<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper presented at *TrustED*.

Citation for the original published paper:

Dam, M., Guanciale, R., Nemati, H. (2013)
Machine Code Verification of a Tiny ARM Hypervisor.
In: ACM Press

N.B. When citing this work, cite the original published paper.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136351>

Machine code verification of a tiny ARM hypervisor

Mads Dam, Roberto Guanciale, Hamed Nemati
KTH Royal Institute of Technology
Stockholm, Sweden
{mfd, robertog, hnnemati}@kth.se

ABSTRACT

Hypervisors are low level execution platforms that provide isolated partitions on shared resources, allowing to design secure systems without using dedicated hardware devices. A key requirement of this kind of solution is the formal verification of the software trusted computing base, preferably at the binary level. We accomplish a detailed verification of an ARMv7 tiny hypervisor, proving its correctness at the machine code level. We present our verification strategy, which mixes the usage of the theorem prover HOL4, the computation of weakest preconditions, and the use of SMT solvers to largely automate the verification process. The automation relies on an integration of HOL4 with BAP, the Binary Analysis Platform developed at CMU. To enable the adoption of the BAP back-ends to compute weakest preconditions and control flow graphs, a HOL4-based tool was implemented that transforms ARMv7 assembly programs to the BAP Intermediate Language. Since verifying contracts by computing the weakest precondition depends on resolving indirect jumps, we implemented a procedure that integrates SMT solvers and BAP to discover all the possible assignments to the indirect jumps under the contract precondition.

Categories and Subject Descriptors

D4.6 [Operating Systems]: Security and Protection; D.2.4 [Software Engineering]: Software/Program Verification—*formal methods, correctness proofs*

Keywords

Binary verification; Hypervisor

1. INTRODUCTION

Designing secure systems requires isolation of the security critical components, guaranteeing that only allowed communication can take place among them and preventing propagation of component failures. The isolation can be achieved

by executing the software components inside partitions running on top of hypervisors and separation kernels, which are low level execution platforms providing isolated partitions on shared resources.

Achieving component isolation by adopting a low level software platform allows us to reduce the hardware complexity and costs and increase modularity compared to solutions based on dedicated hardware devices. However, a key requirement of this kind of solution is the formal verification of the trusted computing base, preferably at the binary level. In fact, machine code verification obviates the necessity of trusting the compilers. Moreover, low level programs mix structured code (e.g. implemented in C) with assembly and use instructions (e.g. mode switches and coprocessor interactions) that are not part of a high level language making it difficult to use verification tools that target user level code.

This paper describes the machine code verification of the PROSPER separation kernel (i.e. hypervisor) for ARMv7 [19, 2] and the tool set developed to automate this process. The machine code verification is part of a wider verification effort that guarantees the main security property of our system. This was to a large extent accomplished in the HOL4 theorem prover [11] using the provided ARM model [8] summarized in Section 3. This exercise showed that the execution traces of arbitrary software components (guests) running on top of the hypervisor are the same as the traces obtained when executing the same guests on physically separated machines, allowing communication to flow only along known communication channels (with one small proviso, a bidirectional hypercall channel). The overall verification strategy is described in the companion paper [7]. Section 2 describes the hypervisor design and the machine code verification strategy. It also summarizes how the results of the machine code verification are used to guarantee the security property.

The overall verification strategy takes the execution of complete hypervisor code fragments (handlers) as atomic, using an instruction counter as an (admittedly fairly crude) measure of time consumption, and specified as a set of contracts. The machine code verification is in charge of establishing if the hypervisor code fragments respect these contracts. A contract is expressed as Hoare triple, stating that if the starting state satisfies the precondition, the execution of the hypervisor assembly fragment terminates and establishes the postcondition. To automate the verification of loop-free assembly fragments we compute the weakest precondition on the initial state, which is the weakest condition on the initial state ensuring that the execution of the fragment terminates

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

TrustED 13, November 4 2013, Berlin, Germany.

Copyright 2013 ACM 978-1-4503-2486-1/13/11..\$15.00.

<http://dx.doi.org/10.1145/2517300.2517302>.

in a state satisfying the postcondition. Then we prove that the precondition entails the weakest precondition.

The weakest precondition of the ARM assembly fragments is obtained using the Binary Analysis Platform (BAP) [5]. BAP utilities (see Section 4) reason on the BAP Intermediate Language (BIL), that is a small and formally specified ML-like language reducing complex instruction effects to atomic register accesses. Section 5 presents one of the major contributions of this paper: a new BAP front-end that transforms ARMv7 assembly programs to BIL programs. This tool is available to be used in other projects that aim to use BAP on ARM assembly code.

Section 6 presents the other tools developed to accomplish the hypervisor verification; the integration of existing GNU utilities, the optimization of the BAP weakest precondition algorithm and the development of a procedure to solve indirect jumps.

We conclude by demonstrating the verification of the hypervisor call (i.e. hypercall) in charge of providing message passing (Section 7) and reporting the costs and measures of the verification process (Section 8).

2. VERIFICATION OF THE HYPERVISOR

The target of the verification is a tiny hypervisor for the ARMv7 architecture that allows the execution of two guests on top of a single physical machine. The two guests are statically allocated into two non-overlapping partitions of the memory, allowing us to simplify the setup of the memory management unit. The hypervisor is in charge of isolating the two guests, providing a communication mechanism among them and implementing a round-robin scheduling, which suspends the active guest and resumes the other one. The hypervisor is designed to provide a paravirtualization platform, that is, a guest operating system must be explicitly ported to use the APIs provided by the hypervisor. Moreover, the hypervisor is the only software executing in privileged modes and all guest activities are executed in user mode. Our verification strategy is based on two main intuitions: the introduction of a top level specification to define the security properties and the notion of observation based bisimulation.

The top level specification (TLS) (which is also called “ideal” system) formalizes the hypervisor functionality. The ideal system resembles a distributed environment consisting of two separate ARMv7 machines that communicate via asynchronous message passing. Each machine has its own registers, flags, coprocessor registers and memory. Each machine of the ideal system is used to execute one of the two guests in a physically isolated environment. Moreover, an external component holds two message boxes that provide a communication device and contain the messages exchanged between the machines. The execution of the two machines is interleaved and externally scheduled, namely at each time only one machine is active and can execute an instruction.

