A Proof Infrastructure for Binary Programs

Ashlie B. Hocking[1], Benjamin D. Rodes[1], John C. Knight[1],
Jack W. Davidson[2], and Clark L. Coleman[2]

[1] Dependable Computing, Charlottesville, VA USA
{ben.hocking,ben.rodes,john.knight}@dependablecomputing.com
[2] Zephyr Software LLC, Charlottesville, VA USA
{jwd,clc}@zephyr-software.com

Abstract Establishing properties of binary programs by proof is a desirable goal when the properties of interest are crucial, such as those that arise in safety- and security-critical applications. Practical development of proofs for binary programs requires a substantial infrastructure to disassemble the program, define the machine semantics, and actually undertake the required proofs. At the center of these infrastructure requirements is the need to document semantics in a formal language. In this paper we present a work-in-progress proof infrastructure for binary programs based on AdaCore's integrated development and verification environment, SPARKPro. We illustrate the infrastructure with proof of a security property.

1 Introduction

Establishing properties of binary programs by proof is a desirable goal receiving significant attention recently [2,4,5,6]. Any approach to proving software properties requires a comprehensive infrastructure that: (a) defines the semantics of the target machine architecture, (b) translates binary programs into a representation suitable for proof based on the defined machine architecture semantics, and (c) operates on translated binary representations to generate proof.

Many languages could be used to define machine semantics, and many proof tools exist. Our infrastructure is based on an application of AdaCore's integrated development and verification environment, SPARKPro [1] and our custom binary-to-SPARK-Ada translator. SPARKPro was chosen for many reasons:

- The SPARK Ada language [1] has been designed for proof and includes syntactic structures to enable definition of the necessary verification conditions.
- SPARK Ada is familiar to many in the community and simple to use.
- SPARKPro proof tools provide the capability to establish necessary proofs.
- SPARKPro has industrial-strength support thereby allowing the technology to be adopted by practitioners.
- SPARKPro provides an executable specification that can be tested.

In this paper, we present a work-in-progress binary proof infrastructure based on SPARKPro. We illustrate the infrastructure with an example binary program and prove the program possesses a desired security property.

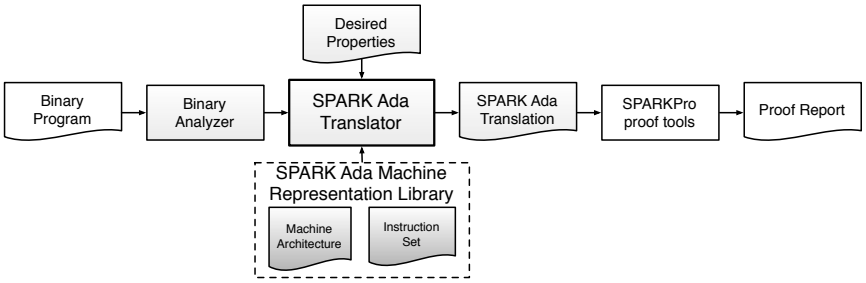

Figure 1. Architecture of proof infrastructure for binary programs. The SPARK Ada Machine Representation is the focus of this paper, and light gray elements indicate other supporting aspects of our work.

2 Proof Infrastructure

Figure 1 shows the architecture of our proof infrastructure. A binary program is first processed by a static analyzer to disassemble the program and recover important program structures. Of particular importance in the analysis is the recovery of function boundaries and control structures such as conditions and loops. A translator then converts the binary program to a SPARK Ada representation. The translator accesses semantics of the target machine architecture and instruction set, both defined within our SPARK Ada library. It also accesses a description of desired program properties to prove and merges them into the representation of the subject program. Finally, the composite representation of the subject program and desired properties is submitted to the SPARK prover.

The proof infrastructure could be applied to any instruction set architecture (ISA); however, our current research focuses on X86–64. Figure 2 shows a high-level organization of the two semantic definitions of the X86–64 ISA.

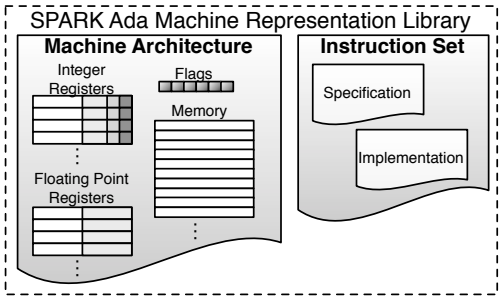

Figure 2. X86–64 Semantic Definition Library

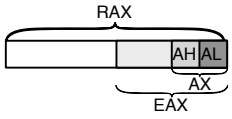

Figure 3. RAX register and alternate access methods

Central to the machine semantics are registers. The integer registers are represented as Ada integers with modulus 2^{64} (`Unsigned64`). As shown in Figure 3, a general-purpose X86–64 register (e.g., `RAX`) can be accessed multiple ways. `RAX` is modeled as `Unsigned64` and is directly accessed for reading and writing. `EAX` is modeled by read/write functions as shown in Figure 4. The read function

