Automating Information Flow Analysis of Low Level Code

Musard Balliu, Mads Dam, Roberto Guanciale
KTH Royal Institute of Technology
SE-100 44, Stockholm, Sweden
{musard,mfd,robertog}@kth.se

ABSTRACT

Low level code is challenging: It lacks structure, it uses jumps and symbolic addresses, the control flow is often highly optimized, and registers and memory locations may be reused in ways that make typing extremely challenging. Information flow properties create additional complications: They are hyperproperties relating multiple executions, and the possibility of interrupts and concurrency, and use of devices and features like memory-mapped I/O requires a departure from the usual initial-state final-state account of noninterference. In this work we propose a novel approach to relational verification for machine code. Verification goals are expressed as equivalence of traces decorated with observation points. Relational verification conditions are propagated between observation points using symbolic execution, and discharged using first-order reasoning. We have implemented an automated tool that integrates with SMT solvers to automate the verification task. The tool transforms ARMv7 binaries into an intermediate, architecture-independent format using the BAP toolset by means of a verified translator. We demonstrate the capabilities of the tool on a separation kernel system call handler, which mixes hand-written assembly with gcc-optimized output, a UART device driver and a crypto service modular exponentiation routine.

Categories and Subject Descriptors

D4.6 [**Operating Systems**]: Security and Protection—*Information flow controls, Security kernels, Verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*

Keywords

Information Flow Security; Formal Verification; Symbolic Execution; Machine Code

1. INTRODUCTION

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660322.

The ultimate goal of information flow analysis is to establish confidentiality and integrity properties of real code executing on commodity CPUs. In the literature, normally this problem is addressed at the source code level. There it may be more forgiving to ignore messy low level problems, e.g. regarding timing, complex control flow, or hardware specifics. Also, one may appeal to special compilers that avoid difficult optimizations, or work around machine features such as caching, instruction reordering, concurrency, I/O, interrupts, bus contention and so on, that are difficult to handle in a precise manner.

Sometimes, however, source level analysis is less suitable. This is certainly the case when dealing with third-party code, but it applies in other cases too, for instance, for heavily optimized or obfuscated code, and for kernel handler routines that manipulate security sensitive peripherals such as privileged processor registers, MMUs, and bus and interrupt controllers.

The literature has two "standard" approaches to information flow control (IFC) for low level languages: (a) For static verification, most authors, cf. [33, 12], have attempted to reimpose typing and high level structure at the assembly or byte code level, in order to reuse standard type-based techniques for high level languages. For instance, [33], uses this approach for typed assembly language, and [12] takes a related approach to Java bytecode. (b) Most work, however, has focused on dynamic techniques, often using some combination with static analysis to generate labels, or tags, to help minimize the dynamic overhead, cf. [13, 29, 46, 31, 21]. For instance, [21] proposes a machine architecture with hardware-supported tag propagation to support dynamic information flow tracking.

Neither of these schools are very helpful, though, when it comes to the problem we have set out to study: Information flow analysis for low level code on commodity processors. In this domain, existing static approaches are too imprecise due to lightweight (data/flow/path/timing-insensitive) analysis, while dynamic approaches suffer from the well known problem of label creep and introduce undesired runtime overhead [42]. Security testing-like techniques [37, 5], which we discuss later, provide impressive results in terms of scalability, however, they are in general unsound and can not directly be used for full verification.

Instead we propose to directly verify relational (i.e. information flow) properties at machine code level, leveraging as much as possible recent progress on low level code analysis tools such as BAP [14], McVeto [48], Vine/BitBlaze [45]. Code for our target machine, ARMv7, is first lifted to

a machine-independent intermediary form, BIL, using the BAP tool [14]. This process uses a lifter that is produced from the Cambridge HOL4 model of ARMv7 [28]. This allows the reuse and extension of BAPs program verification back end to symbolically execute the resulting BIL code. We use this to first perform *unary* analysis and then verify relational properties by propagating relational preconditions through each of a pair of related programs until a pair of observation points are reached, that need to be matched, in order for the relational property to hold. These observation points are memory write events, to locations that are statically determined to be observable by some external agent, because of multithreading, or memory-mapped I/O, or for some other reason. Matching is done by SMT solving using STP [24], on formulas that tend to grow huge, but generally rely only on linear arithmetic, uninterpreted functions, and arrays, and so are not too costly to check. Special care is needed for memory accesses which introduce quantifier alternation, hence we propose an instantiation technique which ensures the resulting formulas are quantifier free.

Three distinguishing features make our information flow analysis both useful and challenging: *loop invariants*, *timing* and *traces*. Loops are handled using (relational) invariants/widening. We point out that relational invariants can be significantly simpler than state invariants as they may not require proving functional correctness of the loop. Our case studies show that the invariants we provide are conjunctions of linear equalities, which, as shown in recent work [44], can be generated automatically. Timing is particularly critical. The timing information is included in the symbolic state and propagated with the other constraints. The model used here scales to functional cost models, i.e. models where the timing cost can be calculated as function of the input instruction, independent of the history. This is evidently realistic only for simple processor architectures such as ARM Cortex-M (but we note that a vast number of such processors are in use today in critical control applications). Richer and tractable timing models that can take into account also features like caches and instruction pipelines are, however, currently not available at ISA level, and we leave this for future work. Finally, the trace-based analysis broadens the number of target applications handled by our technique, including preemptive environments and scheduling.

We are the first to admit that the approach will suffer from scalability problems, for instance due to path explosion, and due to the generally complex and detailed machine state. However, our primary application is separation kernel handler verification, and this domain is generally characterized by critical machine code fragments that are rather small (generally under 1K instructions per handler), but also tricky. The case studies reported in this paper are based on syscall handlers and device drivers of slightly more than 250 lines of ARMv7, produced by a mix of hand-crafted assembly and GCC-optimized C.

Overall, this paper makes both theoretical and practical contributions. On the theoretical side, we present a novel approach for formal relational machine code verification, with focus on information flow security properties. The combination of unary and relational analysis makes our approach appealing for precise security analysis of machine code. We provide a new angle with the inclusion of timing information into the state and with the invariant handling which ensures a nice compositional property over traces. On the practical side, we present the first automated toolset for information flow analysis of ARMv7 binaries. We exercise the tool on non-trivial case studies including separation kernel syscalls, device drivers and crypto routines. For a more thorough discussion of related work, please refer to Sect. 8.

2. THREAT MODEL AND SECURITY

In our target applications, trusted and untrusted agents share and control parts of the system memory. Our goal is to ensure that the only information channels connecting agents to each other are the intended ones. These intended channels can be shared buffers, network connections, or specific communication devices such as the message sending syscall handler considered later in the case study. They can also be memory-mapped devices connecting agents to the external world such as a UART device. In the special case where channels form the usual security lattice, the goal reduces to classical Goguen-Meseguer information flow [25], which requires the state of the untrusted program be unaffected by the state of the trusted one.

We use the ARMv7 program in Fig. 1 to elucidate our threat model. The program loads a memory pointer from the address 2048 into the register `R3`. Subsequently, the memory referenced by the pointer is updated three times. The program always terminates with the following effect on the system state: (i) the memory pointer is loaded into the register `R3`, (ii) the registers `R1` and `R2` are updated to zero (lines 0x110 and 0x118), (iii) the "zero flag" `Z` is enabled (the instruction at line 0x110 contains the `S` suffix, thus overriding `Z` according to the result of the executed arithmetic operation [4]) and (iv) zero is written (line 0x114) into the memory referenced by the pointer.

```
0x0f4 MOV R2, #0
0x0f8 LDR R3, [PC+#0x700] //2048
0x0fc STR R2, [R3]
0x100 LDR R1, [PC+#0x2f8] //1024
0x104 ADDS R1, R1, R2
0x108 MOVEQ R2, #1
0x10c STR R2, [R3]
0x110 MOVS R2, #0
0x114 STR R2, [R3]
0x118 MOV R1, #0
```

Figure 1: ARMv7 Program

Assume that the system memory from address 0 to 2047 contains the state of a trusted agent and that the remaining part of the memory contains the state of an untrusted agent. If an observing agent is not able to access its own memory while the program is executed, then the above program can be considered secure, since after termination the state of the untrusted agent is unaffected by the state of the trusted one. However, in several scenarios this requirement is not satisfied: (i) the observer controls a device that is mapped to a memory area that belongs to the untrusted agent, (ii) the code is interrupted and scheduled in a preemptive environment, (iii) memory stores have side effects, or (iv) the code is executed in a multi-core setting.