The machines exploit two “special” CPUs: ARMv7 processors that atomically execute a kernel functionality when a privileged mode is activated. On the other hand, while the CPU is in the user mode the execution of an instruction behaves equivalently as on a standard CPU, without affecting the state of the “non-active” machine or the environment.

The goal is to verify that the hypervisor correctly mimics the distributed system, i.e. no guest is able to infer any differences between its execution on top of the hypervisor

Figure 1: Relation the real system and the TLS

and on top of the ideal settings. We accomplish this task by exhibiting a bisimulation relation between ideal and real states.

Figure 1 provides an intuitive representation of the bisimulation relation, by depicting the executions of two guests on top of the hypervisor (A) and on top of the ideal system (B). The solid arrows represent the state transitions. White (grey) circles represent states where the first (second) guest is active, respectively, while black circles represent states in privileged modes (i.e. states where the hypervisor is active in the real world). The state transitions can be classified in four groups: (i) transitions between two states is user mode (e.g. $1 \rightarrow 2$) represent executions of instructions that do not cause exceptions (ii) transitions between a user state and a privileged one (e.g. $2 \rightarrow 3$) represent the reception of an interrupt or an exception (i.e. the execution of a software interrupt) (iii) transitions that start from a privileged mode in the real world represent hypervisor activities (iv) transitions that start from a privileged mode in the ideal world represent the atomic execution of an ideal functionality. Intuitively, internal kernel states can not be observed by the guests, since during kernel execution no guest is active. Moreover, since the hypervisor does not support preemption we can easily disregard its internal states, since an hypervisor handler can never be interrupted by a nested exception or interrupt. For these reasons only states in user-mode (the mode used to execute guests) are constrained by the bisimulation relation, which is depicted by the dotted lines.

The hypervisor verification [7] is accomplished in three steps: (i) verification of a general security lemma that guarantees properties of user transitions when the Memory Management Unit is correctly configured by the hypervisor bootstrap [12] (ii) verification of a top level theorem that transforms the relational reasoning into a set of contracts for the hypervisor handlers and guarantees that the bisimulation is established if all contracts are satisfied (iii) the machine code verification that checks the contracts of the handlers and the bootstrap. The present paper describes the third step, which is in charge of establishing properties of transitions in the real world from privileged states to user states (e.g. from 2 to 5).

The interface between the bisimulation proof and the hypervisor code verification relies on Hoare logic. The general proof requires the verification contracts expressed as Hoare triples $\{P\}C\{Q\}$. The binary code verification checks that the hypervisor assembly fragment C establishes the postcondition Q if the precondition P is met. This task is simplified by the hypervisor design choices: (i) the hypervisor does not

exploit self modifying code, allowing us to statically know the behavior of the hypervisor activities and simply verify that no memory update affects the code memory area (ii) the minimal set of functionality allowed the exception handlers to be loop free (iii) the loops (present only in the bootstrap) have always a structured control flow, with a conditional instruction that represents both the enter and exit point of the loop (iv) there are no nested exceptions, since hypervisor exception handlers keep hardware interrupts disabled until the user mode is restored, allowing us to ignore intermediate states. These properties make the adoption of a semi-automatic strategy feasible. We compute the “weakest” precondition of loop-free assembly fragments on the initial state to ensure that the execution of the fragment terminates in a state satisfying the postcondition. Then we prove that the precondition entails the weakest precondition. For the hypervisor code fragments that contain loops (i.e. the bootstrap) we manually split the code into smaller parts and define loop invariants. Standard Hoare rules for sequential composition and structured loops are used to verify the correctness of the hypervisor code. The contract and invariant for the identified loop-free fragments are verified by computing the weakest precondition, while their composition is manually verified using the HOL4 theorem prover.

We developed HOL4 inference procedures specific for our hypervisor structure. These procedures allow us to automatically massage the contracts before the computation of the weakest precondition. The output of the procedures is a proof that guarantees that the original contract $(\{P\}C\{Q\})$ is satisfied if a “simplified” contract $(\{P'\}C\{Q'\})$ is met, that is:

$$\begin{aligned} & \forall \text{prestate, poststate.} \\ & (P'(\text{prestate}) \Rightarrow Q'(\text{prestate, poststate})) \\ & \Rightarrow \\ & (P(\text{prestate}) \Rightarrow Q(\text{prestate, poststate})) \end{aligned}$$

This task is necessary to generate contracts having no universally quantifier in the precondition, allowing us to automate the verification of logical consequences (e.g. checking that a precondition entails a weakest precondition) using Satisfiability Modulo Theory (SMT) solvers that support bitvectors (in our case STP [9]).

3. THE ARM HOL4 MODEL

The general security lemmas that guarantee properties of user transitions and the bisimulation proof (that depends on the hypervisor contracts) have been verified in the theorem prover HOL4, using the ARMv7 architecture model provided by Fox et al. [8, 14]. The new BAP front-end described in Section 5 uses the same model, allowing the machine code verification and the other verification tasks to rely on the same semantics of ARMv7 instructions. The ARMv7 model is constructed from the pseudocode described in the ARM specification and provides a detailed formalization of the effects of the instructions, taking into account the different execution modes, flags etc.

