Formal Verification of Memory Preservation of x86-64 Binaries

Joshua A. Bockenek[1], Freek Verbeek[1], Peter Lammich[2], and Binoy Ravindran[1]

[1] Virginia Tech, Bradley Department of Electrical and Computer Engineering, Blacksburg VA 24061, USA {jabocken,freek,binoy}@vt.edu
https://ece.vt.edu/

[2] University of Manchester, School of Computer Science, Manchester, UK
peter.lammich@manchester.ac.uk
https://www.cs.manchester.ac.uk/

Abstract. Formal verification of a binary can provide high software assurance, even when the source code is unavailable. It is, however, inherently hard due to the low level of abstraction involved; instead of verifying typed and structured source code, one has to verify machine code or reconstructed assembly. This paper presents a semi-automated methodology for formally verifying memory preservation, as well as register preservation, over disassembled binaries. The methodology is based on formal symbolic execution and Floyd-style verification. We show that the methodology is compositional on the function level, which is crucial for scalability. The methodology works for loops, recursion, and both optimized and non-optimized code. It can be used to expose preconditions required for non-exceptional behavior. We demonstrate applicability by verifying a set of functions from the HermitCore unikernel library.

Keywords: x86-64 · Assembly · Isabelle/HOL · Formal verification

1 Introduction

Building high-assurance software greatly benefits from the usage of formal verification. Typically, formal verification shows that a given piece of source code satisfies a certain property. In contrast, this paper considers formal verification of *binaries*. Binary verification can be applied to legacy software or software whose source code is unavailable, e.g., due to proprietary reasons. Moreover, it significantly reduces the trusted computing base (TCB) of the verification effort.

The drawback of binary verification is the semantical gap between a binary and its source code. The compilation process removes information such as types, control flow structure, and data structures such as arrays. Manual proofs over large sequences of assembly are so intricate and user-intensive that they are practically infeasible, and a fully automated proof methodology is theoretically impossible due to the undecidability of semantic properties over programs (Rice's theorem [11]). An approach is required that automates binary verification as far as possible, but still allows user interaction.

This paper combines interactive theorem proving with automated generation of formal proofs. This *semi-automated* approach to binary verification eliminates the need for large and intricate proofs over assembly blocks while still allowing the user to direct the prover whenever necessary. This constrasts with fully automated methods such as SMT solvers [1, 19]. The approach is tailored for a specific property called *memory preservation*. Memory preservation shows that the memory written to by a program is restrained to specified regions. This can then be used to prove the absence of common memory-related issues, such as buffer overflows or some forms of data leakage (the next section discusses memory preservation further). To achieve scalability, the approach uses function-level compositionality.

The methodology is applied to several functions from the HermitCore unikernel library [15]. HermitCore is an operating system (OS) kernel library aiming to provide real-time guarantees for high-performance computing. The functions have been compiled for the x86-64 instruction set architecture (ISA) using the GNU Compiler Collection (GCC). The functions chosen provide a variety of features, including memory operations, loops, recursion, non-trivial data structures, pointers, and subcalls.

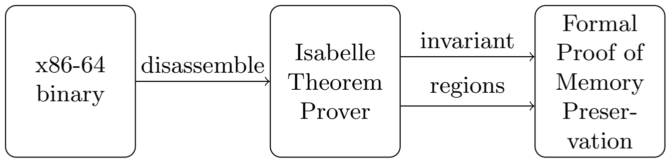

Fig. 1. Overview of methodology

Figure 1 shows an overview of the contribution's methodology. The approach disassembles a binary using an off-the-shelf disassembler, performs analysis on the binary to extract data for automation, and embeds it into a theorem prover using the symbolic execution toolchain of Roessle et al. [26], the machine model of which is based on the work of Heule et al. [9]. Within the theorem prover, two things need to be manually defined: an invariant and the set of regions that the function is allowed to write to. Defining invariants manually is traditionally a hard task, but this paper provides requirements for invariants targeted at memory preservation. Given the manually-added invariants and regions as input, a formal proof of memory preservation is generated largely automatically. The methodology is implemented in Isabelle/HOL [22] for the x86-64 ISA.