In all these cases, a departure from the usual initial-state final-state account of noninterference is required. In particular, all updates to the untrusted memory (e.g. lines 0x0fc, 0x10c and 0x114) can be monitored by the attacker and re-

veal secret information. Hence our example program can not be considered secure. In fact, depending on the content of the memory of the trusted agent, the assembly fragment can have the following executions:

1) If the memory at address 1024 is zero (line 0x100) then (i) the instruction 0x104 enables the flag `Z`, (ii) the instruction 0x108 updates the register `R2` to 1, (iii) thus the address referenced by the pointer is updated three times, with the values 0, 1 and 0, respectively.

2) Otherwise (i) the instruction 0x104 disables the flag `Z`, (ii) the instruction 0x108 has no effect, (iii) thus the memory referenced by the pointer is updated three times, always with the value zero.

Consequently, an attacker capable of observing the relevant memory state will in one case see the referenced memory location flicker, and in another case not. The security condition must prevent this phenomenon.

In this paper we work with *observational determinism* [32], defined as follows. Assume a set of configurations C and a transition relation $\rightarrow \subseteq C \times C$. An *execution* is a maximal finite or infinite sequence

$$\pi = C_0 \rightarrow \cdots \rightarrow C_n \rightarrow \cdots \tag{1}$$

of configurations related by the transition relation. The initial configuration of π in (1) is C_0. A transition may give rise to an observation *obs*. In our case, observations are timed writes to observer readable memory. An *observation trace*, or just *trace*, $trc(\pi)$, of π extracts from π the sequence of observations produced by π. The traces π_1 and π_2 are then *trace equivalent*, if $trc(\pi_1) = trc(\pi_2)$.

Since we assume a concept of observer readable memory, it makes sense to define the relation $C \equiv C'$ by requiring that the observer readable memory of C and C' are the same. This is the familiar notion of observational, or low configuration (state) equivalence. We can then proceed to define observational determinism.

Definition 1 (Observational Determinism). *A set P of executions π is observational deterministic, if for any pair of executions $\pi_1, \pi_2 \in P$ with initial configurations C_1 and C_2, respectively, if $C_1 \equiv C_2$ then π_1 and π_2 are trace equivalent.*

Observational determinism works well for a class of nondeterministic programs, and it is preserved under refinement [32]. In particular it avoids the quantifier alternation in bisimulation-oriented unwinding conditions. To be accurate, observational determinism presupposes that all nondeterminism can be relegated to the initial state. This is true in our case. However, observational determinism is less suitable when alternation is essential, for instance in the case of strategic reasoning.

3. MACHINE MODEL

For the theory development we consider deterministic programs written in a simple machine language (SiML) with the following instruction syntax

$$\begin{array}{lcl} \iota & ::= & \mathit{reg} := \mathit{exp} \mid \mathbf{cjmp}(\mathit{exp}, \mathit{exp}, \mathit{exp}) \mid \mathbf{assert}(\mathit{exp}) \\ & & \mid \mathbf{assume}(\mathit{exp}) \mid \mathbf{store}(\mathit{exp}, \mathit{exp}) \mid \mathbf{halt} \\ \mathit{exp} & ::= & \mathbf{load}(\mathit{exp}) \mid \mathbf{bop}(\mathit{exp}, \mathit{exp}) \mid \mathit{reg} \mid \mathit{PC} \\ & & \mid \mathbf{uop}(\mathit{exp}) \mid v \end{array}$$

There are registers $reg \in Reg$, the program counter PC and values $v \in Val$ that are, for simplicity, taken as primitive. Instructions include register assignments, memory stores, conditional jumps, assertions, assumptions, and halt. Expressions include unary and binary operations on constants, register lookups, and memory loads.

A SiML program is evaluated in the context of a register state $\Delta : Reg \mapsto Val$, a program counter $pc \in Val$ and a memory state $\mu : Val \mapsto Val$. We assume a SiML programs be non-self modifying, thus instructions are stored in a separate instruction memory $\Pi : Val \mapsto \iota$ which is usually addressed via the program counter. The instruction memory is a total function and we assume that for each possible address v outside the executable part of the memory $\Pi(v) = \mathbf{assert}(0)$. A *configuration* C is either a tuple $(\Pi, \Delta, \mu, pc, t)$ of instruction memory, register state, data memory, program counter and current execution time $t \in \mathbb{N}$, or an error configuration $\perp$, used to handle failing asserts. The transition relation has the shape $C \rightarrow C'$ where $C \neq \perp$. SiML is a subset of the BAP Intermediate Language (BIL), which is used for lifting the ARMv7 binaries. We refer to [15] for a complete definition of the operational semantics of BIL. Instructions and expressions are evaluated in the context of a configuration. For instance, the assignment $reg := exp$ assigns to register reg the value of exp, the conditional jump $\mathbf{cjmp}(e_1, e_2, e_3)$ transfers control to e_2 if e_1 is true (non-zero), otherwise to e_3. The $\mathbf{assert}(b)$ statement terminates the program abnormally if b is false, otherwise it has no effect on the state, the $\mathbf{store}(e_1, e_2)$ stores the value of e_2 at memory location e_1 and the expression $\mathbf{load}(e_1)$ loads the value at location e_1. We distinguish between normal and abnormal executions. A *normal* execution, if it ever terminates, executes **halt** as the last instruction. Otherwise the execution is *abnormal* and terminates in the error configuration $\perp$. The length of π is $len(\pi)$, the i-th configuration of π is $\pi(i)$, $\iota(\pi, i) = \Pi(pc_i)$, $pc(\pi, i) = pc_i$, and $t(\pi, i) = t_i$. A *model* $\mathcal{M}$ consists of the set of executions π induced by some set of initial configurations C_0.

Our target applications require reasoning about the execution time of the program. The timing behavior is highly architecture-dependent and is in general very difficult to capture accurately. In this paper we work with a functional time model $\tau : \iota \rightarrow \mathbb{N}$ which assigns a fixed parameter-dependent cost to each instruction. That is if we have $(\Pi, \Delta, \mu, pc, t) \rightarrow (\Pi, \Delta', \mu', pc', t')$ then $t' = t + \tau(\Pi(pc))$. For instance, the execution time of a load instruction will depend on the number of bytes to load from the memory. Under these assumptions, the execution time is history-independent and the timing model is deterministic.

4. UNARY SYMBOLIC ANALYSIS

We reason about the behavior of a program by means of forward symbolic analysis. The analysis allows us to build a logical formula, which corresponds to multiple program executions, and leverage first-order reasoning to statically prove program properties. The program is executed on symbolic inputs and, consequently, the state is also symbolic. Initially, registers are mapped to fresh variables, the memory is a variable representing an uninterpreted function and the program counter is a constant. We use exp^s and e^s to range over symbolic expressions, which are built over these initial variables and constants using the standard machinery and have either type memory or type value. In particular, if exp^s is a memory expression and exp^s_1 and exp^s_2 are

expressions of type value, then $exp^s(exp_1^s)$ is an expression of type value representing the lookup of exp_1^s in exp^s, and $exp^s[exp_1^s \mapsto exp_2^s]$ is a memory expression representing the corresponding update.

A symbolic state is a tuple $\Sigma^s = (\Delta^s, \mu^s, pc)$, where Δ^s maps registers to symbolic expressions, μ^s is a symbolic memory expression and pc is the concrete value representing the program counter. A symbolic configuration C^s is a tuple $(\Pi, \Delta^s, \mu^s, \phi, pc, t)$, which extends a symbolic state with a path predicate ϕ, the instruction memory Π and the execution time t. The path predicate ϕ, also path condition, is a symbolic boolean expression built over the initial variables and constrains the set of concrete initial states that execute the path. Usually, the path condition of the initial configuration entails the program preconditions.