The system behavior is modeled by the transition function $s' = \text{arm_next}(s)$, which describes that the ARM state s reaches the state s' after executing an instruction. An ARM state is a tuple $s = \langle \text{regs}, \text{psrs}, \text{coregs}, \text{mem} \rangle$. Here, regs represents a sequence of 32-bit registers, where $\text{regs}(i)$ is the i -th register and $\text{regs}(15)$ plays the role of program counter. The tuple $\text{psrs} = \langle \text{cpsr}, \text{psr}_{\text{svc}}, \text{psr}_{\text{abort}}, \text{psr}_{\text{undef}}, \text{psr}_{\text{irq}} \rangle$ contains five 32-bit “program status” registers: where

cpsr denotes the current program status and each psr_m indicates the saved program status in the mode m . A program status register encodes: the arithmetical flags, the execution mode (i.e. $M(\text{psr})$), the interrupt disabling, the instruction decoding (i.e. $T(\text{psr})$ and $J(\text{psr})$ to represent the Thumb and Jazelle mode respectively) and the active endianness (i.e. $E(\text{psr})$). The function $\text{mem} \in 2^{32} \rightarrow 2^8$ represents the 32-bit addressable memory. The tuple $\text{coregs} = \langle c1, c2, c3 \rangle$ contains the three 32-bit registers of the coprocessor 15, that are used to control the Memory Management Unit.

The HOL4 model also provides a mechanism to statically compute the effects of an instruction via the arm_steps function. Let m be a CPU mode, j and t be the Jazelle and Thumb decoding flag (both j and t are false for the standard ARM instruction decoding), e be the the encoding of an instruction, then $\text{arm_steps}(m, j, t, e, \text{inst})$ returns the possible execution steps $[st_1, \dots, st_n]$. Each step $st_i = (c_i, t_i)$ consists of the condition c_i that enables the transition and the function t_i that transforms the starting state into the next state: for each possible state $s \in S$ such that $M(s.\text{cpsr}) = m \wedge J(s.\text{cpsr}) = j \wedge T(s.\text{cpsr}) = t \wedge E(s.\text{cpsr}) = e$ and $\text{fetch}(j, t, e, s) = \text{inst}$, if $c_i(s)$ holds then $\text{arm_next}(s) = t_i(s)$. Notice that the instruction fetching depends on the instruction decoding and endianness. For standard ARM decoding and little-endian setup the function fetch reads four bytes from memory starting from the address pointed to by the program counter.

The theorems produced by arm_steps are intended to be used as input to other automated tools. In our case, they are the main building block of our ARM front-end for BAP. In fact, implementing the new front-end on top of this model allows us to enable the BAP analysis tools without trusting an ARM assembly semantic potentially different from the one used by the bisimulation theorem.

4. THE BINARY ANALYSIS PLATFORM

The Binary Analysis Platform (BAP) [5] is a framework providing different binary program analysis techniques and algorithms. The BAP architecture is divided into front-ends and a set of back-end utilities which interact through the BAP Intermediate Language (BIL). BIL is a small and formally specified language, designed to explicitly represent all the side-effects of executing machine instructions, thus simplifying the development of the analysis tools. Since BIL is architecture independent, the analysis tools can be developed regardless of the target platform and used for several machine architectures. In order to enable BAP to reason about a specific machine architecture, an architecture dependent front-end transforms the machine code of a given programs to BIL programs.

The back-end of BAP provides utilities to extract control flow graphs and program dependence graphs, to perform symbolic execution and to compute weakest preconditions. The weakest precondition algorithm provides an effective method to speed up the verification of loop-free assembly fragments, by reducing the problem of verifying Hoare triples to the problem of proving first-order formulas.

In order to clarify the target language of the new BAP front-end, presented in Section 5, in the following we summarize the main constructs of the BIL language.

4.1 BIL

A BIL program is a sequence of statements. Each statement (Figure 2) can affect the system state by (i) assigning the evaluation of an expression to a variable, (ii) (conditionally or unconditionally) modifying the control flow, (iii) terminating the system in a failure state if an assertion does not hold and (iv) unconditionally halting the system in a successful state. As usual, labels are used to refer to the specific locations in the program and can be the target of jump statements.

```

program ::= stmt*
stmt    ::= var := exp | jmp(exp) | cjmp(exp, exp, exp)
          | assert(exp) | halt(exp) | label(string)

```

Figure 2: Syntax of BIL programs

The BIL data types are boolean, words (e.g. $u32$) and memories ($u32 \rightarrow u8$). The latter are represented as a function that takes a 32-bit address and returns a byte. The main constituent of BIL statements are expressions, whose syntax is depicted in Figure 3. Here, constants (cst), binary ($\diamond_b$) operators, unary ($\diamond_u$) operators and type casting function ($cast(cast_kind, \tau_{reg}, exp)$) have the standard meaning and the variables (var_{mem} and var_{reg}) are used to represent the system state.

The expression $load32(var_{mem}, exp)$ reads thirty two bits from the memory referred by var_{mem} starting from the address exp , yielding a value of type $u32$. Similarly, the functions $load16$ and $load8$ read sixteen and eight bits from the given memory respectively.

The expression $store32(var_{mem}, exp_1, exp_2)$ returns a new memory in which all the locations have the same values as the initial memory except the addresses $exp_1 + i$ where $i \in \{0, 1, 2, 3\}$ that contain the chunks of exp_2 . Similarly, the functions $store16$ and $store8$ update two and one bytes of the input memory respectively.

5. THE NEW BAP FRONT-END

The existing BAP front-end to lift ARM programs to BIL supports only ARMv4, it lacks the management of processor status registers, it does not handle banked registers for privileged modes and coprocessor registers. This makes the front-end inadequate for verification of ARMv7 architecture machine code and requires the development of a new front-end. The new BAP front-end for ARMv7 is called Lifter and converts an ARM assembly fragment to a BIL program. The lifter module is built on top of the *arm_steps* library of the HOL4 ARM model. Consequently, the soundness of the transformation from an ARM assembly instruction to its corresponding BIL program relies on the correctness of the ARM model inside HOL4.

The translation procedure involves the following steps, (i) mapping HOL4 ARM states to BIL states and (ii) for each instruction of the given assembly fragment producing the BIL fragment that emulates the *arm_steps* outputs. To map an ARM state to the corresponding BIL state we use a straightforward approach. A BIL variable is used to represent a single component of the machine state: for example, the variable $R0$ represents the register number zero and the variable MEM represents the system memory. In particular, instead of using a 32-bit variable to represent a program status register, we use a different variable for each flag (i.e.

the variable Z represents the “zero flag” of the current program status register and the variable Z_SVC represents the “zero flag” of the saved program status register for the mode *svc*).