2 Memory Preservation

A program that satisfies memory preservation does not write to locations outside of pre-specified memory regions. Memory preservation is an important property for the following reasons:

Security. Various vulnerabilities occur in software whose memory usage is unbounded, such as buffer overflows or data leakage. An example of data leakage from the past few years that had a significant affect on security was the Heartbleed vulnerability, wherein invalid input caused out-of-bounds memory accesses, leaking potentially sensitive data. Memory preservation can be used as a starting point to expose such vulnerabilities.

Composition. Any verification effort over software is scalable only when it is compositional. If one targets proofs of full functional correctness over a large suite of software, that suite needs to be decomposed into separate chunks. Separation logic provides a *frame rule* that allows such decomposition [25]. This rule intuitively states that if a program can be confined to a certain part of a state, properties of this program carry over when the program is part of a bigger system. Memory preservation essentially discharges the most involved part of this frame rule when it comes to functions in a binary: it shows a function is confined to specific regions of the memory. Being able to prove memory preservation is thus a prerequisite for any larger proof effort over binaries.

Concurrency. Reasoning over concurrent programs is complicated due to potential interactions between threads. Interactions can be intended, e.g., via IPC, I/O, or interrupts. Shared memory can be a cause of *unintended* interaction between threads. By showing that the functions in two threads write to specifically allowed regions of shared memory only, unintended interactions can be removed.

2.1 Formal Definition

The formal definition of memory preservation starts with the notion of *state*. In this implementation, states are defined by a record that stores registers, flags, and 64-bit addressable byte-level memory. Moreover, a *machine model* is required. Let S denote the type of states and let A denote the type of instructions. The machine model provides a function step $:: A \times S \mapsto (S \mid \bot_E)$. This function takes as input an instruction and a state σ. It is a partial function, producing either the constant $\bot_E$ (indicating an exception) or some state σ'.

From the machine model, we manually derive a run function run_until $::$ $(S \mapsto \mathbb{B}) \times S \mapsto (S \mid \bot_E \mid \bot_{NT})$. This partial function takes as input a state predicate H and a state σ. Predicate H denotes the *halting condition.* Typically, the halting condition instructs the run function to stop at a certain line of the assembly, such as at a `ret` instruction. The run function iteratively fetches the current instruction via the current value of the instruction pointer and uses the machine model to execute it. Whenever an exception occurs, it stops and returns $\bot_E$. If the execution were to continue forever without an exception or reaching the halting condition (e.g. due to an infinite loop), the function returns $\bot_{NT}$. Formally, this is achieved by a standard least-fixed-point construction.

A *Hoare triple* denotes a pre- and postcondition for a certain program. Let P and Q be state predicates. In our notation, $\{P\}\ H\ \{Q\}$ denotes that, for any state σ, assuming precondition P and termination, run_until(H, σ) produces a non-exceptional state that satisfies postcondition Q. Note that this differs from standard textbook Hoare triples [10,20] as it uses a halting condition instead of an explicit program statement. Instead, the program statement is characterized by the addresses of its initial and ending instructions, defined in P and H.