Forward symbolic semantics is given by the transition rules on symbolic configurations depicted in Fig. 2. Here, we use $\Delta^s, \mu^s \vdash exp \Downarrow exp^s$ to represent the symbolic evaluation of an expression exp in the context (Δ^s, μ^s). For instance, if $\Delta^s, \mu^s \vdash exp \Downarrow exp^s$ then $\Delta^s, \mu^s \vdash \mathbf{load}(exp) \Downarrow \mu^s(exp^s)$. Notice that, since we assume non-self modifying code, we omit the constant instruction memory Π from the rules and the time is increased independently of the processor state. The **cjmp** rules evaluate the jump target in the current symbolic state and then update the program counter and the path condition depending on whether the jump condition is satisfiable. The jump target can be a symbolic expression which requires to resolve all possible targets in the current context. This can be addressed by enumerating all concrete jump targets that are consistent with the path condition, for example using a decision procedure that returns all satisfying assignments of the formula ϕ' in state Σ^s. Another complication arises when considering memory load and store operations. Memory addresses can be symbolic as reported in the rules for **load** and **store** instructions. This would require to evaluate the symbolic expression in the context of a symbolic state and a path predicate ϕ, and then compute all concrete addresses as for the **cjmp** rules. This process can in general be infeasible due to the huge amount of possible concrete addresses at a given point, most of which will be irrelevant to the final analysis. The solution we adopt in Fig. 2 is to propagate the symbolic expression and postpone the address resolution when needed. The **assert** rules use a first order oracle to decide the validity of the asserted expression, while the **assume** rule propagates the constraint as expected. Another problem that arises with the proof system is the possible nontermination due to unbounded loops, which we tackle by providing invariants, as discussed later.

The correctness of forward symbolic execution can be justified in terms of the strongest postcondition transformer [22]. We start with a SiML program Π and a property vector F, both having the same length. The property vector assigns to each program location l a formula F_l, which represents a property of executions reaching l. The strongest postcondition vector $sp(\Pi, F)$ consists of entries $sp(\Pi, F)_l$ representing the pointwise strongest, i.e. smallest, condition which guarantees that, when property F_j holds in the prestate and control passes from instruction j to l, then $sp(\Pi, F)_l$ holds in the poststate. The construction uses the iterator $spstep(\iota, \phi, j, l)$ which handles the case of control transfers from j, with ϕ holding at j, to l, with ι the instruction being executed. The sp function is pointwise monotone, and hence, using standard techniques, the largest cumulative fixed point F_{lim} satisfying $F_{lim} = sp(\Pi, F_{lim}) \sqsubseteq F_{init}$ can be obtained from an initial property vector F_{init}. The iterative computation is evidently not guaranteed to terminate, but, by choosing the F_{init} vector in an intelligent way and providing invariants, it is in fact possible to compute $sp(F_{init})$ in many concrete situations [20], even for programs with convoluted control flow.

5. RELATIONAL SYMBOLIC ANALYSIS

We now turn to the relational analysis for proving information flow properties defined in the previous sections. The main idea is to perform forward symbolic execution on a pair of programs and verify the information flow relation at each observation point.

5.1 Symbolic Observation Trees

The threat model assumes the attacker has access to part of the memory, can observe any store on his memory addresses and count the time elapsed up to the point where an observation occurs. The symbolic analysis accounts for the observation points, which are represented as symbolic constraints. We use a predicate P_O to define the range of the observable memory addresses. For instance, in Example 1, the untrusted agent has assigned memory addresses higher than 2K, i.e., $P_O(v) = v \geq 2048$, hence an explicit enumeration is quite expensive. Tracking dependencies on observable memory is tricky because the store instructions can be symbolic and thus potentially write to both observable and unobservable addresses. Therefore it is necessary to distinguish between observable stores, which affect the attackers state and unobservable stores, which do not affect the attackers state. We solve the issue by forking the symbolic execution engine each time we consider a store instruction. As reported in Fig. 3, we first evaluate address exp_1 and expression exp_2 in the symbolic context, and then distinguish between stores at observable addresses and stores at unobservable addresses. The predicate P_O partitions the symbolic execution into one branch where the store is *always* observable and one where the store is *always* unobservable. This process is important to guarantee the correctness of the entire approach.

The first rule captures the paths where the store instruction only affects observable addresses and thus is relevant for the subsequent security analysis. The second rule captures the paths where the store instruction affects the unobservable addresses, hence the analysis proceeds normally. The first rule is used to extract a *symbolic observation* tuple.

Definition 2 (Symbolic observation). *Consider a symbolic configuration* $C^s = (\Pi, \Delta^s, \mu^s, \phi, pc, t)$ *such that* $\Pi(pc) = \mathbf{store}(exp_1, exp_2)$. *Then a symbolic observation is the tuple* $obs = (\phi, exp_1^s, exp_2^s, t)$ *obtained after applying the first rule in Fig. 3.*

Intuitively, a symbolic observation captures how the concrete executions, starting from initial states that satisfy ϕ, affect the observable memory when they reach control point pc. This is done by recording the execution timestamp t and the possible values (exp_2^s) stored in each observable address (exp_1^s).

Alg. 1 tracks all symbolic observations occurring in the program by building a symbolic observation tree for a starting configuration C_0^s and a range predicate P_O. We use **fse** to represent all configurations produced in one step by the

$$\frac{\Pi(pc) = \textbf{cjmp}(e_1, e_2, e_3) \quad \Delta^s, \mu^s \vdash e_1 \Downarrow e_1^s \quad \phi' = (\phi \wedge e_1^s \neq 0) \quad \Delta^s, \mu^s \vdash e_2 \Downarrow pc^s \quad \phi' \wedge (pc^s = pc')\ \textit{consistent}}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta^s, \mu^s, \phi', pc', t')}$$

$$\frac{\Pi(pc) = (reg := e) \quad \Delta^s, \mu^s \vdash e \Downarrow e^s}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta^s[reg \mapsto e^s], \mu^s, pc + 1, t')} \qquad \frac{\Pi(pc) = \textbf{store}(e_1, e_2) \quad \Delta^s, \mu^s \vdash e_1 \Downarrow e_1^s \quad \Delta^s, \mu^s \vdash e_2 \Downarrow e_2^s}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta, \mu^s[e_1^s \mapsto e_2^s], \phi, pc + 1, t')}$$

$$\frac{\Pi(pc) = \textbf{assume}(e) \quad \Delta^s, \mu^s \vdash e \Downarrow e^s \quad \phi' = (\phi \wedge e^s \neq 0)}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta^s, \mu^s, \phi', pc + 1, t')} \qquad \frac{\Pi(pc) = \textbf{cjmp}(e_1, e_2, e_3) \quad \Delta^s, \mu^s \vdash e_1 \Downarrow e_1^s \quad \phi' = (\phi \wedge e_1^s = 0) \quad \Delta^s, \mu^s \vdash e_3 \Downarrow pc^s \quad \phi' \wedge (pc^s = pc')\ \textit{consistent}}{(\Delta^s, \mu^s, \phi, pc.t) \rightarrow (\Delta^s, \mu^s, \phi', pc', t')}$$

$$\frac{\Pi(pc) = \textbf{assert}(e) \quad \Delta^s, \mu^s \vdash e \Downarrow e^s \quad \models (\phi \Rightarrow e^s \neq 0)}{(\Delta^s, \mu^s, \phi, pc.t) \rightarrow (\Delta^s, \mu^s, \phi, pc + 1, t')} \qquad \frac{\Pi(pc) = \textbf{assert}(e) \quad \Delta^s, \mu^s \vdash e \Downarrow e^s \quad \not\models (\phi \Rightarrow e^s = 0)}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow \bot}$$