5.1 Lifting instructions

To transform an ARM instruction to the corresponding BIL fragment we need to capture all the possible effects of its execution in terms of affected registers, flags and memory locations. The generated BIL fragment should simulate the behavior of the instruction executed on an ARM machine.

Given an instruction, the output of the *arm_steps* library is a list of steps of the form (c_i, t_i) (see Section 3). Therefore, to obtain a BIL fragment for an instruction we need to translate the predicates c_i and their corresponding transformation functions t_i . This task is accomplished using symbolic evaluation of the predicates and the transformation functions. The input of the evaluation is a symbolic state in which independent variables are used to represent each state register, flag, coprocessor register and memory. This approach allows us to obtain a one-to-one mapping between the symbolic state variables and the BIL state variables. To transform a predicate c_i , we apply the predicate to a symbolic ARMv7 state, thus obtaining a symbolic boolean expression in which free-variables are a subset of the symbolic state variables. Similarly, to map a transformation function t_i , we apply t_i to a symbolic state, thus obtaining a new state in which each register, flag and affected memory location is assigned a symbolic expression.

The following algorithm illustrates the procedure to lift an assembly instruction to BIL:

1. applying *arm_steps*, we compute the next steps for each instruction (i.e. $[(c_1, t_1) \dots (c_n, t_n)] = arm_steps(m, j, t, e, inst)$) and we prepare a symbolic state s_0 ¹
2. for each step, we apply the condition to the symbolic state to obtain a boolean predicate ($b_i = c_i(s_0)$). We transform the resulting predicates to BIL boolean expressions ($\llbracket b_i \rrbracket$) and emit the following fragment:

```

label GUARD_1
cjmp  $\llbracket b_1 \rrbracket$ , EFFECT_1, GUARD_2
...
label GUARD_N
cjmp  $\llbracket b_n \rrbracket$ , EFFECT_n, ERROR
label ERROR
assert false

```

The code of each label “guard- i ” checks if the corresponding step can be applied to compute the next state. The last label always fires a failure, informing us if an application can reach an unpredictable state

3. for each step, we apply the transformation to obtain a new symbolic state ($s_i = f_i(s_0)$), then for each field of the resulting state (i.e. memory locations, registers, flags and coprocessor registers) that has been updated (e.g. $s_i.regs.r1 \neq s_0.regs.r1$) we transform the resulting symbolic expression and assign it to the corresponding BIL variable, e.g.

¹Notice that the hypervisor design allows us to know at compile time if the exception handlers use Thumb mode, Jazelle mode and little-endianness

exp	$::= cst \mid exp \diamond_b exp \mid \diamond_u exp \mid cast(cast_kind, \tau_{reg}, exp) \mid var \mid lab(string)$ $\mid load8(var_mem, exp) \mid load16(var_mem, exp) \mid load32(var_mem, exp)$ $\mid store8(var_mem, exp, exp) \mid store16(var_mem, exp, exp) \mid store32(var_mem, exp, exp)$
var_{reg}	$::= (string, : \tau_{reg})$
var_{mem}	$::= (string, : u32 \rightarrow u8)$
var	$::= var_{reg} \mid var_{mem}$
$\diamond_b$	$::= +_w, -_w, *_w, /_w, /s_w, \%_w, \%s_w, \ll_w, \gg_w, \ggg_w, \&_w, or_w,$ $xor_w, ==_w, <_w, <s_w, \leq_w, \leq s_w, \&b, or_b, ==_b, <_b$
$\diamond_u$	$::= \sim_w, \sim_b$
cst	$::= n(: \tau_{reg})$
$cast_kind$	$::= unsigned \mid signed \mid high \mid low$
τ_{reg}	$::= bool \mid u2 \mid u3 \mid u4 \mid u5 \mid u8 \mid u16 \mid u32 \mid u64$

Figure 3: Syntax of BIL expressions

```

label EFFECT_i
R1 =  $\llbracket s_i.regs.r_1 \rrbracket$ 
MEM = store32(MEM,  $\llbracket address \rrbracket$ ,  $\llbracket exp \rrbracket$ )
    for instructions dealing with memory
jmp  $\llbracket s_i.regs.pc \rrbracket$ 

```

The described lifting procedure is straightforward. However, its soundness depends on the correct transformation of HOL4 terms (e.g. $\llbracket b_n \rrbracket$ and $\llbracket s_i.regs.r_1 \rrbracket$) to BIL expressions. Since the number of HOL4 operators that occur in the generated expressions is huge, we can not rely on a simple syntactical transformation to obtain a robust conversion of them to BIL. Moreover, the transformation of HOL4 terms to BIL expressions is used to convert the pre-post conditions of our contracts from HOL4 to BAP. The following section handles these challenges, presenting a procedure that lifts expressions and “certifies” that its output adheres with the input expressions.

5.2 Lifting expressions

The module to transform HOL4 terms to BIL must be able to handle the pre/post condition of the hypervisor contracts and the terms produced by the symbolic evaluation. These terms are based on basic mathematical (logical or arithmetic) and load (from memory) operations that do not use recursion. The converter is constructed by a formal model of BIL expressions in HOL4, theorems that establish the equivalence between atomic HOL4 expressions and composed BIL expressions, a procedure that converts the HOL4 terms into BIL counterparts and a serialization mechanism.

The semantic of the BIL operators is defined by mapping them to the corresponding operators of the HOL4 word theory. Since some HOL4 operators are not available in BIL, theorems have been proved to guarantee that the missing operators can be emulated by combining native BIL operators. The most time consuming part of this task is the verification of the theorems required to handle HOL4 operators based on finite Cartesian products. In fact, in the word theory “words” (or “bit vectors”) are represented as “finite Cartesian product”. Therefore, a bit vector has type $bool[\alpha]$, where α indicates the number of bits. Using this representation, several HOL4 operators are defined by iterating the bits of the words. However, there is no possibility of defining BIL expressions that use iterative routines.

