Establishing a Refinement Relation between Binaries and Abstract Code

Freek Verbeek
Virginia Tech Blacksburg, USA
freek@vt.edu

Joshua Bockenek
Virginia Tech Blacksburg, USA

Abhijith Bharadwaj
Virginia Tech Blacksburg, USA

Ian Roessle
Joint Artificial Intelligence Center, Washington DC, USA

Binoy Ravindran
Virginia Tech Blacksburg, USA

***Abstract*—This paper presents a method for establishing a refinement relation between a binary and a high-level abstract model. The abstract model is based on standard notions of control flow, such as if-then-else statements, while loops and variable scoping. Moreover, it contains high-level data structures such as lists and records. This makes the abstract model amenable for off-the-shelf verification techniques such as model checking or interactive theorem proving. The refinement relation translates, e.g., sets of memory locations to high-level datatypes, or pointer arithmetic to standard HOL functions such as list operations or record accessors. We show applicability of our approach by verifying functions from a binary containing the Network Security Services framework from Mozilla Firefox, running on the x86-64 architecture. Our methodology is interactive. We show that we are able to verify approximately 1000 lines of x86-64 machine code (corresponding to about 400 lines of source code) in one person month.**

I. Introduction

Formal verification provides a way to get trustworthy correctness proofs of software. Typically, formal verification revolves around source code. In contrast, this paper concerns formal verification of binaries. Binary verification minimizes the trusted computing base (TCB) of the verification effort [1]. It is, however, harder than traditional source code verification, due to the large semantical gap between source and machine code. Traditional methods of verification cannot directly be applied. For example, a binary model typically has a large and unstructured memory model, which creates a huge state space making model checking unscalable. Similarly, applying interactive theorem proving (ITP) to the small step semantics of individual assembly instructions does not scale either, due to the amount of intricate user interaction involved.

This paper provides a methodology for proving a refinement relation between assembly in a binary and abstract code. The abstract code consists of structured control flow, function calls, scoping, local- and global variables, malloc/realloc, and compound data structures such as lists and records. It does *not* have a heap, pointers, or goto's. As a result, the abstract code can easily be verified by traditional methods of verification, i.e., model checking or ITP with Hoare logic. The refinement relation preserves safety- and liveness properties; it suffices to verify the abstract code to verify the binary. An example is given in Figure 1. The start is the original x86-64 assembly. The final result is formally proven correct abstract code.

Binary verification has been an active research field for years. The predominant techniques are *decompilation-into-logic* (DiL) [2], [3], *verified compilation* [4], [5] and *translation validation* [6], [7], [8]. This paper can be seen as an extension to DiL: the abstract code is on a higher level of abstraction than a binary embedded into HOL using DiL, making it easier to verify. The main difference between our approach and both verified compilation and translation validation is that our approach requires few assumptions on the source language and the compilation process. In principle, our approach can be used in a setting where source code is unavailable, e.g., proprietary software, native code or components written in assembly to begin with. Section IV will discuss this further.

We demonstrate applicability of the methodology by formally verifying a part of the Network Security Services (NSS) framework included in the Firefox browser[1]. This framework implements security protocols such as SSL and TLS. Dealing with actual production code, instead of artificial examples, requires dealing with binaries that are the result of a complex build toolchain. The code contains advanced control-flow, signed and unsigned arithmetic, various casting between types, large and deeply nested structs, and function calls to both functions inside the current binary as well as outside (system calls and dynamically linked code). Arrays and structs are translated to standard HOL datatypes (i.e., lists and records). Pointer arithmetic is translated to HOL operations such as taking the nth element of a list or dropping a number of elements. We show that we are able to verify approximately 1000 lines of x86-64 machine code (corresponding to about 400 lines of source code) in one person month [2].

II. Methodology

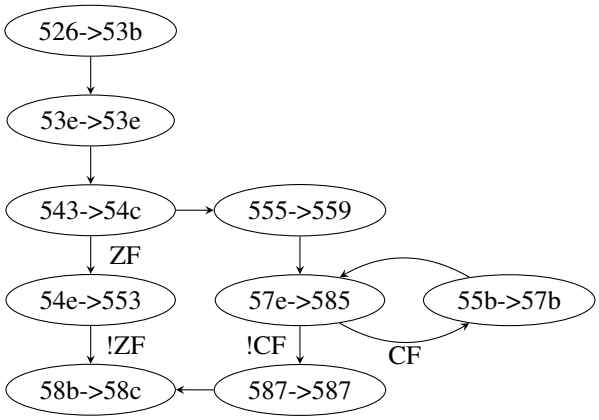
Figure 1 presents – with an example – an overview of the methodology used for obtaining formally proven correct abstract code from a binary. Key is formulating a *simulation relation*. A simulation relation maps *concrete* states to *abstract*