Figure 2: Symbolic Semantics of Instructions, where $t' = t + \tau(\Pi(pc))$

$$\frac{\Pi(pc) = \textbf{store}(exp_1, exp_2) \quad \Delta^s, \mu^s \vdash exp_1 \Downarrow exp_1^s \quad \Delta^s, \mu^s \vdash exp_2 \Downarrow exp_2^s}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta, \mu^s[exp_1^s \mapsto exp_2^s], (\phi \wedge P_O(exp_1^s)), pc + 1, t + \tau(\Pi(pc)))}$$

$$\frac{\Pi(pc) = \textbf{store}(exp_1, exp_2) \quad \Delta^s, \mu^s \vdash exp_1 \Downarrow exp_1^s \quad \Delta^s, \mu^s \vdash exp_2 \Downarrow exp_2^s}{(\Delta^s, \mu^s, \phi, pc, t) \rightarrow (\Delta, \mu^s[exp_1^s \mapsto exp_2^s], (\phi \wedge \neg P_O(exp_1^s)), pc + 1, t + \tau(\Pi(pc)))}$$

Figure 3: Observable and Unobservable Stores

Algorithm 1 Program to Symbolic Observation Tree

INPUT: Predicate P_O, C_0^s
OUTPUT: Tree T
1. $W := [(C_0^s, createNode(Start))]$
2. **While** $W \neq \emptyset$
3. pop (C^s, T) from W
4. $(\Pi, \Delta^s, \mu^s, \phi, pc, t) := C^s$
5. **If** ϕ is satisfiable
6. **If** $\Pi(pc) ==$ **store**(e_1, e_2)
7. $(C_o^s, C_u^s) :=$ **fse**(C^s, P_O)
8. $(\Pi, \Delta_o^s, \mu_o^s, \phi_o, pc_o, t_o) := C_o^s$
9. **If** ϕ_o is satisfiable
10. $TreeN :=$ createNode$(obs(\phi_o, (\Delta^s, \mu^s \vdash e_1), (\Delta^s, \mu^s \vdash e_2)), t_o))$
11. push $TreeN$ to $T.children$
12. push $(C_o^s, TreeN)$ to W
13. push (C_u^s, T) to W
14. **Else If** $\Pi(pc) ==$ **halt**
15. push $createNode(End)$ to $T.children$
16. **Else**
17. **for** $C_1^s \in$ **fse**(C^s, P_O)
18. push (C_1^s, T) to W

unary analysis described in Sect. 4. In particular, we assume that for store instructions, the first element of the output of **fse** corresponds to the first rule in Fig. 3 (which uses P_O). The procedure evaluates program instructions one by one and creates a tree node each time it reaches an observable store (line 10). The algorithm starts with the tree root *T.Start*, the observation range predicate P_O and the worklist containing the initial configuration C_0^s. For each statement, the algorithm updates the symbolic states by calling **fse** (line 7 and 17). We discard all symbolic states that have a non feasible path (line 5). When considering a store (line 6-13), the symbolic execution applies the rules in Fig. 3 and potentially produces an observable state C_o^s and an unobservable state C_u^s. If C_o^s is feasible, (line 9), the algorithm creates a new node containing the symbolic observation and attaches it to the current tree node. Subsequent observations will be attached to the freshly created node. If a **halt** statement is reached (line 14-15), the current branch is terminated, else the worklist is updated with all symbolic states obtained by executing the non-store instruction (line 16-19).

We prove correctness of Alg. 1 only for bounded programs and discuss generalizations in Sect. 5.4. In a nutshell, we show that the tree produced by the algorithm contains all the information needed to verify the security of the original program Π. The symbolic observation tree T contains the observation traces induced by the model of Π. Given a concrete initial configuration C_0, the trace is extracted by considering a path in T (the node sequence from *Start* to *End*) such that C_0 satisfies all path predicates and the observation trace is obtained by the evaluation of symbolic observations in C_0, for each node along the path.

THEOREM 1. *Consider a SiML program Π and the corresponding symbolic observation tree T. Then, the set of observation traces of the model $\mathcal{M}_\Pi$ of Π and the set of observation traces of T are the same.*

5.2 Relational Analysis

Relational analysis relates two symbolic observations by means of a relation Ψ which we call a *connector*. A connector Ψ is a predicate over free variables of the symbolic observation pair. In our setting, the connector Ψ forces the equality of initial observable parts of the memory. Moreover, the connector can cope with declassification policies with no additional effort [7]. Indeed, one can refine the initial state indistinguishability relation in Def. 1 and Ψ, accordingly, to express the declassification policy.

DEFINITION 3 (RELATIONAL VALIDITY). *Consider* $obs_1 = (\phi_1, e^s_{1,1}, e^s_{1,2}, t_1)$ *and* $obs_2 = (\phi_2, e^s_{2,1}, e^s_{2,2}, t_2)$*, two symbolic observations, and a connector* Ψ*. The triple* (obs_1, obs_2, Ψ) *is* relationally valid *if*

$$\mathcal{R} := (\Psi \wedge \phi_1 \wedge \phi_2 \wedge P_O(e^s_{1,1}) \wedge P_O(e^s_{2,1})) \Rightarrow (e^s_{1,1} = e^s_{2,1} \wedge e^s_{1,2} = e^s_{2,2} \wedge t_1 = t_2) \text{ is valid}$$

Relational validity is a key property for enforcing the security condition over traces. It basically states that a pair of symbolic observations is secure if for any execution pair which initially agrees on observable memory (enforced by the connector Ψ) and reaches the observable program points (enforced by the path conditions ϕ_1, ϕ_2), if the stores are performed on observable memory (enforced by $P_O(e^s_{1,1}), P_O(e^s_{2,1})$), then they write the same values ($e^s_{1,2} = e^s_{2,2}$) at the same observable addresses ($e^s_{1,1} = e^s_{2,1}$) at the same time ($t_1 = t_2$). This implies that the observable memory is not affected by changes on the secret memory, hence the program is secure wrt. that observation pair.

Relational symbolic analysis on symbolic trees is described in Alg. 2. The algorithm takes as input a tree T, a copy T' with all variables renamed and a connector Ψ which defines the relation between variables, in the usual style of self-composition [10]. It then calls the procedure *Validity-Check* which visits the tree per levels and checks relational validity for each observation pair (the Cartesian product on sets of nodes $T(l)$ and $T'(l)$ in line 1-2). It is worth noting that the *End* node is considered as a special observation, which corresponds to normal termination. A first order oracle is used to determine the validity of the condition $\mathcal{R}$. If $\mathcal{R}$ is not valid, the oracle returns a counterexample. This corresponds to a pair of concrete initial states giving rise to a pair of concrete executions that falsify the security condition, i.e. a security attack. Otherwise, if all pairs are valid for all levels, then the program is secure.

Algorithm 2 Relational Verification on SOTs

INPUT: Symbolic Tree pair *T, T'*, Connector Ψ
OUTPUT: Secure or Insecure + Attack
1. *level* := 1
2. **Call** ValidityCheck(*T(l), T'(l),* Ψ)

ValidityCheck(*T(l), T'(l),* Ψ)
1. **For all** (TreeN, TreeN') in *T(l)* × *T'(l)*
2. $((\phi_1, e^s_{1,1}, e^s_{1,2}, t_1), (\phi_2, e^s_{2,1}, e^s_{2,2}, t_2)) :=$ *(TreeN.obs, TreeN'.obs)*
3. $A :=$ *Valid*$(\Psi \wedge \phi_1 \wedge \phi_2 \wedge P_O(e^s_{1,1}) \wedge P_O(e^s_{2,1})) \Rightarrow (e^s_{1,1} = e^s_{2,1} \wedge e^s_{1,2} = e^s_{2,2} \wedge t_1 = t_2))$
4. **If** Invalid(A) **return** *Insecure, A*
5. ValidityCheck(*T(l+1), T'(l+1),* Ψ)
6. **return** *Valid*

THEOREM 2 (TREE SECURITY). *Let* T *be a symbolic observation tree and* $\mathcal{M}_T$ *the set of observation traces associated with* T*. Then* $\mathcal{M}_T$ *is observational deterministic if Alg. 2 returns Valid in line 6.*

THEOREM 3 (SECURITY). *Let* Π *be an SiML program and* T *the symbolic observation tree obtained by running Alg. 1 on* Π *. Let also Alg. 2 return* Valid *on input* T *and connector* Ψ*. Then* $\mathcal{M}_\Pi$ *is observational deterministic if Alg. 2 returns Valid in line 6.*