To guarantee the soundness of the expressions conversion, the translation procedure *liftExp* “certifies” its output:

$$liftExp(exp) = (exp', \vdash exp = exp')$$

Here, exp is the HOL4 input term and exp' is the HOL4 output term that uses only BIL operators.

In order to dynamically generate the certifying theorem, the translation procedure is implemented in ML, which is the HOL4 meta language. The translation syntactically analyzes and deconstructs the input expressions to select the theorems to use in the HOL4 conversion and rewrite rules. For terms composed by nested expressions the procedure acts recursively. As explained in the following section, an important feature of this procedure is the automatic instantiation of theorems containing polymorphic types by syntactically analyzing the input term.

The last step to convert HOL4-terms is the serialization of the generated BIL expressions, which is accomplished by a set of custom-defined pretty printers.

5.2.1 Handling numeric expressions

All HOL4-terms generated by the symbolic execution have either type boolean or word. However, some expressions (e.g. the ones that represent shift and overflow calculations) have sub-expressions of type number. Transforming these expressions to BIL is challenging, since BIL provides only boolean and word operators having finite precision. This requires the verification of theorems that allow us to rule out all numerical sub-expressions. Clearly, the assumptions of these theorems define a bound over the value of operands and the result of the operations, thus guaranteeing the correct emulation of the numeric arithmetic using finite precision. Constituents of these assumptions are of polymorphic types (e.g. see the following *num2word_lt_thm* theorem). The converter procedure automatically verifies that the involved expressions meet the theorem assumptions, thus enabling their usage in the simplification rules.

We exemplify this approach by demonstrating the computation of the carry flag for the instruction `adds r1, r2`. The carry flag is computed by the HOL4 ARM model as follows:

```

 $\sim_b(w2n\ R1 : bool[32] + w2n\ R2 : bool[32]) < 0x100000000$ 

```

$R1$ and $R2$ are the variables that represent the registers `r1` and `r2`. Moreover, $w2n$ is the function that converts a word to the equivalent number. The resulting HOL4-term consists of components of types words (e.g. $R1$), numeral (e.g. $w2n\ R1 + w2n\ R2$) and boolean (e.g. $w2n(R1 : bool[32]) + w2n(R2 : bool[32]) < 0x100000000$). Since two variables $R1$ and $R2$ are words of size 32-bits, then application of $w2n$ returns numbers smaller than $2^{32} = 0x100000000$. To transform the term, the converter module uses the theorem *num2word_lt_thm*:

```

num2word_lt_thm =
 $\vdash \forall (arg1 : num)(arg2 : num).$ 
 $arg2 < dimword(: \alpha) \ \&_b\ arg1 < dimword(: \alpha)$ 

```

$$\begin{aligned} &\Rightarrow (arg1 < arg2) \\ &= ((n2w(arg1) : bool[\alpha]) <_{sw} (n2w(arg2) : bool[\alpha])) \end{aligned}$$

Here, the function *dimword* returns the maximum number of words having α -bits, and *n2w* is the function that converts numbers to words. The procedure instantiates the two operands (*arg1* and *arg2*) with $w2n(R1 : bool[32]) + w2n(R2 : bool[32])$ and $0x100000000$, respectively. Moreover, the procedure automatically selects 64 to instantiate the type parameter α , thus satisfying side conditions of the theorem. The result of this step is the following expression:

$$\sim_b (n2w(w2n R1 : bool[32] + w2n R2 : bool[32]) : bool[64]) <_{sw} 0x100000000w : bool[64])$$

Then, the procedure uses the theorem *word_add_n2w*, which is part of the HOL4 word theory, to convert the *addition* over numerals to its equivalent counterpart over words:

$$\begin{aligned} &\sim_b ((n2w(w2n R1 : bool[32]) : bool[64]) \\ &\quad +_w (n2w(w2n R2 : bool[32]) : bool[64]) \\ &\quad <_{sw} (0x100000000w : bool[64])) \end{aligned}$$

Finally, the last expression is simplified into following BIL expression:

$$\sim_b (cast(unsigned, u64, R1) +_w cast(unsigned, u64, R2)) <_{sw} 0x100000000w : u64)$$

6. SUPPORTING TOOLS

Effective application of the verification strategy required the implementation of several tools and optimization of the weakest precondition algorithm of BAP.

Weakest preconditions can grow exponentially with regard to the number of instructions. Even though this problem can not be solved in general, we can handle the most common case for ARM binaries, namely the sequential composition of several conditionally executed arithmetical instructions. This pattern matches the optimization performed by the compiler to avoid small branches. For example, naively computing the weakest precondition of $R1 > 10$ for the assembly fragment `addeq R1 #7; mueq R1 #2` produces the predicate

$$\begin{aligned} Z \Rightarrow (Z \Rightarrow ((R1 + 7) * 2 > 10) \wedge \neg Z \Rightarrow ((R1 + 7) > 10)) \\ \wedge \\ \neg Z \Rightarrow (Z \Rightarrow (R1 * 2 > 10) \wedge \neg Z \Rightarrow (R1 > 10)) \end{aligned}$$

that is equivalent to $(Z \Rightarrow ((R1 + 7) * 2 > 10)) \wedge (\neg Z \Rightarrow (R1 > 10))$. We improved the BAP weakest precondition algorithm by adding a simplification function that identifies these cases. The application of this straightforward strategy reduced the size of the weakest precondition for a fragment of the kernel bootstrap code (containing one “switch” and a subsequent “if then else”, consisting of 27 C lines compiled to 35 machine instructions) from 8 GB to 15 MB.