[1] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS

[2] All materials available at: https://filebox.ece.vt.edu/~freek/reassembly_verification-code_abstraction.zip

```
526:  push   rbp
527:  mov    rbp,rsp
52a:  sub    rsp,0x20
52e:  mov    eax,edi
530:  mov    [rbp-0x14],al
533:  movzx  eax,[rbp-0x14]
537:  shl    rax,0x2
53b:  mov    rdi,rax
53e:  call   malloc
543:  mov    [rbp-0x8],rax
547:  cmp    [rbp-0x8],0x0
54c:  jne    555
54e:  mov    eax,0x0
553:  jmp    58b
555:  mov    [rbp-0x9],0x0
559:  jmp    57e
55b:  movzx  eax,[rbp-0x9]
55f:  lea    rdx,[rax*4+0x0]
567:  mov    rax,[rbp-0x8]
56b:  add    rax,rdx
56e:  mov    [rax],0x0
574:  movzx  eax,[rbp-0x9]
578:  add    eax,0x1
57b:  mov    [rbp-0x9],al
57e:  movzx  eax,[rbp-0x9]
582:  cmp    al,[rbp-0x14]
585:  jb     55b
587:  mov    rax,[rbp-0x8]
58b:  leave
58c:  ret
```

(a) Assembly



(b) CFG

```
526->53b;
53e->53e;
543->54c;
if ZF then
  54e->553
else
  555->559;
  while True do
    57e->585;
    if CF then
      55b->57b
    else
      break
    fi
  od;
  587->587
fi;
58b->58c
```

(c) Control Flow

```
55b->57b:
  [(RSP0 - 17),1] := [(RSP0 - 17),1] + 1
  [[(RSP0 - 16),8] + ucast([(RSP0 - 17),1]) * 4,4] := 0
587->587:
  RAX := [(RSP0 - 16),8]
```

(d) Symbolic Execution (two blocks shown)

`[(RSP0 - 17),1]`	$\mapsto$	local variable i
`[(RSP0 - 16),8]`	$\mapsto$	address of array x with 4-byte elements
`RAX`	$\mapsto$	return variable *ret*

(e) Simulation Relation

build_array$(n) =$

1: $x \circ \mathcal{L} \coloneqq$ **malloc**$(n * 4)$;
2: **if** $\lambda\sigma \cdot x(\mathcal{L}\,\sigma) = \text{None}$ **then**
3: $\lambda\sigma\sigma' \cdot$ *ret*$(\mathcal{C}\,\sigma') = \text{None}$
4: **else**
5: $\lambda\sigma\sigma' \cdot i(\mathcal{L}\,\sigma') = 0$;
6: **while** $\lambda\sigma \cdot i(\mathcal{L}\,\sigma) < n$ **do**
7: $\lambda\sigma\sigma' \cdot i(\mathcal{L}\,\sigma') = i(\mathcal{L}\,\sigma) + 1$
8: $\wedge\ x(\mathcal{L}\,\sigma') = x(\mathcal{L}\,\sigma)[i(\mathcal{L}\,\sigma) \coloneqq 0]$
9: **od**;
10: $\lambda\sigma\sigma' \cdot$ *ret*$(\mathcal{C}\,\sigma') = x(\mathcal{L}\,\sigma)$
11: **fi**

(f) Abstract Code

Fig. 1. Example for initializing array

states. A concrete state is the state of the binary, consisting of registers, flags, and memory. An abstract state is defined by the user: it solely contains variables (i.e., no registers, flags or memory).

We start with assembly (see Figure 1a), obtained by running off-the-shelf disassembly tools (in our case `objdump`). We use off-the-shelf tools [9] to extract the control flow graph (CFG) from the binary (Figure 1b). The CFG has as nodes addresses that mark the begin and end of a basic block. From the CFG, the control flow (e.g., while- and if-statements) is inferred (Figure 1c). For each basic block, we run symbolic execution (Figure 1d): the instructions of a basic block are run on some universally quantified, concrete – but completely unspecified – state s. The result is a new concrete state s', with register, flag, and memory updates wrt. s. In the example, for block `[55b->57b]` the new concrete state s' consists of two updates: the 1-byte value at address `RSP0-17` is incremented, and 0 is written to 4 bytes at a certain memory location.

Figure 1e shows the simulation relation. This relation needs to be defined manually. In this example, it maps the 1-byte of memory at location `RSP0-17` in the concrete world to local variable i in the abstract world. It maps the 8-byte

memory region at address RSP0−16 to an array-pointer, and the register RAX to the return variable *ret*.

