Renee 1.0

Scalable Translation Validation of Unverified Legacy OS Code

Amer Tahat, Sarang Joshi, Pronoy Gawsamy, Binoy Ravindran
Presenter: Amer Tahat
System Software Research Group-SSRG
Department of Electrical and Computer Engineering
Virginia Tech University

Acknowledgments: This work is supported in part by ONR (Office of Naval Research) under grant number N00014-18-1-2665. We are also very grateful to Dr. Natarajan Shankar and Sam Owr from SRI for providing us with pvs7-dev and its patches that helped us to use PVS7 in our work.
Question

Is there any feasible methodology to produce a trustworthy *formal* model of a large OS? What about multiple OSes?
1. They may not have the source code available (only the binary).
2. They may not have the formal semantic of the high source code - possibly written in multiple languages (if the source is available).
3. Gap between formal model and the code. Expensive?
4. Large number of LOC, developers, and a complex life cycle.
5. Smaller number of formal verification engineers.
Related Work

1. **SeL4**, assumes that complete high-level source code of the OS is available to the verifier in a subset of the C language, called C0[1,2]

2. **CompCert**, presents the formal proof for a compiler, but restricts it to a subset of C called C-light[8].

3. **TAL**, presents a verification toolchain that targets a typed assembly language, which is transformed into a typed machine language to generate a safe binary.

4. **Hyperkernel***, an approach for designing a new OS kernel from scratch that is verifiable using SMT solvers, but the approach scopes out verifying legacy operating system [9].

5. [10] establishes that seL4’s binary code is equivalent to its C 0 source, but is restricted to the already verified seL4’s C0 code

6. **ARM in HOL** (2006-2010) [12,14], ARM in HOL [ 2011 - 2016] [13,15].

Hyperkernel* Best paper award in On Symposium on Operating Systems, Principles (SOSP17).
Related Work Cont’d

**ASL**: ARM Specification Language 2016 (*Trustworthy and Machine Readable*).


| Applications | Translation into many theorem provers, smt solvers other external specification languages ASL into SAIL [then into multiple theorem provers] [spisa19]. |

[https://alastairreid.github.io/specification_languages/](https://alastairreid.github.io/specification_languages/) *(More about ASL)*
Renee toolchain for the formalization of arm binary code

Fig. 1: Toolchain Workflow. $r2 =$ radare2, $tr =$ Translation, $D =$ Data transfer, $Ex =$ Theories Export, $VA =$ Validation.
ASL for Renee

Assisted us in many ways:

- Translating the instructions into PVS7,
- Generating Tests to validate ASl2PVS7 tr,
- Building a decoder, and an encoder from/to the theorem prover and radare2.
PVS7-Dev a game Changer
PVS7-Dev Background

- Theory parameters; e.g.;
  - bv: Theory [n : Nat] , n is visible in theory

- Dependant types
  - bvec[n] : Type = [ below(n) -> bit]

- Generic Theories
  - ( OOF- Object Oriented Formalization)
PVS7-Dev Theory Declaration

Ex: Let $A$ be an abstract PVS theory with two bit vectors attributes; called $a_1$ and $b_1$.
We can declare:

$B : \text{Theory} = A \text{ with } \{ \{ a_1 := \text{bv}[2](0b01) \} \}$

$C : \text{Theory} = A \text{ with } \{ \{ a_1 := \text{bv}[3](0b101), b_1:= \text{bv}[2](0b10) \} \}$
Renee’s Core Formalization Idea

- Every byte code in the target can be represented -in PVS7- as an instance of an abstract instruction’s Theory (translated from ASL-XML file)!
From ASL to PVS7
RSL: PVS7 Instructions Theories

Ands_log_shift: Theory [ (importing armstate) p : arm-state ]

BEGIN

Diag : bv[64] // will be instantiated by Translator with a bit vector

Decoding part

Listing 1: ASL-XML ands-log-shift bits diagram

Addr : bv[64]

Listing 2: PVS ands-log-shift diagram load
~ 1-1 Formalization ASL into PVS7

Listing 4: PVS ands-log-shift Operational

```
sts3: ASL(p)= sts2 with [operand1:= p.X(n)]
sts4: ASL(p)= sts3 with [operand2:= ShiftReg(64, p.X(m), shift_type, shift_amount)]
sts5: ASL(p)= if invert then sts4 with [operand2:= NOT(sts4.operand2)]
    else sts4 endif
sts6: ASL(p) = Cond sts4.op = LogicalOp_AND -> sts5
    with [result := AND (sts5.operand1, sts5.operand2)],
    sts4.op = LogicalOp_ORR ->
    sts5 with [result := OR (sts5.operand1, sts5.operand2)],
    sts4.op = LogicalOp_EOR ->
    sts5 with [result := XOR (sts5.operand1, sts5.operand2)] EndCond
% post state
pl:s = if sts6.setflags then p with [.PSTATE.NZCV:=
    let result_63 = field(64, sts6.result, 63,63)
    in bv[2](0b00) o IsZeroBit(64,sts6.result) o
    result_63]
    else p endif
post: s = p1 with [.X(d) := sts6.result]
```

