Schwarz, and M. Dam. Machine assisted proof of ARMv7 instruction level isolation properties. In *Certified Programs and Proofs (CPP)*, pages 276–291, 2013.
14. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *SOSP*, pages 207–220, 2009.
15. 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 *Programming Language Design and Implementation*, PLDI, pages 109–120, 2011.
16. T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In *Security and Privacy*, pages 415–429, 2013.
17. F. Pottier and V. Simonet. Information flow inference for ML. In *Principles of Programming Languages*, POPL, pages 319–330, 2002.
18. A. Procter, W. L. Harrison, I. Graves, M. Becchi, and G. Allwein. Semantics driven hardware design, implementation, and verification with ReWire. In *Languages, Compilers and Tools for Embedded Systems*, LCTES, pages 13:1–13:10, 2015.
19. O. Sibert, P. A. Porras, and R. Lindell. The Intel 80x86 processor architecture: Pitfalls for secure systems. In *Security and Privacy*, SP, pages 211–222, 1995.

20. R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *Comp. and Comm. Security*, pages 1169–1184, 2015.
21. M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Softw.*, 7(5):52–64, 1990.
22. J. Tan, H. J. Tay, R. Gandhi, and P. Narasimhan. AUSPICE: Automatic safety property verification for unmodified executables. In *Working Conference on Verified Software: Tools, Theories and Experimems (VSTTE)*, pages 202–222, 2015.
23. 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 *International Symposium on Computer Architecture*, ISCA, pages 189–200, 2011.
24. M. M. Wilding, D. A. Greve, R. J. Richards, and D. S. Hardin. Formal verification of partition management for the AAMP7G microprocessor. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 175–191. Springer, 2010.