(EAX) returns the lower 32-bits of RAX and Write_EAX sets those bits, while setting the upper bits to zero. AL, and AH and AX are specified similarly, except with appropriate bits preserved instead of set to zero. Each function includes a post-condition in the SPARK Ada syntax describing the expected result. These post-conditions are verified by the SPARKPro proof tools. Flag registers (OF, SF, ZF, AF, CF, and PF) are modeled as Boolean. Floating-point registers (e.g., XMM and YMM) are not currently modeled.

```
133 function EAX return Unsigned32 with
134   Global => (Input => RAX),
135   Post => (EAX'Result = Unsigned32(RAX and 16#00000000FFFFFFFF#));
136 procedure Write_EAX(Val : in Unsigned32) with
137   Global => (In_Out => RAX),
138   Post => ((EAX = Val) and ((RAX and 16#FFFFFFFF00000000#) = (16#0000000000000000#)));
```

Figure 4. EAX specification

Memory is modeled as an array of 2^{64} 8-bit elements. The declaration of this array is shown in Figure 5, along with 16-bit reads and writes operating on the memory array.

```
12 type Mem_Array is array (Unsigned64) of Unsigned8;
13 Memory: Mem_Array := Mem_Array'(others => 0);
14 function ReadMem16(addr: in Unsigned64) return Unsigned16 with
15   Global => (Input => Memory),
16   Post => (((ReadMem16'Result and 16#00FF#) = Unsigned16(Memory(addr))) and
17           ((ReadMem16'Result and 16#FF00#) = Unsigned16(Memory(addr+1))*16#100#));
18 procedure WriteMem16(addr : in Unsigned64; Val : in Unsigned16) with
19   Global => (In_Out => Memory),
20   Post => ((ReadMem16(addr) = Val) and (for all i in Unsigned64 =>
21    (if ((i /= addr) and (i /= addr + 1)) then (Memory(i) = Memory'Old(i)))));
```

Figure 5. Memory type specification

Many X86–64 instructions are modeled as SPARK Ada functions operating on memory and registers. For example, the instruction setnbe is specified as shown in Figure 6. In some cases, instructions match an operator in Ada (e.g., addition), and for those instructions the Ada operator is used directly. Similarly, jump instructions are modeled using Ada control statements (e.g., loops). Other approaches to modeling jumps are possible, but difficult to prove. For example, a binary program could be modeled as an array of instructions and a location counter that is used as an array pointer. Jump instructions could then set the instruction counter accordingly. The lack of loop details, however, would make synthesis of loop invariants and subsequent proof almost impossible.

```
622 procedure setnbe_CL with
623   Global => (Input => (ZeroFlag, CarryFlag), In_Out => RCX),
624   Post => (if ((not CarryFlag) and (not ZeroFlag)) then (CL = 1) else (CL = 0));
```

Figure 6. Specification of setnbe

3 Example

To illustrate the proof infrastructure and to highlight areas of current work, we examine an example challenge function for security, zero_array, the C representation of which is shown in Figure 7. The zero_array function is passed a

pointer to an array and a size parameter. The function proceeds to zero out `size` elements of the array. This function presents a typical security challenge since `zero_array` might result in a buffer overflow that could corrupt, among other things, function return addresses depending on the value of the `size` parameter.

```
24 void zero_array(int *array, int size) {
25   for (int i = 0; i < size; i++) array[i] = 0;
26 }
```

Figure 7. Implementation of `zero_array`

In an example program (not illustrated) `zero_array` is called from two different functions, each of which passes a pointer to an array of a different size. In the program, the `size` parameter is always set to the size of the array, i.e., while `zero_array` is potentially dangerous, its use in this example does not introduce a security vulnerability. The example program was compiled with `gcc` and the raw disassembled binary as produced by `objdump` was examined.[3]

The SPARK Ada representation of the `zero_array` function is shown in Figure 8, with the associated disassembled code included as comments. Line 19 of Figure 8 represents the `mov` instruction as `Write_EAX`; however, for lines 9–11, instead of modeling the `test` instruction as a procedure, the result of `test` (i.e., assignment of flag registers) is represented explicitly in the translated code. Additionally, the binary analysis detects write-after-write situations affecting flags. For example, the flags that would be set by the `add` instruction (lines 23–24) are not read prior to the following `cmp` instruction, so there is no need to model the setting of these flags.

The loop on line 20 and the `if` statement on line 13 are examples of control structures recovered by the static analyzer from analysis of jump instructions.

To prove security properties about the SPARK Ada representation, constraints are added to the initial version of the representation (not illustrated). So as to prove the integrity of other items on the stack, the constraint in this example is that the loop index of `zero_array` will not exceed the `size` parameter. With this constraint in the example, using the SPARKPro prover (`gnatprove`) with the `cvc4` backend we are able to prove that the example program will not overwrite any function's return address.

This proof requires approximately 8 seconds to complete when using all 8 cores of a MacBook Pro (Retina, Mid 2012). We plan to publish further discussion of automatic constraint development in the future.