EXAMPLE 1. *We demonstrate our approach using the program in Fig. 1 and omit the equivalent SiML program. Suppose all memory addresses higher than* $2KB$ *are observable by the attacker. That is* $P_O(v) = (v \geq 2048)$*. Algorithm 1 yields the symbolic observation tree depicted in Fig. 4. Each root-leaf path represents observation traces of the program. The first branch is introduced by the line 7 of the algorithm: the left path is taken if the address updated by instruction 0x0fc is observable, otherwise the right path is taken. The second branch is introduced by the conditional instruction 0x108, which updates the register* `R2` *only if the content of the memory at the address 1024 is zero (since the instruction contains the EQ suffix).*

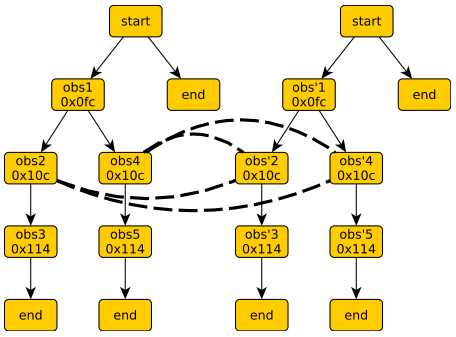

Figure 4: Symbolic Observation Trees

Suppose Alg. 1 has started with an initial symbolic configuration that bounds the i-th register to the fresh variable `Ri` *and the memory to the fresh variable* M*. The observations introduced by the instruction 0x10c are* obs_2 *and* obs_4 *according to the branch taken by the conditional instruction 0x108:*

$$obs_2 = ((M(R3) \geq 2048 \wedge M(1024) = 0), M(R3), 1)$$

$$obs_4 = ((M(R3) \geq 2048 \wedge M(1024) \neq 0), M(R3), 0)$$

Algorithm 2 takes the symbolic tree T *and a copy* T' *with all variables in symbolic observations renamed, say* obs'_i*, and the connector* Ψ*. Assuming that the registers* `R1` *and* `R3` *do not contain secret information, then* $\Psi :=$ `R1` $=$ `R1'` $\wedge$ `R3` $=$ `R3'` $\wedge (\forall v. P_O(v) \Rightarrow M(v) = M'(v))$*. The symbolic tree has four levels (excluding the root) and the Cartesian product leads to 16 cases to be considered (i.e. four for each level).*

We consider the second level of the tree and the corresponding observations obs_2 *and* obs_4 *and* obs'_2 *and* obs'_4 *as depicted in Fig. 4. In particular,* $\mathcal{R}_{2,2} = (\Psi, obs_2, obs'_2)$ *is relationally valid, while* $\mathcal{R}_{2,4} = (\Psi, obs_2, obs'_4)$ *is not. For instance, if* `R3` $=$ `R3'` $= 2048$*,* $M(2048) = M'(2048) = 2052$*,* $M(1024) = 0$ *and* $M'(1024) = 1$ *then* $\mathcal{R}_{2,4}$ *is false. In fact, the program writes into the observable address* 2052 *different values depending on the content of the secret memory address* 1024*.*

5.3 Instantiation

Relational validity requires proving the validity of the following predicate $\mathcal{R} = (\Psi \wedge \phi_1 \wedge \phi_2 \wedge P_O(e^s_{1,1}) \wedge P_O(e^s_{2,1})) \Rightarrow (e^s_{1,1} = e^s_{2,1} \wedge e^s_{1,2} = e^s_{2,2} \wedge t_1 = t_2)$ where the free variables can

include the variables used to represent the initial registers and memories (we write R_i, R_i', M and M' for registers and memories respectively). A connector Ψ is the predicate that forces the (relational) equality of initial observable parts of the memory and the equality of registers that do not contain secret information, that is $\Psi := R_1 = R_1' \wedge \cdots \wedge (\forall v. P_O(v) \Rightarrow M(v) = M'(v))$. The resulting formula is clearly not quantifier free, hence it may result difficult for automatic theorem provers. This mainly depends on the observable predicate P_O which defines the range of observable addresses. If the range is small, one can simply enumerate the addresses and introduce the constraint $\Psi = \bigwedge_{v \in P_O(v)} (M(v) = M'(v))$. Since this range can be up to 2^{32} concrete addresses (i.e. 4GB), we extract from $\mathcal{R}$ all expressions that correspond to memory accesses, $M(e)$, and instantiate e for v in Ψ. This is recursively repeated for e and for all expressions in $\mathcal{R}$. Clearly, the number of such expressions can be huge, but still bounded by the number of memory accesses in the program code.

We illustrate the instantiation process with an example. Let the $\mathcal{R}$ predicate to contain the expression $M(M(R1) + M(R2))$, then the instantiation includes the constraints (i) $P_O(R1) \Rightarrow M(R1) = M'(R1)$, (ii) $P_O(R2) \Rightarrow M(R2) = M'(R2)$ and (iii) $P_O(M(R1) + M(R2)) \Rightarrow M(M(R1) + M(R2)) = M'(M(R1) + M(R2))$. Namely, for all expressions in $\mathcal{R}$ which represent memory accesses, we generate a constraint stating that if the address is observable, then the initial memory values are the same. The constraints we generate are sufficient to conclude about relational validity.

5.4 Invariants

The approach presented so far may not terminate due to the unbounded loops that might occur in the program. We handle this issue by decorating program loops with loop invariants [20]. Let Π_{Loop} be the program slice corresponding to the loop and let the loop be uniquely identified by a pair (pc_i, pc_e). We remove all back edges to cut the loop, namely the edge from pc_e to pc_i and apply the transformations from [9], which allow to cut the loop in a sound manner.

Proving invariants in this fashion is not sufficient for relational analysis. The main reason is that the approach only accounts for state invariants and fails to capture the number of observations that might be produced in each loop iteration. Therefore, a naive application of invariants in Alg. 2 would be unsound. Moreover, state invariants may require proving functional correctness of the loop. In fact, if a variable is updated in the loop body and later it contributes to an observation, the invariant must be sufficiently strong to identify the exact value of the variable. This may not be needed for proving security, therefore we use relational invariants which are in general simpler.

We propose a modification of Alg. 1 and Alg. 2, which is correct for programs with *natural* loops [9] (no jumps escaping the loop body). The main idea is to enforce relational invariants during the analysis of Alg. 2 and ensure that the loop pair is executed the same number of times. This requires to prove that not only the invariant but also the equivalence of the branch condition pair is preserved at each iteration. We first modify Alg. 1 to create a tree T_{Loop}, which consists of the symbolic observation tree of the loop body, the branch condition B annotating the root node and the symbolic configurations C^s annotating the leaf nodes. The tree T_{Loop} is uniquely identified and represents an approximated model of traces of Π_{Loop}. Similarly, non-loop trees are extended with the corresponding symbolic configurations annotating their leaf nodes. As a result, we obtain a tree which can be a *normal* tree, i.e. labeled with symbolic observations and symbolic states on leaf nodes or a *loop* tree, which in addition contains the branch condition labeling the root node. At this point, it is possible to generalize Alg. 2 to handle loops by means of relational invariants Ψ. To illustrate this, consider a program $P := P_1; (\mathit{while}\ B\ \mathit{do}\ P_2;) P_3$ and the corresponding trees $T := T_1; T_2; T_3$ obtained as described above. Let T, T' be the input tree pair to Alg. 2, Ψ_1 be the initial connector, and Ψ_2 the relational invariant of the loop tree T_2. As for traditional invariant verification, we first check that the (relational) invariant Ψ_2 holds before the loop entry, i.e. $\Psi_1 \wedge C_1^s \wedge C_1^{'s} \Rightarrow \Psi_2$, and it is preserved by the loop body, i.e. $\Psi_2 \wedge C_2^s \wedge C_2^{'s} \Rightarrow \Psi_2'$. This is done using the symbolic configurations C_i^s, $C_i^{'s}$ from the leaves of the observation trees T_i, T_i', where Ψ_i' denotes the connector after the execution of the pair (T_i, T_i'). In addition, we enforce that $\Psi_1 \wedge C_1^s \wedge C_1^{'s} \Rightarrow (B \Leftrightarrow B')$ and $\Psi_2 \wedge C_2^s \wedge C_2^{'s} \Rightarrow (B \Leftrightarrow B')$ to ensure that the loops are executed the same number of times. Finally, Alg. 2 can be applied to the symbolic observation trees (T_1, T_1'), (T_2, T_2') and (T_3, T_3'), using the connectors Ψ_1, $(\Psi_2 \wedge B \wedge B')$ and $(\Psi_2 \wedge \neg B \wedge \neg B')$, respectively. Two different cases can be encountered during a run of Alg. 2. If a pair of normal nodes is reached, the relational validity is checked as before. If a pair of a loop node and a normal node is reached, they must be inconsistent. These conditions make our approach compositional with respect to observation traces. The process is repeated recursively for all pairs of nodes and, if successfully verified, it guarantees the security condition in Def. 1. The following example illustrates the relational verification of the UART driver routine of a separation kernel. For sake of clarity, here we reason at the C level and describe the case study more in detail later.