Finally, the abstract code for the binary as a whole is the result of combining the inferred control flow and the abstract code per block (Figure 1f). For each basic block, the simulation relation maps the concrete state s' to an abstract state σ' (we will use resp. s and σ to denote concrete and abstract states). For example, for basic block [55b->57b], applying the simulation relation to s' produces an abstract state σ' in which local variable i has been incremented and one element of array x has been assigned 0 (see lines 7 and 8 of Figure 1f).

Symbolic execution is done with a custom proof method built in Isabelle / HOL [10] and is formally proven correct wrt. a given *machine model*. The control flow is inferred based on informal tools and manual inspection; the result is formally proven in Isabelle/HOL to be a simulation of the original binary. All rewrite and inference rules have been formally proven correct, except for our treatment of malloc. The TCB of our approach consists solely of: 1.) the machine model, 2.) the embedding of the binary in Isabelle/HOL, 3.) our semantics given to malloc.

Some limitations include: we do not support concurrent code, self-modifying code, or indirect branching. We do not prove termination. We have implemented the approach for x86-64 specifically; implementing it for other architectures is mostly a matter of engineering.

III. Applications

As a case study, we have selected a number of functions from the Firefox NSS library. Moreover, we have applied our methodology to several standard examples, such as the Linux wc program, functions such strncmp and memcpy, and to a binary containing functions pertaining a stack data structure written in C++. The NSS case study was selected based on the following criteria. First, complexity: the functions contain advanced flow control, do pointer arithmetic, contain arrays, deeply nested records, linked lists, malloc and realloc, etc. Second, relevance: the functions concern the SSL functionality of Firefox and are security-critical.

Table I provides an overview. We have used Isabelle/HOL and made heavy use of 1.) its word library [11], 2.) Eisbach, its library for defining proof methods [12], 3.) Sledgehammer, its tool for proof automation [13], and 4.) its integrated SMT solvers Z3 [14] and CVC4 [15]. The second column provides the number of assembly instructions in the binary. For each function, we have inferred abstract code, and established a formal simulation relation between the binary and that abstract code. The abstract code is then translated to Promela and verified using SPIN 6.4.8. In SPIN, assertions are added to verify basic sanity conditions, such as no null-pointer dereferences or buffer overflow. Column 3 shows the fully explored state space (states / transitions). We exposed some simple undocumented preconditions concerning integer parameters that should be non-zero. Since the functions do not have a formal specification, it is not possible to determine whether these are actually indicators of bugs. The functions where Column 3 is empty are not verified in isolation, since they do not do anything interesting on their own. Instead, they are all called by function sslBuffer_AppendVariable, so this function was verified. The state space of this function could not be explored fully. We have modified some of its constants, e.g., decreasing a maximum list size, to allow a state space exploration of this function and all callees.

We consider a code snippet from the NSS library as example. It demonstrates how structs are dealt with and how pointer arithmetic is abstracted to list operations. Consider the C code below (the code is shown just for purpose of presentation, it was not used during verification). The code uses two C structs. An sslReadBuffer consists of a pointer buf of type PRUint8* (platform-independent 8-bit integer) and a length len. Struct sslReader is defined by an offset offset and an sslReadBuffer ssl_buf. The code copies the offsetted pointer of the given reader to out.

```
#define SSL_READER_CURRENT(r)
        ((r)->buf.buf + (r)->offset)

SECStatus sslRead_Read
   (sslReader *reader, ...,
    sslReadBuffer *out) {
  ...
  out->len = count;
  out->buf = SSL_READER_CURRENT(reader);
  ...
}
```

The abstract code defines two records that match the structs. For example, the sslReadBuffer record contains a field buf of type "8 word list option". The state of the caller must include a reader and an out: they are call-by-reference variables. Code abstraction provides the following code (only a part is shown):

$$\begin{aligned} &\texttt{len}(\texttt{out}(\mathcal{C}\,\sigma')) = \mathit{count} \\ \wedge\ &\texttt{buf}(\texttt{out}(\mathcal{C}\,\sigma')) = \\ \mathrm{drop}(&\texttt{offset}(\texttt{reader}(\mathcal{C}\,\sigma)),\ \ \texttt{buf}(\texttt{ssl\_buf}(\texttt{reader}(\mathcal{C}\,\sigma)))) \end{aligned}$$

In words, copying an offsetted pointer is abstracted to dropping a certain amount of elements from a list. In similar fashion, we have proved a simulation between various pointer-based operations in assembly and abstract list operations. For example, for one of the NSS functions we proved an assembly implementation of a linked list of structs equivalent to a recursive HOL data structure.