Listing 5: ASL ands-log-shift Operational

```
bits(datsize) operand1 = X[n];
bits(datsize) operand2 = ShiftReg(m,
    shift_type, shift_amount);
if invert then
    operand2 = NOT(operand2);
case op of when LogicalOp_AND
    result = operand1 AND operand2;
when LogicalOp_ORR
    result = operand1 OR operand2;
when LogicalOp_EOR
    result = operand1 EOR operand2;
%Post state
if setflags then PSTATE.<N,Z,C,V> =
    result<datasize-1>:IsZeroBit(result):'00';
X[d] = result;
```
From radare2 to PVS7
Translation Process

Bin

Extracts Data from JSON

Extracted Decoder

Pattern matching

ASL XML

Decoding Dictionaries

Python

Radare2PVS

Translator

RSL semi-auto ASL2PVS7

Code into PVS7 files

ASL to PVS7 translation into abstract theories

Validation Tools: UniV7 and Reverse Dictionaries

Radare2PVS

Loads info into PVS abstract theories
Theories to reproduce the formal code
Radare2PVS7: Basic Block Tr

New Object of subs_addsub_imm

Original binary code-basic block stripped using radare2 analysis agf
Radare2PVS7: Basic Blocks CFG Tr

CFG: Control flow graph

PVS working directory/zircon/terminals
Functions Translation (CFG)

(Main file for each functions)

Auto Proofs - TCCs

E.g; Main_acrh_mp_send_ipi.pvs
Filling the Gap:
1- Unicorn 2 PVS7
UniVS7: Unicorn to PVS7 Validation Tool

Import Abstract model

Map pre-state unicorn state

Instantiate PVS7 model with the byte code!

Check the value emulated in PVS vs unicorn’s

Validate it!

Listing 6: ands_log_shift UniVS7 generic test format
Filling the Gap:
2- Reverse Dictionaries
Radare2PVS Validation via Reverse Dictionaries

We encode PVS instructions back to ARM binary using a reversed algorithm of the decoder and compare the outputs with radare’s code.
Renee on Google’s Zircon & Linux
Simple demo

Click here: Renee_v1_tr_from_r2pv7
### Statistics & Results

![Instruction Class Usage](image)

**Fig. 2: Instructions classes usage in Zircon and Linux.**

#### Table 1: Toolchain statistics on Zircon microkernel and Linux Kernel.

<table>
<thead>
<tr>
<th>Properties</th>
<th>Zircon</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Languages Used</td>
<td>C++, C, asm, Py, Shell</td>
<td>C,asm</td>
</tr>
<tr>
<td>Size of High Source</td>
<td>71K</td>
<td>100K</td>
</tr>
<tr>
<td># Byte Code Analyzed</td>
<td>200K</td>
<td>300K</td>
</tr>
<tr>
<td># Functions Formalized</td>
<td>127</td>
<td>243</td>
</tr>
<tr>
<td>Target CFG&amp;FCG</td>
<td>Tree,Term</td>
<td>Tree,Term</td>
</tr>
<tr>
<td>Target Byte Code</td>
<td>383</td>
<td>665</td>
</tr>
<tr>
<td># PVS7&amp;Proof-Lite LOC</td>
<td>2253</td>
<td>6950</td>
</tr>
<tr>
<td>Overall Tests &amp; TCCs</td>
<td>150K</td>
<td>150K</td>
</tr>
<tr>
<td>Reverse Dic Tests &amp; TCCs</td>
<td>3056</td>
<td>5320</td>
</tr>
<tr>
<td>Translator TCB LOC</td>
<td>1430</td>
<td>1430</td>
</tr>
<tr>
<td>Time for Tr</td>
<td>10m</td>
<td>15m</td>
</tr>
<tr>
<td>Time for Tr VA</td>
<td>30m</td>
<td>45m</td>
</tr>
</tbody>
</table>
Limitations

1. We formalized a subset of ARMv8.v3-A64 instructions (used in our targets’ selected functions).
2. We are also restricted to Linear-terminal functions (essential to formalizing almost all other functions).
3. We supported sequential deterministic code.
Work in progress

- Adding more A64 instructions classes (more coverage),
- Adding more 32bits-instructions (back compatibility),
- Functions with loops,
- Proving security properties: Adding formal assurance against (DOP, JOP, ROP attacks).
Questions?

The End!

THANK YOU!
REFERENCES