EXAMPLE 2. *This code snippet transforms a 32 bit integer* `n` *into a hexadecimal number and, at each iteration, notifies the UART by updating three observable addresses (line 8-10).*

```
void printf_hex(uint32_t n) {
1.    int h,  i = 32 / 4 - 1;
2.    do {
3.        h = (n >> 28); n <<= 4;
4.        if(h < 10) h += '0';
5.        else h += 'A' - 10;
6.        usart_registers *usart0 = USART0_BASE;
7.        while(usart0->tcr != 0){  ;  }
8.        buffer_out[0] = h;
9.        usart0->tpr = (uint32_t)buffer_out;
10.       usart0->tcr = 1;
11.    }while(i--); }
```

The internal loop, which we discuss later, implements a polling routine on register `tcr`*, which is externally modified whenever the UART is ready to receive the next digit. Let* $\Psi_1 = (\mathtt{n} = \mathtt{n'})$ *be the initial connector relation and* $\Psi_2 = (\mathtt{n} = \mathtt{n'} \wedge \mathtt{i} = \mathtt{i'})$ *be the relational invariant of the external loop. The connector* Ψ_1 *holds of line* 1 *by the assumption that* `n` *is low. Moreover no observations occur, hence the first part is secure. The connector* Ψ_2 *holds of the external loop (lines 11-2) since the value of* `h` *written at the low address* `buffer_out[0]` *only depends on* `n`*, which is low, while the*

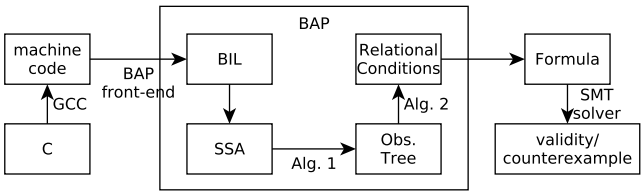

Figure 5: Verification process

next two observations are fixed constants. The loop iterates the same number of times since the value of `i` *is preserved by the relational invariant* Ψ_2*. Finally, the initial connector* (`n` = `n'`) *and the symbolic state pair at the loop entry* (`i` = `i'` = 7) *trivially entail* Ψ_2*. Observe that if* `n` *were a secret location , then* Ψ_2 = `true`*, and the verification would fail. Indeed, it is possible an execution pair goes through lines 4 and 5, and writes different values in* `buffer_out[0]`*.*

It is worth noting that the proposed verification approach ensures termination-sensitive noninterference, even for time-insensitive attacker models. We didn't find the security condition restrictive in our case studies. However, for loops without observations, one can relax the requirement on equal number of loop iterations and ensure termination-insensitive noninterference.

6. PROTOTYPE IMPLEMENTATION

We implemented the relational analysis as a new back-end for the CMU Binary Analysis Platform framework [14]. The analogies between the BAP Intermediate Language (BIL) and SiML make the implementation of the prototype tool easier after enforcing minor syntactical constraints over the input programs. For instance, we do not allow multiple write accesses from a single instruction (i.e. each BIL instruction that writes in the memory always updates 4 bytes). Implementing the analysis as a new BAP back-end provided us several additional benefits: (i) the resulting prototype tool is architecture independent, (ii) there exists a verified transformer from ARMv7 assembly to BIL [20] and (iii) we can take benefit of the existing exporters to SMT solvers. We had to reimplement the symbolic execution engine of BAP v0.7 in order to handle our case studies. Variable substitution and conditional jumps constitute the main sources of exponential blowup and thus need special care. Fig. 6 depicts the workflow of the verification process. We start from GCC compiled machine code and lift it to the BIL language. The resulting BIL program is transformed into a single static assignment form (SSA) to enable efficient symbolic analysis and avoid the expensive substitution operation. In addition, several simplification and constant propagation routines have been implemented to further speed up the analysis. Loop invariants are currently provided manually and symbolic jumps are resolved statically. This phase produces the symbolic observation tree as described in Alg. 1. Subsequently, the relational analysis module generates quantifier-free formulas for a given pair of symbolic observation trees and a connector relation, as described in Alg. 2. Finally, using existing BAP exporters, the formulas are sent to the STP solver [24], which either validates the security property or provides a counterexample which violates the policy.

In the worst case, the size of observation tree can be exponential due to the well-known path explosion problem. We leverage standard optimizations such as expression substitution and constant propagation to reduce the tree size when possible. In addition, one can make use of the CFG to further control the exponential blow-up.

However, scalability issues are expected when fully verifying machine code. Consider symbolic jumps or symbolic memory. A bug-finding tool would simply concretize the symbols and continue the analysis. For verification one has to resolve all possible concretizations or carry the symbolic constraints throughout the analysis.

7. CASE STUDIES

The tool has been used to verify several programs. In all cases, the analysis has been performed directly on the ARMv7-A machine code produced by the GCC compiler. The benchmark of these experiments is summarized in Table 1. Among other statistics we report the memory footprints and the number of SOT nodes to get an understanding of how big a piece of code we can currently check. Here, we summarize three case studies: (i) the IPC syscall of a separation kernel, (ii) a UART device driver and (iii) a modular exponentiation routine used by crypto services. The case studies are taken from real software. Our experience shows that small programs (order of 1000s instructions) are perfectly reasonable in high assurance contexts, and well within the reach of our tool with some standard engineering.

7.1 Case Study 1: Send syscall

The target separation kernel is a low level execution platform for ARMv7. The kernel implements minimal functionalities and consists of 1028 machine code instructions, mixing hand written assembly with GCC optimized output. The kernel must execute the partitions in isolation and control the communication appropriately. Each partition is allowed to access a non-overlapping part of the system resources: (a) a contiguous part of the physical memory, that contains the partition's executable and data, (b) the logical message box, stored in the kernel memory and, (c) the virtual registers, which are stored in the kernel memory while the partition is suspended, and are stored in the standard registers while the partition is active.

The IPC mechanism is provided by the "send" syscall; first the active partition (the sender) stores the message in the register `R1` and raises a software interrupt, then the kernel handler stores the message in the message box of the receiver and restores the sender. While executing, the kernel backs up the sender's CPU state into its own memory and restores it when the syscall terminates. To appropriately control the communication, the kernel must ensure that: (1) the sender infers no information about the receiver and (2) the receiver only infers the content of sender's register R1 (the delivered message) and nothing more.

The above requirements have been verified by executing the relational analysis of the "send" syscall twice; considering observable the resources allocated to either the sender and the receiver. The resources allocated to the observing agent directly drive the definition of the connector relation (consisting in 30 lines of statements). Moreover, to take into account the designed declassification, the initial connector guarantees that the value of `R1` is equal in the two initial configurations. In the other experiments, we have modified the preconditions to test the tool with non-secure versions of the send syscall.