Machine code (and BIL) lacks information on data types (except for the native types like word and bytes) and represents the whole memory as a single array of bytes. Writing predicates and invariants is complex because their definition depends on location, alignment and size of data-structure fields. Moreover, the behavior of compiled code often depends on the content of static memory used to represent constant values of the high level language. We developed a set of

tools that integrate HOL4 and GDB to extract information from the C source code and the compiled assembly. With the support of these tools we are able to write the invariants and contracts of the hypervisor independently by the actual symbol locations and data structure offsets produced by the compiler. For example, the following invariant constraints the data structure managing the virtual machines to be a circular linked list:

```
(load32 (memory, global_vm_0_add+
          o_virtual_machine__next)) =
      global_vm_1_add /\
(load32 (memory, global_vm_1_add+
          o_virtual_machine__next)) =
      global_vm_0_add
```

Before the execution of the proof scripts and the generation of the BAP pre/post-conditions, all constants are substituted with the corresponding actual values. The circular linked list invariant is then transformed into

```
(load32(memory, 0x81D8) == 0x8080 /\
(load32(memory, 0x8124) == 0x8134
```

Verifying a Hoare triple by computing the weakest precondition depends on the absence of indirect jumps. Even if the hypervisor code reduces their explicit usage (e.g. by avoiding function pointers), the compiler introduces an indirect jump for each function exit point (e.g. the instruction at the address `0x20C` in Figure 4, is an indirect jump). Solving an indirect jump is not independent of checking the correctness of other properties of the application (e.g. the code in Figure 4 requires that the register *R2* does not point to the same memory area referenced by the stack pointer). Since we are interested in solving indirect jumps of code fragments that must respect contracts (Hoare triples $\{P\}C\{Q\}$), we implemented a simple iterative procedure that uses STP to discover all the possible assignments to the indirect jumps under the contract precondition *P*.

1. We compute (using BAP) the control flow graph of the *C* fragment. From the control flow graph we extract the list of reachable labels containing an indirect jump (in the example `pc_0x20C`)
2. For each label we generate a code fragment *C'* as follows:
 - (a) we transform the selected instruction `jmp exp` into `assert(exp <> freevar);halt`, where `freevar` is a free variable in the fragment *C*
 - (b) we suppress all other indirect jumps, transforming `jmp exp` into `halt`
 - (c) we compute the weakest precondition $WP(C', true)$
 - (d) using the SMT solver, we search for an assignment $\{value/freevar\}$ that is a counterexample of the formula $P \Rightarrow WP(C', true)$
3. If for all indirect jumps the SMT solver does not find a counterexample, then all indirect jumps are unreachable instructions and all possible targets have been discovered. We transform *C* into *C''* by changing all indirect jump into `assert false`
4. If for the indirect jump at the label `1b1` the SMT solver discovers the counterexample having the assignment $value/exp$, we transform *C* into *C''* by changing `jmp exp` into

```

0x100 bl #0x200    label pc_0x100 PC = 0x100;
                    LR = PC+4; jmp pc_0x100
0x104 ...          label pc_0x104 ...
0x108 bl #0x200    label pc_0x108 PC = 0x108;
                    LR = PC+4; jmp pc_0x100
0x10C b R1         label pc_0x10C PC = 0x10C;
                    jmp R1
...
// function
0x200 push LR      label pc_0x200 PC = 0x200;
                    mem=store32(mem, SP, LR);
                    SP=SP+4
0x204 str R1, R2   label pc_0x204 PC = 0x204;
                    mem=store32(mem, R1, R2);
0x208 pop LR       label pc_0x208 PC = 0x208;
                    LR=load32(mem, SP+4);
                    SP=SP+4
0x20C b LR         label pc_0x20C PC = 0x20C;
...               jmp LR

```

(a) Assembly

(b) BIL

Figure 4: Indirect jump example

```

cjmp exp==value; value; new_label
label new_label: jmp exp

```

and the procedure is restarted.

7. EXAMPLE: THE “SEND” HYPERCALL

This section presents the verification of the message delivery service of the hypervisor. The service-request convention of the hypervisor allows a guest to request a service by invoking a hypercall using the software interrupt instruction. The requested hypercall is uniquely identified by the number X embedded into the **SWI X** instruction. Moreover, a hypercall can be invoked only in standard ARM mode; any request raised from states in Thumb or Jazelle mode has to be handled as NoOperation. To send a message to the other guest, the sender stores the message into its register $R1$ and executes **SWI 1**. Then the hypervisor stores the message into the message box of the receiver and resumes immediately the sender guest.

When a software interrupt is raised, the CPU switches to *SVC* mode and the program counter is changed to the vector table entry for the software interrupt. The hypervisor jumps to the address of the assembly function `impl_swi`, which stores the user context (i.e. the registers and flags accessible in user mode) into the hypervisor stack. The assembly function reads from the guest memory the last executed instruction and decodes the number of the invoked hypercall. The function terminates by calling the C function referenced by the pointer `family_callback_swi`, which is configured by the bootstrap to point to `swi_handler`. The C function dispatches the request to the proper hypercall and terminates by returning to the assembly fragment `exception_bottom`. This low level fragment restores the user context (including the program counter) from the stack and switches the CPU to user mode.

We comment on a part of the contract that constraints the behavior of the hypervisor code when the first guest requests to send a message. The contract is automatically translated from HOL4 to BAP using the lifter presented in Section 5. Moreover, the value of the constants have been generated by the tools that interact with GDB to discover variable linking and data structure field offsets. The contract precondition consists of 210 statements, including:

- the hypercall has been invoked by first guest (i.e. the first guest is active):
 $\text{load32}(\text{mem}, 0x8130) == 0x8134$
- the CPU registers and flags represent the registers and flags of the first ideal machine, for example:
 $R1 == M1_R1$ and $N == M1_N$
- since the second guest is suspended, the hypervisor memory contains the registers and flags of the second ideal machine, for example:
 $\text{load32}(\text{mem}, \text{load32}(\text{mem}, 0x811C) + 0x4) == M2_R1$
and
 $\text{load32}(\text{mem}, \text{load32}(\text{mem}, 0x811C) + 0x40) \& 0x1 == M2_N$

- the hypervisor memory contains the message boxes of the two ideal machines, for example:

```

load32(mem, 0x81DC) == M1_MSG
and
load32(mem, 0x8128) == M2_MSG

```

- several hypervisor invariants hold: e.g. the data structure holding the information about the two guests is a circular linked list

- the hypervisor memory contains the page table

- if the guest invoked the hypercall in standard ARM mode, then the last executed instruction is **SWI 1**:

```

(M1_J ∨ M1_T) ∨ (load32(mem, LR_SVC 0x4)
==
0xEF000001)