Before memory preservation can be defined, some further notations and definitions need to be introduced. A *memory region* $[a, s]$ is defined by its address a (a 64-bit word) and size s in bytes (a natural number). A memory region is assumed not to overflow, i.e., the address plus the size is less than 2^{64}. To read a region of memory in the state, we use the notation $\sigma : *[a, s]$. If it is clear from context which state is meant, that state will be omitted. This function reads a list of bytes from the given address, reverses it (since we are dealing with a little-endian architecture) and converts it to a word. The following notation denotes writing a word v to address a in state σ: $\sigma(\!| a \overset{M}{=} v |\!)$. This function decomposes the given word into bytes, reverses them and then writes it into memory. Note that an explicit size is not necessary, since that information is enclosed in the type of v. Similarly, operators $\overset{R}{=}$ and $\overset{F}{=}$ write to registers and flags, respectively; $:=$ is also used for register assignment in places. Central notions concerning memory regions are separation, enclosure, and overlapping:

Definition 1. *Two regions $r = [a, s]$ and $r' = [a', s']$ are* separated*, $r \bowtie r'$, if and only if $s = 0 \vee s' = 0 \vee a + s \leq a' \vee a' + s' \leq a$. Region $r = [a, s]$ is* enclosed by *region $r' = [a', s']$, $r \sqsubseteq r'$, if and only if $a \geq a' \wedge a + s \leq a' + s'$. Two regions* overlap *if they are not separate.*

Memory preservation is defined as a Hoare triple. Assume a predicate P that characterizes the initial state, e.g., sets the instruction pointer to the first instruction of a function body. Moreover, let R be a set of regions that the function is allowed to write to. Set R includes the stack frame and utilized data sections from the binary as well as any utilized heap memory. Memory preservation formulates that any byte not within any region in R has to remain unchanged. This is formalized as follows.

Definition 2. *Let R be a set of regions, let P be a precondition and let H denote a halting condition. A piece of assembly provides* memory preservation *if and only if, for any address a and byte-value v_0:*

$$(\forall r \in R \ \cdot \ r \bowtie [a, 1]) \Longrightarrow \{P \wedge *[a, 1] = v_0\} \ H \ \{*[a, 1] = v_0\} \qquad (1)$$

3 Blocks

This section describes how to prove memory preservation over blocks of assembly. A *block* is defined as a sequence of assembly instructions whose behavior can be described using only state transitions and branches. A block always terminates and has no loops.

3.1 Symbolic Execution

The main proof technique applied is *symbolic execution*, which uses rewrite rules to establish the semantics of a block. Since we do symbolic execution within Isabelle/HOL, each rewrite rule is formally proven correct. Rewrite rules essentially allow lifting the level of abstraction. For example, the next subsection defines rewrite rules for writing into memory. Instead of unfolding the write function – which contains details on byte-level little-endian memory – the write function is kept abstract: the fact that writing decomposes a value into a byte list and reverses it is invisible in the rewritten state.

An inherent difficulty caused by symbolic execution is the alias problem. Consider the following symbolic state: $\sigma(\!|a \overset{M}{=} v, a' \overset{M}{=} v'|\!)$. Two values have been written into memory, first value v to address a and then value v' to address a'. The addresses are however completely symbolic, meaning that it is unknown whether regions $[a, |v|]$ and $[a', |v'|]$ overlap or not ($|x|$ meaning the size of value x). If they do not overlap, then this is indeed the most concise symbolic representation of the current state. In that case, reading from address a will simply return value v. However, problems occur when the regions do overlap. Consider, e.g., $a = a'$ and $|v| = |v'|$. In that case, the most concise symbolic representation is actually $\sigma(\!|a \overset{M}{=} v'|\!)$. Reading from address a will then return v' instead of v. This becomes more complicated when the regions do overlap but the addresses are not equal or the sizes of the values are different, such as when writing multi-byte objects into a byte array and vice versa.