Software	ARM (LOC)	BIL (LOC)	Tool (Sec)	SMT (Sec)	Memory (Byte)	SOT Size (Nodes)	Secure (Y/N)
send syscall sender	80	1017	220	17	599M	31	Y
send syscall receiver	80	1017	220	16	599M	31	Y
send syscall sender 1	80	753	27	16	77M	31	N
send syscall sender 2	80	1015	215	18	599M	31	N
UART print char	13	100	1	1	85K	3	Y
UART print hex	32	305	5	1	29 M	7	Y
UART print bin	30	286	2	1	30M	7	Y
Exp. timing	19	151	1	1	319K	8	Y
Exp. timing	19	160	1	1	463K	7	N

Table 1: Experimental results

The absence of loop in the syscall freed us from defining the corresponding invariants. We also took benefit from the existing results obtained by a previous verification of functional correctness of the kernel: (i) the resolution of indirect jumps, (ii) the identification of data structures invariants and (iii) the analysis of constant parts of the memory.

These results reduced the set of reachable code of the syscall to 80 instructions (which corresponds to reducing the BIL code from 10K lines to 1017 lines) and provided us the necessary handler precondition (consisting of 400 lines of statements).

7.2 Case Study 2: UART device driver

The UART (Universal asynchronous receiver/transmitter) is a hardware device for communication over a serial interface. The driver is implemented in C and resembles the functionality of the well known `printf`. We limit our verification to the low level interface of the driver. Example 2 provides the C code that sends to the UART a 32 bit integer in the form of an hexadecimal number. The function contains two loops: the outer loop computes the eight hexadecimal digits of the number, the nested loop polls on the device register `tcr` before writing the current digit to the UART.

The function binary code consists of 32 instructions, that are lifted to 305 BIL lines. Initially the parameter `n` is represented by the register `R0` and the outer loop uses the register `R4` and `R5` to store the variable `n` and `i`, respectively. Since the UART delivers the written characters to the external world, we consider observable the UART registers (64 bytes starting from `USART0_BASE`) and the DMA buffer (32 bytes starting from `buffer_out`). Moreover, since the input "must be sent" to the UART, we consider non secret the initial value of `R0`.

We first verified the nested loop. This fragment polls on the device register `tcr`, which is updated externally by the device driver. To emulate this external effect on the system memory, we inline the behavior of the device in the loop body. At each iteration a shadow variable `tcrWait` (which models an oracle knowing the number of iterations needed by the UART to receive a message) is decremented and `tcr` is resetted if the `tcrWait` value is zero. The given relational invariant `tcrWait=tcrWait'` states that the oracle provides the same answer in both executions. The tool automatically checks that the relational invariant is preserved and that the loop conditions are equivalent in both configurations. Notice that the nested loop does not produce observations, thus its verification is required only to guarantee termination sensitive noninterference.

Next we verify the outer loop. The relational invariant, `R4=R4' & R5=R5' & tcrWait=tcrWait'`, states that the values of `n`, `i` and the oracle answer are consistent in both configurations. The tool automatically checks that the relational invariant is preserved, the loop conditions are equivalent, the nested loop invariant is satisfied and the relational validity of the three observations (the update of the output buffer and the two UART registers). The relational verification is significantly simpler than the functional (total-)correctness of the loop; the relational invariant does not need to relate the values of `n` and `i`, the value of `h` is not constrained and no variant is needed.

The last verification step must ensure that starting from the initial connector Ψ, the condition of the outer loop is equivalent in both configurations and that the outer invariant is established. The initial connector Ψ simply relates the integer sent to the UART (the initial value of `R0`) and the content of the observable memory. The tool spotted that without further preconditions the code is not secure: before using the registers `R3`, `R4` and `R5` to represent the local variables, the function pushes their initial (secret) values on the stack. This yields an non (relationally) valid observation if the stack pointer is unconstrained.

7.3 Case Study 3: Modular exponentiation

The modular exponentiation routine is a simplified version of the case studies in [34]. The authors provide two programs with the same functionality. The insecure version branches on the secret boolean variable `i`, while the secure version computes the results independent of the branch condition, stores them in an array `A`, and returns `A[i]`. We point out that it is critical to verify this code at the machine level, as the compiler can perform optimizations that break the security.

We assume the attacker can only observe the execution time of the two routines. The elapse of time is modeled by a shadow variable, which is incremented for each instruction, following the functional time model described earlier. The tool detects the control-flow side channel of the non-

secure routine and validates the secure one, even if we do not require program counter equivalence between every pair of possible configuration.

8. DISCUSSION AND RELATED WORK

Formal Verification of Low Level Code. Related kernel information flow verification efforts have been reported recently by several authors. For instance, [36] showed a noninterference property of the seL4 microkernel, essentially reducing to show absence of information flow from the scheduler to the next scheduled thread state. Similarly, [19] established system-level information flow security of a simple hypervisor at the level of ARMv7 assembly, by proving a trace equivalence property with respect to an ideal model that reflects the isolation properties that are desired of the hypervisor. Other machine code verification work includes the work by Heitmeyer et al. [27] and a series of works on the INTEGRITY kernel [41]. All these works use interactive theorem proving (ITP) techniques to establish the desired security property and consequently require serious manual effort. By contrast, this paper shows that automatic verification of small kernel routines is possible with less effort. Formal verification of device drivers has been applied to serial interfaces such as UART and USB devices. These works focus on functional correctness and ignore information flow properties. Moreover, the verification task is performed at C level [3, 35] or uses ITP [23].

Relational Verification. Relational program verification has been used to prove non-functional properties such as compiler optimization correctness [10], program equivalence [39, 38] and information flow security [11, 8, 30]. Neither addresses verification at the machine level. Barthe et al. [11] introduce self-composition as a method for checking 2-safety properties, including information flow. A related paper [10] presents product programs as a mean for reducing relational verification to classical functional verification. Several authors have studied algorithms for constructing and verifying over/under approximations of product programs automatically using typing [47], abstract interpretation [30, 38] and symbolic execution [8, 39, 44]. This work differs in several aspects. First, prior works do not consider timing channels and trace-based observations, giving rise to weaker security guarantees and simpler computational models. Second, we combine unary and relational analysis to avoid the expensive construction of the product program and reuse previously computed results. The unary analysis extracts necessary program dependences, while the relational one performs the verification. Breadth first search algorithms enable the alternation of these steps and allow efficient verification on the fly. As our case studies show, machine code verification requires flow and path sensitive techniques due to register reuse and complex data/control flow. Hence, compared to [30, 38], our techniques is more precise. Third, our approach addresses additional complications due to the lack of support for data structures. For instance, the secret state cannot be tied to program variables, and it may depend on complex pointer arithmetic. Consequently, it is unknown a priori whether an instruction accesses a secret or public memory location. Recently, Caselden et al. [17] presented a way to recover a hybrid information flow/control flow graph using trace based analysis of machine code. This graph is used to find paths that trigger a given vulnerability condition. We find this work relevant and believe that their ideas can be applied to our setting and speed up the symbolic execution. However, the technique requires structural knowledge which one may not always have and ignores timing channels.

Timing. Eliminating timing channels by purely software approaches is difficult due to architecture dependent features such as caches, pipelines and more[1]. However, the timing constraints generated by our analysis can be used as a software contract to be enforced by hardware features. Our execution time model is history-independent and deterministic. We can not precisely represent the effect of caches and pipeline data dependencies. However, we can verify absence of side channels for simple architectures (e.g. ARM Cortex-M) or under the assumption that the attacker is not able to access to information that are affected by the wall-clock. Moreover, symbolic analysis provides memory access patterns which can be later validated wrt. a given architecture model. The model is similar to [26], which considers timing analysis for JavaCard-like bytecode. Molnar et al. [34] introduce the notion of PC-security which can avoid control flow side channels for crypto operations. Their security model can be easily accommodated in our work. Several authors propose mitigation [49] and padding [2] techniques to address timing channels. The results produced by our analysis are complementary to mitigation and padding. Indeed, they can be combined with mitigation to reduce the leakage bandwidth or to enable the required padding. We point out that worst case execution time is insufficient to verify that the execution time is independent of the secret, although it can remove information flows using mitigation. Finally, our model is suitable for systems where the external scheduler is instruction-based.

Loop Invariants. Finding loop invariants is definitely the most time consuming verification task and, for tricky examples, this is inherited by our works as well. However, relational invariants are in general simpler than functional invariants. We only need to enforce that the loop pair has the same low memory effects, without saying what these effects are. Recent work considers automatic generation of relational loop invariants for machine code using data driven techniques [44]. After executing the programs a certain number of times, concrete memory and register values are used to determine linear equality relationships between variables. Our approach can be used to check if the inferred invariants are sufficient to enforce equivalence for traces. The fact that we consider traces makes automatic invariant generation harder since traces do not compose in general. In [43], Saxena et al. introduce loop-extended symbolic execution which relates number of iterations of different program loops. This can be used to make our trace analysis more precise, although it was not needed in our case studies.