IV. Related Work

This work is inspired by and builds upon two major results in binary verification: *decompilation-into-logic* [2], [3] and *translation validation* [6], [7], [16], [8].

Decompilation-into-Logic (DiL) reads machine code and embeds it into HOL. Blocks are given pre- and postconditions, and loops are translated to recursive functions. A *certificate theorem* relates the decompiled function to the machine code.

Name	NoA	State Space	Name	NoA	State Space
*_Append	47	13K / 13K	*_AppendNumber	48	27K / 29K
*_AppendVariable	85	Inconclusive	*_Grow	70	135 / 139
*_InsertLength	49	1.5M / 1.6M	*_read	63	1.3M / 1.4M
*_ReadNumber	83	124K / 132K	*_ReadVariable	53	1.5M / 1.6M
*_Skip	43	92 / 95	***	75	1.5M / 1.6M
*_Clear	36	45 / 45	**	154	
**_Number	37		**_Variable	36	
sslServerCert	65		SSL_EncodeUintX	43	

TABLE I
*=SSLBUFFER,**=SSL3_APPENDHANDSHAKE,***=SSL_GETCERTIFICATEREQUESTCAS

It provides preconditions such that the machine code executes correctly (no unspecified behavior). DiL provides more automation than our interactive methodology and has been applied to several architectures. The main difference between this work and DiL is that our abstract code is on a higher level of abstraction. DiL embeds assembly into HOL, this work *lifts* assembly operations (see Figure 1). For example, we translate malloced memory regions to HOL lists, and memory operations (i.e., pointer arithmetic) to list operations. The memory model of the abstract code solely consists of variables. As a result, our abstract code is easily verifiable using off-the-shelf verification techniques. Moreover, we support advanced control flow not supported by DiL.

Translation validation is a technique which considers an artifact written in some source language, and its compiled version. It is then verified whether the semantics of the compiled artifact relate to the semantics of the source. Sewell et al. successfully applied translation validation to verify the binary of seL4 [8]. They prove a refinement relation between compiled ARM machine code and C source code. This method shows excellent scalability. Translation validation inherently requires source code. This paper can be seen as a step towards translation validation without source code, i.e., *inference validation*. The cost of the absence of source code is that more manual interaction is required in defining a simulation relation, and running symbolic execution to create abstract code.

Significant results have been achieved in verified compilation [17], [18], [19], [20]. The CompCert project[4], [21], [22], [23] provides verified compilation for C99 with optimizations. The tool-chain CakeML provides proof synthesis and in-logic execution of ML code [5]. Verified compilation is top-down, whereas our approach can be considered bottom-up.

Finally, various studies apply ITP to verify machine code directly. Myreen et al. present Hoare logic for machine code [24], [25]. Similar to this work, their logic deals with complexities typical for binaries, e.g., stack frames, heap and the binary are within the same memory space. Goel et al. use the ACL2 theorem prover to verify properties over x86 machine code [26]. They verify, among others, a simplified version of the program `wc`. Our treatment of basic blocks is similar to them: using symbolic execution and automated methods such as SAT solvers, basic blocks can be dealt with largely automatically. Matthews et. al. use ACL2 combined with a simple machine model called TINY as well as Java bitcode to apply VCG to machine code [27]. Both of these languages feature a stack rather than registers such as x86, ARM, and most other mainstream ISAs do. Generally, applying ITP directly to machine code suffers from scalability due to the amount of user interaction required. Studies typically either assume a simplified machine model, or modify code to simplify proofs. In contrast, we target unmodified production code of the Firefox browser running on x86-64.

V. CONCLUSION

This paper presents a methodology for establishing a refinement relation between assembly in a binary and abstract code. We show that the abstract code can easily be verified by off-the-shelf verification techniques. The abstract code has a relatively small state space and closely matches PROMELA, the verification language of the model checker SPIN. This allows explicit-state model checking to verify safety- and liveness properties. Moreover, within the same formal environment as the code abstraction, Hoare logic can be used to interactively prove functional properties over binaries.

The methodology is applied to programs compiled for the x86-64 architecture. We consider programs with non-tail-recursive functions, various types of loops, function calls to both in- and outside the binary, type casting, signed and unsigned arithmetic, mallocs / reallocs, arrays, large and deeply nested structs, and linked lists. For each function, we derive abstract code where pointer arithmetic concerning arrays is translated to standard HOL list operations, and pointer arithmetic concerning structs is translated to HOL records. When applied to production code of the Firefox Mozilla browser, we were able to verify approximately 1.000 lines of machine code in one person month.