3.2 Rewriting of Memory Accesses

Symbolic execution of a block of assembly will result in a symbolic state with a series of memory writes: $\sigma(\!|a_0 \overset{M}{=} v_0, a_1 \overset{M}{=} v_1, \ldots|\!)$. In order to read from such a state, the alias problem must be solved: if it is unknown whether any of the written regions overlap, then reading from memory cannot be resolved deterministically. To solve the alias problem, rewrite rules are formulated that ensure that the symbolic state always satisfies the following form: any two regions written to memory are separate. This is an invariant over the form of the symbolic state, e.g., it prevents a state of the form $\sigma(\!|a_0 \overset{M}{=} v_0, a_0 \overset{M}{=} v_0|\!)$. Given this invariant, reading a region $r = [a, s]$ can be achieved by looping over the written regions $r_0, r_1, \ldots$ one by one. If a region r_n is found such that $r \sqsubseteq r_n$, then a value can be read. Any region r_n such that $r \bowtie r_n$ can be ignored. It might be possible that no single written region encloses region r completely, but a set of written regions encloses it. In that case, that set of regions can be *merged* into one region. Subsequently, that new region encloses region r and can thus be used to resolve the read.

Writing to memory. In order to preserve the region separation invariant, writing a region into memory can require region merging, defined as follows. Let r_0 be the region to be written and let r_1 be a region already in memory.

If the regions overlap, the state after having written region r_0 will contain one region that is the result of overwriting region r_1 with r_0. To define that merged region, we use list functions $\mathrm{tk}(n, x)$ and $\mathrm{dr}(n, x)$ (for taking/dropping the first n elements of list x) and list appending (@). The merged region is defined as $\mathrm{mrg}([a_1, v_1], [a_0, v_0]) \stackrel{\text{def}}{=} [\min(a_0, a_1), r']$, where

$$\begin{aligned} r' = \mathrm{tk}(\max(0, a_0 - a_1), v_0) \mathbin{@} v_0 \mathbin{@} \\ \mathrm{dr}(a_0 + |v_0| < a_1 + |v_1| \ ?\ a_0 + |v_0| - a_1 \ :\ |v_1|, v_1) \end{aligned} \tag{2}$$

Rewrite Rule 3 shows the rewrite rule used whenever a new region is written into memory. That rule preserves the necessary invariant. The right hand side underlines the redexes in the rewritten statement (note this notation is only used for this particular rewrite rule). That is, after application of this rewrite rule, non-underlined parts will not be rewritten any further. For this rule only, we use an alternative notation for writing to memory: e.g., $\sigma(\!|a_0 \overset{M}{=} v_0, a_1 \overset{M}{=} v_1|\!)$ is equivalent to $\mathrm{w}(a_1, v_1, \mathrm{w}(a_0, v_0, \sigma))$, and we also have $r_0 = [a_0, |v_0|]$ and $r_1 = [a_1, |v_1|]$.

$$\mathrm{w}(a_0, v_0, \mathrm{w}(a_1, v_1, \sigma)) \equiv \begin{cases} \mathrm{w}(a_1, v_1, \underline{\mathrm{w}(a_0, v_0, \sigma)}) & \text{if } r_0 \bowtie r_1 \\ \underline{\mathrm{w}(\mathrm{mrg}(r_1, r_0), \sigma)} & \text{otherwise} \end{cases} \tag{3}$$

In order to admit this rule to the Isabelle/HOL logic, it needs to be formally proven correct. The proof is based on two lemmas. First, writing separate blocks is commutative. Second, the merge function is correct: the produced region is the result of two sequential and overlapping memory writes.

Reading from memory. Let $r = [a, s]$ be a region to be read in a state with a series of memory writes. Rewrite Rule 4 provides a rule for this case.

$$\sigma(\!|a_1 \overset{M}{=} v_1|\!) : *[a, s] \equiv \begin{cases} \mathrm{tk}(s, \mathrm{dr}(a - a_1, v_1)) & \text{if } [a, s] \sqsubseteq [a_1, |v_1|] \\ \sigma : *[a, s] & \text{if } [a, s] \bowtie [a_1, |v_1|] \end{cases} \tag{4}$$

If an enclosing region has been found, the read can occur. A separate region can be ignored. However, the rule is incomplete: the memory might contain a written region that overlaps with r but does not enclose it. Two cases can arise. First, it can be the case that the set of overlapping regions is still not sufficient to enclose region r. In that case, no further rewriting is possible. This corresponds to a case where memory that has not been written to is read. The second case occurs when there is a set of overlapping regions enclosing region r. In that case, those regions have to be merged before Rule 4 can be applied. The proof of Rewrite Rule 4 is among other things based on correctness of the functions that a.) split a word value into a byte list, b.) reverse that list, and c.) concatenates that byte list back to a word value.