Security Analysis For Machine Code. Security analysis for machine code is a well studied research area [6]. The majority of works focus on bug finding techniques for malware analysis, vulnerability checking, automatic exploit generation and more [37, 5, 18, 16]. Typically, they use typing, taint analysis or lightweight symbolic execution to ensure good path coverage and still maintain scalability. Other works take a more formal approach to machine code verification [40]. All these approaches fail to capture the information flows considered in this paper. Our focus is on full verification of small kernel handlers and device drivers. We admit that scalability remains an issue for larger programs and the techniques used by cited works can improve our tool.

Information Flow Analysis. Information flow has been pervasively applied to software security using static and dynamic verification techniques [42, 31]. If applicable, security type systems (TS) would be very efficient. Unfortunately, none of our case studies can be handled with TSs, at least not without significant modification. There are several case where a TS approach would fail: (i) Low observations are memory writes of shape $M[R_i] = exp$, hence dataflow analysis is needed to determine the values of R_i to know which address is updated. (ii)TSs donâĂŹt support low memory writes under high branches. (iii)Our case studies use preconditions that guarantee certain invariants (describing the execution context). Data/control flow analysis is needed to determine the observations enabled in those contexts. (iv)Since traces do not compose in general, a global analysis through the symbolic trees is needed. (v)Unreachable or semantically secure code is also problematic for TSs. (vi)Declassification can be challenging and the timing analysis may require the TS to perform symbolic computation.

The verification approach for trace-based information flow analysis of ARMv7 machine code is novel. We leverage symbolic execution to reduce relational verification to automatic theorem proving of quantifier-free formulas. We are not aware of any tool that performs full verification of information flows for machine code.

Self-Modifying Code. Our analysis requires the code to be non self-modifying. For programs executed in user space this property is usually enforced at run-time by the underlying OS, for instance by configuring as non-writable the virtual memory containing the program code. On the other hand, privileged code can dynamically change its behavior by writing into its instruction memory, changing the coprocessor registers that control the MMU or updating the page tables. Considering these events as observable enables our analysis to verify that privileged code is non self-modifying. This can be done by checking that the corresponding symbolic observation tree is empty. It also enables the use of relational analysis for low level code that does not reconfigure the memory layout, but it accesses protected memory areas in privileged mode (e.g. the send syscall accesses the message boxes stored in the kernel memory) or performs privileged instructions (e.g. the syscall accesses the ARM banked registers to back up and restore the CPU context of the interrupted partition).

9. CONCLUSIONS

We presented a novel approach to relational verification for machine code. A distinguishing feature of our proposal is the ability to precisely verify information flow properties in the presence of features like a preemptive execution environment and memory mapped devices. We have implemented a tool and verified several real world case studies, including separation kernel routines and device drivers. This shows that information flow analysis for security critical routines is not only important, but also feasible.

There are several challenges we leave out as future work. The technique introduced in Alg. 1 can be combined with a security type system to automatically infer and refine types. This would improve the relational analysis by using Alg. 2 only when the typing fails. Timing is particularly critical to apply our approach to real processor architecture. We also are confident that, due to their simplicity, relational invariant generation can be automated. Other future plans include engineering the tool and improving on symbolic execution.

Acknowledgments

The authors thank the anonymous reviewers for valuable comments. This work is supported by framework grant "IT 2010" from the Swedish Foundation for Strategic Research.

10. REFERENCES

[1] O. Aciiçmez and Çetin Kaya Koç. Microarchitectural attacks and countermeasures. In *Cryptographic Engineering*, pages 475–504. 2009.

[2] J. Agat. Transforming out timing leaks. In *POPL*, pages 40–53, 2000.

[3] E. Alkassar, M. A. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev. Formal device and programming model for a serial interface. In *VERIFY*, 2007.

[4] ARMv7-A architecture reference manual.

[5] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. In *NDSS*, 2011.

[6] G. Balakrishnan and T. Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32:23:1–23:84, August 2010.

[7] M. Balliu, M. Dam, and G. L. Guernic. Epistemic Temporal Logic for Information Flow Security. In *Proceedings of the ACM SIGPLAN Programming Languages and Analysis for Security*, june 2011.

[8] M. Balliu, M. Dam, and G. L. Guernic. ENCoVer: Symbolic Exploration for Information Flow Security. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pages 30–44, june 2012.

[9] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87, 2005.

[10] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, pages 200–214, 2011.

[11] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[12] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In *Proceedings of the 16th European Conference on Programming*, ESOP'07, pages 125–140, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] J. Brown and T. F. Knight Jr. A minimal trusted computing base for dynamically ensuring secure information flow. 2001.

[14] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *CAV*, pages 463–469, 2011.

[15] D. Brumley, I. Jager, E. J. Schwartz, and S. Whitman. The bap handbook. `http://bap.ece.cmu.edu/doc/bap.pdf`, October 2013.

[16] D. Brumley, H. Wang, S. Jha, and D. X. Song. Creating vulnerability signatures using weakest preconditions. In *CSF*, pages 311–325, 2007.

[17] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song. Hi-cfg: Construction by binary analysis and application to attack polymorphism. In *ESORICS*, pages 164–181, 2013.

[18] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[19] 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 *ACM Conference on Computer and Communications Security*, pages 223–234, 2013.

[20] M. Dam, R. Guanciale, and H. Nemati. Machine code verification of a tiny arm hypervisor. In *TrustED@CCS*, pages 3–12, 2013.

[21] A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hritcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *POPL*, pages 165–178, 2014.

[22] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.

[23] J. Duan and J. Regehr. Correctness proofs for device drivers in embedded systems. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[24] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[25] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Los Alamitos, Calif., 1982. IEEE Comp. Soc. Press.

[26] D. Hedin and D. Sands. Timing aware information flow security for a javacard-like bytecode. *Electr. Notes Theor. Comput. Sci.*, 141(1):163–182, 2005.

[27] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. Mclean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *CCS*, pages 346–355. ACM, 2006.

[28] HOL4. `http://hol.sourceforge.net/`.

[29] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your ifcexception are belong to us. In *IEEE Symposium on Security and Privacy*, pages 3–17, 2013.

[30] M. Kovács, H. Seidl, and B. Finkbeiner. Relational abstract interpretation for the verification of 2-hypersafety properties. In *ACM Conference on Computer and Communications Security*, pages 211–222, 2013.

[31] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[32] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992.

[33] R. Medel, A. Compagnoni, and E. Bonelli. A typed assembly language for non-interference. In *Proceedings of the 9th Italian Conference on Theoretical Computer Science*, ICTCS'05, pages 360–374. Springer-Verlag, 2005.

[34] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC*, pages 156–168, 2005.

[35] D. Monniaux. Verification of device drivers and intelligent controllers: a case study. In *EMSOFT*, pages 30–36, 2007.

[36] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. In *CPP*, pages 126–142, 2012.

[37] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[38] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *SAS*, pages 238–258, 2013.

[39] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *CAV*, pages 669–685, 2011.

[40] T. W. Reps, J. Lim, A. V. Thakur, G. Balakrishnan, and A. Lal. There's plenty of room at the bottom: Analyzing and verifying machine code. In *CAV*, pages 41–56, 2010.

[41] R. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer US, 2010.

[42] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE J. on Selected Areas in Communications*, 21(1):5–19, 2003.

[43] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *ISSTA*, pages 225–236, 2009.

[44] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken. Data-driven equivalence checking. In *OOPSLA*, pages 391–406, 2013.

[45] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, Dec. 2008.

[46] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ACM SIGOPS Operating Systems Review*, volume 38, pages 85–96. ACM, 2004.

[47] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, pages 352–367, 2005.

[48] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps. Directed proof generation for machine code. In *CAV*, pages 288–305, 2010.

[49] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, pages 99–110, 2012.