```

The corresponding postcondition consists of 177 statements:

- the user mode is restored: $M == 0x10$
- the active guest is not changed:
 $\text{load32}(\text{mem}, 0x8130) == 0x8134$
- the registers and flags of the two guests are unchanged (namely, the hypercall does not affect the content of the virtual registers or correctly restores their content) e.g.
 $R1 == M1_R1$
and
 $\text{load32}(\text{mem}, \text{load32}(\text{mem}, 0x811C) + 0x4) == M2_R1$
- all hypervisor invariants still hold: e.g.
 $\text{load32}(\text{mem}, 0x8124) == 0x8134$
and
 $\text{load32}(\text{mem}, 0x81D8) == 0x8080$
- the allocated memory for the two guests is not affected by the hypervisor:

```

(
  (free_add1 < 0x100000
   or
    free_add1 >= 0x400000)
  or
  (load8(mem, free_add1)
   ==
    load8(mem_old, free_add1))
)
and

```



```
(
  (free_add2 < 0x400000
   or
    free_add2 >= 0x700000)
 or
  (load8(mem, free_add2)
   ==
    load8(mem_old, free_add2))
)
```

- if the guest invoked the hypercall in standard ARM mode then the message (content of the virtual register *R1* of the first guest) is copied into the message box of the other guest, otherwise the message boxes are not changed

```
((M1_J or M1_T) or
 (load32(mem, 0x81DC) = M1MSG
 and
  load32(mem, 0x8128) = M1_R1)
)
and
((~M1_J and ~M1_T) or
 (load32(mem, 0x81DC) = M1MSG
 and
  load32(mem, 0x8128) = M2MSG)
)
```

Since the control flow is acyclic, we can directly verify the contract by computing the weakest precondition. However, the code of the software interrupt contains four indirect jumps. We use the strategy presented in Section 6 to solve the indirect jumps. The process terminates by discovering:

- The instruction at the address 0x204 (i.e. the last instruction of `impl_swi`) always jumps to the address 0x6E4 (i.e. the address of `swi_handler`). In fact, the target address depends on the function pointer `family_callback_swi` and its value is constrained by the hypervisor invariants.
- The precondition ensures that the “send” hypercall has been invoked (i.e. `load32(mem, LR_SVC - 0x4) == 0xEF000001`), causing the indirect jumps at the addresses 0x24C and 0x6F0 to be unreachable. In fact, these two addresses are the exit points (e.g. `bx lr`) of the other hypercalls.
- The instruction at the address 0x804 always jumps to 0x6E4 (i.e. the address of `exception_bottom`). This instruction is the exit point of the last function invoked by the C implementation of the “send” hypercall.

We compute the weakest precondition and then we use STP to verify that the resulting expression is implied by the precondition. The verification procedure discovered two bugs. The first bug was caused by the `exception_bottom` procedure. It uses `msr spsr, r0` instruction to restore the guest flags, however this instruction does not override the control and status fields of the control register. We fixed the error by replacing the instruction with `msr spsr_fsrc, r0`. The second problem arises if the guest exploits thumb mode. The software interrupt handler must first load the SWI instruction that caused the exception into a register. Then, the handler examines the instruction bits to determine the required hypercall. The original hypervisor code always loaded four bytes from the guest memory to read the guest instruction. However, in thumb mode, the instruction is encoded using two bytes: accessing two additional bytes can

cause accesses to a part of the memory that is not allocated to the running guest. A possible solution is to use the `LDRH` instruction to load a half word if the processor was in thumb mode before activating the hypervisor handler. On the other hand, the TLS specification requires that all hypercalls invoked in thumb mode are handled as no operations; in this case there is no reason to access the guest memory to decode the SWI instruction and the hypervisor has been fixed by enforcing this policy.

8. EVALUATION

The main development task has been the implementation of the new BAP front-end for ARMv7, which consist of 4k lines of HOL4 code. The verified binary kernel consists of 3k ARM instructions, which are transformed into 7k lines of BIL in about an hour. The tools developed to integrate GDB and solve indirect jumps required 2k lines of HOL4 and python code.

The kernel binary code is verified with respect to sixteen contracts, each of them consisting of ~ 400 lines of assertions. In the worst case, the verification of one contract required ~ 30 minutes using one Intel(R) Xeon(R) X3470 core; the contract is transformed from HOL4 to BIL in ~ 5 minutes, the indirect jumps are solved in ~ 2 minutes, the weakest precondition is computed in ~ 10 minutes and the SMT solver verifies the validity of the resulting condition in ~ 15 minutes.

The assembly verification required four person months, mainly used to formalize the invariants of the kernel data structures and to check the kernel boot code, which contains several loops. Several properties of the hypervisor code allowed the usage of weakest preconditions to be effective. In particular, the loops have always a structured control flow, with a conditional instruction that represents both the enter and exit point of the loop.

9. RELATED WORK

Verifying low-level software, like operating systems and hypervisors, at the machine code level has the benefit of providing results that are not dependent on the correctness of the compilers. Moreover, low-level software needs to mix code developed in high level languages (e.g. C) with assembly fragments to perform special machine instructions (e.g. to access registers available in privileged modes, to change execution mode, to affect the system behavior by interacting with the coprocessor). This makes it difficult to adapt and use verification tools that target user-land programs (e.g. VCC [6]). Shadrin et al [1, 18] addresses this challenge by providing a joined semantics of C and assembler. They map the complete machine state using C global variables and emulate the assembler code in C, thus enabling the adoption of VCC as verification tool.

The verification of binary code requires a detailed model of the machine behavior. We adopted the HOL4 model for ARMv7 provided by Fox and Myreen [8]. On top of this model, the authors extended the Hoare logic to deal with binary code [15], providing inference rules to reason about machine code. The same model has been used to provide a semi-automatic verification approach [16, 17]. Here, the authors implemented a “decompilation” procedure which produces an HOL4 function that behaves equivalently (i.e. implements the same state transformation) to the decompiled

assembly code. This result allows to lift the verification of properties of assembly programs to reasoning on HOL4 functions.