3.3 Reasoning over Memory Regions

The previous subsection showed that we need to reason over separation and enclosure of memory regions. Given assumptions on the memory layout, it needs to be automatically inferred whether two regions overlap or not. We first detail how to formulate these assumptions, and then show what steps are needed to set up automatic inference of memory region properties.

Without any assumptions, the memory model is a simple flat function from 64-bit words to bytes. Symbolic execution then places the data sections of a binary in some part of the memory and places the stack frame in some other part of the memory. Naturally, these should not overlap. We use function $\bigotimes$ to formulate such assumptions. This function takes as input a set of regions annotated with a unique ID. This ID allows reasoning over (in)equality of regions: without an ID, it is impossible to decide whether two regions of the same size are equal.

Definition 3. *Let R be a set of pairs of unique IDs and regions. Set R is* separated *if and only if all of its regions are separated:*

$$\bigotimes(R) \overset{def}{=} \forall (i_0, r_0), (i_1, r_1) \in R \;\cdot \mathit{if}\ i_0 = i_1\ \mathit{then}\ r_0 = r_1\ \mathit{else}\ r_0 \bowtie r_1 \tag{5}$$

Typically, set R contains large regions, such as the stack frame. The rewrite rules typically concern small regions, such as the region of a local variable within the stack frame. We thus need rules that infer properties over small regions from larger ones.

$$r_0 \bowtie r_1 \equiv r_1 \bowtie r_0 \tag{6}$$
$$r_0 \sqsubseteq r_2 \wedge r_1 \sqsubseteq r_3 \wedge r_2 \bowtie r_3 \Longrightarrow r_0 \bowtie r_1 \tag{7}$$
$$r \sqsubseteq r \tag{8}$$
$$r_0 \sqsubseteq r_1 \wedge r_1 \sqsubseteq r_0 \Longrightarrow r_0 = r_1 \tag{9}$$
$$r_0 \sqsubseteq r_1 \wedge r_1 \sqsubseteq r_2 \Longrightarrow r_0 \sqsubseteq r_2 \tag{10}$$
$$r_0 \bowtie r_1 \wedge r_0.\mathit{size} \neq 0 \wedge r_1.\mathit{size} \neq 0 \Longrightarrow r_0 \not\sqsubseteq r_1 \tag{11}$$
$$\bigotimes(R) \wedge (i_0, r_0), (i_1, r_1) \in R \wedge i_0 \neq i_1 \Longrightarrow r_0 \bowtie r_1 \tag{12}$$

Fig. 2. Rewrite rules for properties over memory regions.

Figure 2 shows such rules. These rewrite rules are able to infer, from the assumptions over larger regions, the properties separation and non-enclosure over smaller regions. However, they can *not* sufficiently infer enclosure. Often, the only way to prove enclosure is to unfold its definition. This introduces two inequalities over words (see Definition 1). Such inequalities can be solved using the Isabelle/HOL tool *unat_arith*, which is a solver for arithmetic bit-vector equations [6]. This tool is augmented with several heuristics and auxiliary lemmas to facilitate proofs of enclosure. These proofs are time-consuming and can significantly clutter the proof effort. Therefore, we introduce the concept of *parent regions*. A parent region is a member of set R, and is thus a region annotated with an ID. The parent region for each memory region occurring in an assembly block must be manually established. For example, local variables have as parent region the stack frame, whereas constants have as parent frame some data section. The following notation is used to link a memory region r_0 to a parent region r_1 with ID i: $\text{parent}(r_0, i, r_1)$. The parent regions are thus manually defined. Given that information, the proof of enclosure is done automatically, and only once. The established enclosure properties are then used in the inference based on the rules in Fig. 2.