4 Related Work

Zhao et al. [6] propose binary software fault isolation techniques (ARMor) based on a model of the ARM ISA [3] and Hoare logic. Their approach modifies a binary program by inserting guards at possibly dangerous instructions. Proofs are then generated about security of the modified code. XFI is an approach similar to Zhao et al. developed to support binary programs on Windows [2]. XFI's verification is based primarily on the defined properties of security guards. Software

[3] The binary analyzer uses a combination of objdump and IDA Pro.

```
6  procedure zero_array is
7  begin
8    --100000ed4: test esi,esi
9    X86.ZeroFlag := (X86.ESI = 0);
10   X86.SignFlag := (X86.ESI > X86.MaxSignedInt32);
11   X86.OverflowFlag := False;
12   --100000ed6: jle 100000eec <_zero_array+0x18>
13   if (X86.ZeroFlag or X86.SignFlag /= X86.OverflowFlag) then
14     --100000eec: f3 c3 repz ret
15     X86.RSP := X86.RSP + 8;
16     return;
17   end if;
18   --100000ed8: mov eax,0x0
19   X86.Write_EAX(0);
20   loop
21     --100000edd: DWORD PTR [rdi+rax*4],0x0
22     X86.WriteMem32(X86.RDI +(X86.RAX*4), 0);
23     --100000ee4: add rax,0x1
24     X86.RAX := X86.RAX + 1;
25     --100000ee8: cmp esi,eax
26     X86.ZeroFlag := ((X86.ESI - X86.EAX) = 0);
27     X86.SignFlag := (X86.ESI < X86.EAX);
28     X86.OverflowFlag := ((X86.SignFlag and (X86.EAX > X86.MaxSignedInt32) and
29      (X86.ESI <= X86.MaxSignedInt32)) or ((not X86.SignFlag) and
30      (X86.ESI > X86.MaxSignedInt32) and (X86.EAX <= X86.MaxSignedInt32)));
31     --100000eea: jg 100000edd <_zero_array+0x9>
32     exit when(not(X86.ZeroFlag=False and X86.SignFlag=X86.OverflowFlag));
33   end loop;
34   --100000eec: repz ret
35   X86.RSP := X86.RSP + 8;
36   return;
37 end zero_array;
```

Figure 8. Implementation of zero_array

modifications simplify the development of constraints and proofs; however, modifications add overhead and do not allow isolation of weaknesses in the original binary program. In our approach, binary modifications are not necessary, but could be used as a last resort when proofs cannot be established.

AUSPICE is also an approach based on a model of the ARM ISA using Hoare logic [5]. AUSPICE supports security property verification for binary programs without the need for modifications. To avoid manual development of invariants and function pre-/post-conditions, AUSPICE makes simplifying assumptions. In particular, machine-code instructions are not allowed to alter memory addresses greater than the current function's frame pointer address. This restriction is not practical for most real-world programs.

Prior versions of the Binary Analysis Platform (BAP) support some security analysis through manual insertion of predicates into intermediate representations of the binary program [4]. This approach is limited to intraprocedural analysis of functions that do not call other functions. Further, the BAP approach does not complete proofs unless loops are unrolled and the code is free of indirect jumps. More recent versions of BAP no longer appear to support formal analysis.

5 Conclusion

Reverse engineering of binary programs into a formal language and including formal specifications of desired properties admits the possibility of proving those

properties. We have presented our infrastructure based on SPARKPro for proving properties about binary programs. Binary programs are analyzed and translated into SPARK Ada. Properties are specified using SPARK Ada and proven using the SPARKPro toolchain. We illustrated the application of our approach with an example binary program, proving an important security property.

The SPARKPro toolchain has the advantage of being able to run multiple proofs in parallel with most proofs discharged automatically. Additionally, the SPARK Ada representation can be compiled into an executable program that could allow for verification by testing for representational accuracy. A current disadvantage of the toolchain is that, when proofs are not discharged automatically, completing the proof manually can be difficult. We plan to discuss specific details of translating binary programs and producing constraints for security properties in future publications; however, the work presented here lays the foundation for and focuses the direction of further research and development.

Acknowledgments This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under contract W31P4Q–14-C–0086. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. The authors thank the software engineers of AdaCore, in particular, Yannick Moy for providing support.

References

1. Barnes, J.: SPARK: The Proven Approach to High Integrity Software. Altran Praxis (2012)
2. Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., Necula, G.C.: XFI: Software guards for system address spaces. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. pp. 75–88. OSDI '06, USENIX Association, Berkeley, CA, USA (2006)
3. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: First International Conference on Interactive Theorem Proving. pp. 243–258. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
4. Jager, I., Brumley, D.: Efficient directionless weakest preconditions (cmu-cylab-10-002). CyLab p. 27 (2010)
5. Tan, J., Tay, H.J., Gandhi, R., Narasimhan, P.: AUSPICE: Automatic safety property verification for unmodified executables. Working Conference on Verified Software: Tools, Theories and Experimems (VSTTE) (2015)
6. Zhao, L., Li, G., De Sutter, B., Regehr, J.: ARMor: Fully verified software fault isolation. In: Proceedings of the Ninth ACM International Conference on Embedded Software. pp. 289–298. EMSOFT '11, ACM, New York, NY, USA (2011)