David S. Hardin et al. [10] implemented an automatic verification tool for the AAMP7G architecture using ACL2. The authors use the approach of “compositional cutpoints” to handle loops. The tool uses symbolic evaluation and the automatic machinery of ACL2 to generate a “cutpoint to cutpoint” proof, establishing that if the precondition holds then the loop-free fragment establishes the specified post-state.

Several verification techniques (e.g. symbolic execution and computation of weakest preconditions) for binary code verification require to solve indirect jumps. Finding the target addresses for the indirect jumps is known to be difficult using static analysis, since it is not independent of checking the correctness of other properties of the application like the stack management and the absence of buffer overflows. Approaches to resolve the symbolic indirect jumps fall into two main categories. The first methodology (e.g. [3]) reasons about the entire program using source and binary analysis tools. The tools must be able to explore the program structure to find all possible assignments to function pointers and all possible callers of a function. Then the code is annotated with the assertions that allow to identify misbehaviors (e.g. a jump to a function from a location not in the identified set of callers). A different approach is to use SMT solvers to find all satisfying answers to first order formulas that represent path predicates and symbolic jumps (e.g. [20]).

Several works address the verification of hypervisor and microkernels. Among them, the seL4 project [13] verified the functional correctness of the C implementation using the Isabelle theorem prover. Recently [21], translation validation has been applied to guarantee that the kernel binary produced by the compiler conforms to the C source code.

10. CONCLUSIONS

We have presented the verification of the ARMv7 assembly of a hypervisor. Automation of the verification process is done using the BAP utilities, to compute weakest preconditions, and STP solvers. To achieve this goal, we developed a new BAP front-end for ARMv7 machine code and a procedure to solve indirect jumps. The development of the new front-end on top of the HOL4 ARM model allowed us to mix HOL4 proofs with external tools, without relying on different semantics of ARMv7 instructions. Moreover, the resulting tool chain can be used in other projects that require verification of ARM machine code.

To adopt our tools to verify the binary code of other hypervisors and low-level softwares three main constraints must be satisfied: (i) the MMU setup allows that binary code to run under identity virtual-to-physical address translation (ii) the kernel does not allow preemption, that is kernel handlers can not be interrupted by nested exceptions or interrupts (iii) only one physical core is used. We are currently working to extend the HOL4 ARM model and our BAP front-end to relax the first constraint and enable the verification of kernels that executes under a fixed (but not identity) address translation.

The verification of the entire software at machine code level avoids trusting a legacy C compiler. Moreover, we can transparently handle software that mixes high level languages and assembly. The binary code can be generated

from different sources or can contain binary blobs for which sources are unavailable. Our verification task has been simplified by the hypervisor design, which uses a structured control flow, where each loop has a instruction representing both its entry and exit point (i.e. no `break` instruction is used in the C code). Verifying software having unstructured control flow requires to extend the algorithm to compute the weakest preconditions as proposed in [4].

Checking functional correctness of the hypervisor handlers (by verifying if the machine code respects the contracts) is not sufficient to guarantee that the hypervisor isolates the virtualized guests. For this reason, the verification of the machine code is part of a wider verification effort. In [7] we use HOL4 to compose the main “building-block” of our verification strategy and guaranteeing the main security property of our system.

11. ACKNOWLEDGMENTS

Work supported by framework grant “IT 2010” from the Swedish Foundation for Strategic Research.

12. REFERENCES

- [1] Shadrin Andrey. *Mixed Low- and High Level Programming Languages Semantics. Automated Verification of a Small Hypervisor: Putting It All Together*. PhD thesis, Saarland University, Saarbrücken, 2012.
- [2] ARMv7-A architecture reference manual. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c>.
- [3] Gogul Balakrishnan and Thomas Reps. Wysinyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32:23:1–23:84, August 2010.
- [4] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE ’05, pages 82–87, New York, NY, USA, 2005. ACM.
- [5] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *CAV*, pages 463–469, 2011.
- [6] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying Concurrent C. In *Proc. TPHOLs’09*, pages 23–42, 2009.
- [7] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemat, and Oliver Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proc. CCS’13*. ACM, 2013. To appear.
- [8] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proc. ITP’10*, pages 243–258, 2010.
- [9] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV’07*, pages 519–531, 2007.
- [10] David S. Hardin, Eric W. Smith, and William D. Young. A robust machine code proof framework for

- highly secure applications. In *Proc. ACL2'06*, pages 11–20, 2006.
- [11] HOL4. <http://hol.sourceforge.net/>.
 - [12] Narges Khakpour, Oliver Schwarz, and Mads Dam. Machine assisted proof of ARMv7 instruction level isolation properties. In *CPP'13*, 2013. To appear.
 - [13] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proc. SOSP'09*, pages 207–220, 2009.
 - [14] Magnus O. Myreen, Anthony C. J. Fox, and Michael J. C. Gordon. Hoare logic for arm machine code. In *Proceedings of the 2007 international conference on Fundamentals of software engineering*, Proc. FSEN'07, pages 272–286, Berlin, Heidelberg, 2007. Springer-Verlag.
 - [15] Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In *Proc. TACAS'07*, pages 568–582, 2007.
 - [16] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In *Proc. FMCAD'08*, pages 1–8, 2008.
 - [17] Magnus O Myreen, Michael JC Gordon, and Konrad Slind. Decompilation into logic-improved. In *Proc. FMCAD'12*, pages 78–81, 2012.
 - [18] Wolfgang J. Paul, Sabine Schmaltz, and Andrey Shadrin. Completing the automated verification of a small hypervisor - assembler code verification. In *Proc. SEFM'12*, volume 7504 of *Lecture Notes in Computer Science*, pages 188–202. Springer Berlin Heidelberg, 2012.
 - [19] PROSPER Project. <http://prosper.sics.se/>.
 - [20] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, pages 317–331, 2010.
 - [21] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified os kernel. In *Proc. PLDI'13*, pages 471–482, 2013.