As a concrete example, consider a two-byte array starting at address 10 and having ID 5. The region for this array would be $[10, 2]$, with ID formulation $(5, [10, 2])$. If we take the two bytes of the array as child regions, the region relations would be $\text{parent}([10, 1], 5, [10, 2])$ and $\text{parent}([11, 1], 5, [10, 2])$.

4 Loops

When using symbolic execution to analyze code, loops pose a significant problem. First, they result in significant path explosion. There exist methodologies to reduce the number of paths to execute when using loops [23, 27]. However, these are not formally verified and therefore not usable within Isabelle/HOL. Second, deciding the looping condition on a symbolic state may produce nondeterminism, which can cause symbolic execution itself to loop infinitely.

We instead apply a method similar to Floyd verification [7]. This style of verification assumes that, for each loop, at least one instruction is annotated with a state predicate. In this way, blocks lie between annotated state pairs. If, for each annotated state, the succeeding annotated state satisfies its state predicate, a Hoare triple can be inferred for the program as a whole. Floyd-style verification allows breaking up a larger program with loops into smaller blocks, each of which is verifiable using symbolic execution.

A *Floyd invariant* is a function $I :: L \mapsto ((S \mapsto \mathbb{B}) \mid \bot)$. For each program location L it can optionally provide a state predicate. We use $\text{loc}(\sigma)$ to get the location of the given state (e.g., the current instruction pointer). Notation $I(\sigma)$ applies the Floyd invariant to the current state, i.e., $I(\sigma) = I(\text{loc}(\sigma)) \neq \bot \wedge I(\text{loc}(\sigma), \sigma)$.

Definition 4. *A Floyd invariant* I holds *if and only if, for any state* σ*,*

$$I(\sigma) \longrightarrow \sigma' \neq \bot_E \wedge (\sigma' = \bot_{NT} \vee I(\sigma')), \tag{13}$$

where $\sigma' = \text{run\_until}((\lambda\sigma \cdot I(\text{loc}(\sigma)) \neq \bot), \sigma)$*.*

If the Floyd invariant holds in the current state σ, then running to the next annotated location does not produce an exception. If it terminates, the produced state σ' satisfies the Floyd invariant.

The following theorem states that a Floyd invariant can be used to prove a property over the program as a whole:

Theorem 1. *Assume that Floyd invariant I holds and provides an annotation for locations l_0 and l_f (the initial and final location). Let halting condition H stop at location l_f, i.e., $H(\sigma) \longrightarrow \text{loc}(\sigma) = l_f$. Then $\{I(l_0)\}\ H\ \{I(l_f)\}$.*

Intuitively, Floyd style verification allows a program to be modeled as a control flow graph (CFG). In that CFG, each arrow can be seen as an implication.

5 Composition

Compositionality is crucial for scalability. It is required for two different reasons. First, at the level of function calls, compositionality should ensure that when a function is called, a previous verification effort over that function can be reused, without opening up the function body. Second, compositionality can drastically improve scalability *within* a function body as well. Consider the following pseudocode, which sequentially executes an if-statement and some program P:

`if` b `then` x `else` y; P

The assembly code corresponding to this code can be verified using symbolic execution. This would first consider the case where b is true, execute x and subsequently symbolically execute program P. Then it would consider the case where b is false, execute y and then P. Program P is thus symbolically executed twice. Without compositionality, programs with if-statements may require certain parts to be executed a number of times exponentially in the number of if-statements. With compositionality, program P needs to be symbolically executed only once.

The notion of Hoare triples as defined in this paper (see Section 2.1) uses a halting condition. Standard composition [10, 20] does not apply to this kind of Hoare triples. Consider a run obtained by halting condition H'. It is possible to break this run into two, by first running until a halting condition H, and then until H'. This requires that H' is *stronger* than H, i.e., H' implies H. This ensures that the run first stops at H before it stops at H'.