In the near future we aim to achieve more automation. This requires the development of proof methods that further automate symbolic execution, flow control inference, and inference of abstract operations out of concrete state changes. The most intricate part – from a user perspective – is defining the simulation relation. Further research can focus on deciding a simulation relation automatically. Ultimately, we aim to automatically infer formal abstract specifications from black-box binaries.

ACKNOWLEDGMENT

We thank the reviewers for their insightful comments which have significantly improved the paper. This work is supported in part by ONR under grant N00014-17-1-2297 and NAVSEA/NEEC under grant N00174-16-C-0018.

References

[1] R. Kumar, E. Mullen, Z. Tatlock, and M. O. Myreen, "Software verification with ITPs should use binary code extraction to reduce the TCB," in *International Conference on Interactive Theorem Proving*. Springer, 2018, pp. 362–369.

[2] M. O. Myreen, M. J. C. Gordon, and K. Slind, "Machine-code verification for multiple architectures – an application of decompilation into logic," in *2008 Formal Methods in Computer-Aided Design*, Nov 2008, pp. 1–8.

[3] M. O. Myreen, M. J. Gordon, and K. Slind, "Decompilation into logic – Improved," in *Formal Methods in Computer-Aided Design (FMCAD), 2012*. IEEE, 2012, pp. 78–81.

[4] X. Leroy *et al.*, "The CompCert verified compiler," *Documentation and users manual. INRIA Paris-Rocquencourt*, 2012.

[5] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 179–191.

[6] H. Samet, *Automatically proving the correctness of translations involving optimized code*. Stanford University, 1975, vol. 259.

[7] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998, pp. 151–166.

[8] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified os kernel," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 471–482.

[9] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17)*, 2017.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[11] J. Dawson, "Isabelle theories for machine words," *Electronic Notes in Theoretical Computer Science*, vol. 250, no. 1, pp. 55–70, 2009.

[12] D. Matichuk, T. Murray, and M. Wenzel, "Eisbach: A proof method language for Isabelle," *Journal of Automated Reasoning*, vol. 56, no. 3, pp. 261–282, 2016.

[13] J. C. Blanchette, S. Böhme, and L. C. Paulson, "Extending sledgehammer with smt solvers," in *International Conference on Automated Deduction*. Springer, 2011, pp. 116–130.

[14] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[15] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 171–177.

[16] G. C. Necula, "Translation validation for an optimizing compiler," in *ACM sigplan notices*, vol. 35, no. 5. ACM, 2000, pp. 83–94.

[17] A. Chlipala, "A certified type-preserving compiler from lambda calculus to assembly language," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 54–65.

[18] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[19] M. O. Myreen, "Verified just-in-time compiler on x86," in *ACM Sigplan Notices*, vol. 45, no. 1. ACM, 2010, pp. 107–118.

[20] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond, "Verified compilation of floating-point computations," *Journal of Automated Reasoning*, vol. 54, no. 2, pp. 135–163, 2015.

[21] D. Kästner, X. Leroy, S. Blazy, B. Schommer, M. Schmidt, and C. Ferdinand, "Closing the gap – the formally verified optimizing compiler CompCert," in *Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium*, ser. SSS'17. Bristol, United Kingdom: Safety-Critical Systems Club, Feb. 2017, pp. 163–180. [Online]. Available: https://hal.inria.fr/hal-01399482

[22] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "CompCertTSO: A verified compiler for relaxed-memory concurrency," *Journal of the ACM (JACM)*, vol. 60, no. 3, p. 22, 2013.

[23] R. Krebbers, X. Leroy, and F. Wiedijk, "Formal C semantics: CompCert and the C standard," in *International Conference on Interactive Theorem Proving*. Springer, 2014, pp. 543–548.

[24] M. O. Myreen and M. J. Gordon, "Hoare logic for realistically modelled machine code," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 568–582.

[25] M. O. Myreen, A. C. Fox, and M. J. Gordon, "Hoare logic for arm machine code," in *International Conference on Fundamentals of Software Engineering*. Springer, 2007, pp. 272–286.

[26] S. Goel, W. A. Hunt, M. Kaufmann, and S. Ghosh, "Simulation and formal verification of x86 machine-code programs that make system calls," in *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '14. Austin, TX: FMCAD Inc, 2014, pp. 18:91–18:98. [Online]. Available: http://dl.acm.org/citation.cfm?id=2682923.2682944

[27] J. Matthews, J. S. Moore, S. Ray, and D. Vroon, "Verification condition generation via theorem proving," in *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2006, pp. 362–376.