Theorem 2. *Hoare triples are compositional with respect to stronger halting conditions:*

$$\frac{\{P\}\ H\ \{Q\} \qquad \{Q\}\ H'\ \{R\} \qquad \forall \sigma \cdot H'(\sigma) \longrightarrow H(\sigma)}{\{P\}\ H'\ \{R\}} \tag{14}$$

Consider the block of assembly associated with the pseudocode example. Let l_f denote the final location, and let l_P denote the initial location of program P. Theorem 2 can be used by instantiating H with halting at either location l_f or l_P, and H' with halting at l_f. Assuming programs x and y do not contain goto's, condition H' is actually equivalent to halting at l_P. Since H' is stronger than H, compositionality is then possible.

Generally, compositionality over function calls requires a proof that the stack pointer remains unchanged after execution of a function call. Consider a function body of function f starting in a text section at location l_0. The function is called from a different text section by `call f` at location l_{call}. This means the return address is $l_{call} + 5$ (the size of the `call` instruction is 5). After execution of the instruction `call f`, the program is at location l_0 and the stack pointer has some value rsp_0. In order to apply compositionality to function calls, the pre- and postcondition have to meet the following requirements. The precondition must imply that the return address is pushed on the stack (which has been done by `call`): $*[rsp_0, 8] = l_{call} + 5 \wedge \texttt{rsp} = rsp_0$.

The postcondition must imply that after `ret`, the net effect of the function body is that the stack pointer has been incremented by 8: $\texttt{rsp} = rsp_0 + 8 \wedge \text{loc} = l_{call} + 5$. Note that `call` has decremented it with 8, so this implies the net effect from the point of view of the caller is that the stack pointer has been unchanged. Also, the postcondition shows that the location has been set back to right after the call.

Besides the stack pointer, modern calling conventions have other *callee-saved* registers, such as `rbp` and `r12-r15`. It is generally assumed that the net effect of a function call does not touch these registers. Consider a situation in which `rbp` contains an address, to which a value is written after a function call. In order to prove memory preservation, it must be known that `rbp` is preserved. Generally, this is easy to prove by strengthening the pre- and postcondition with a conjunct $\texttt{rbp} = rbp_0$. The proof is generally not complicated, since these callee-saved registers are pushed onto the stack at the beginning of the functional calls, and popped at the end.

6 Case Study: HermitCore

A relatively recent trend in the field of virtualization is the usage of *unikernels*: programs designed for specific tasks that are compiled with all the kernel code necessary to run the programs on a hypervisor or even bare metal without an intermediary OS [17]. Unikernels allow an application to include only the necessary parts of the OS, increasing security by reducing the attack surface. HermitCore is such a unikernel [15]. It is designed for the x86-64 ISA and is written in C with some inline assembly. HermitCore is an interesting target for verification, as it aims to provide a high-speed and real-time environment for cloud software. In order to demonstrate the applicability of our methodology, we verified a subset of HermitCore's library functions. These functions contain loops, recursion, structs, unions, pointers, and function calls. Generally, both non-optimized and optimized versions have been verified. The proofs and all associated code are available at https://doi.org/10.6084/m9.figshare.7356110.v2.

Machine Model. The machine model must provide a step function that provides semantics for instructions. We have used the machine model of Roessle et al. [26], which is built upon the work of Heule et al. [9]. Heule et al. used machine learning to derive semantics by executing instructions on an actual x86-64 machine. Their semantics have been validated against the Intel reference manual. The formal model is obtained by embedding these semantics into Isabelle/HOL.

Table 1. Summary of functions analyzed

Functions	Count	SLOC	Insts†	Loops	Recursion	Pointer args	Globals	Subcalls	`-O3`
`dequeue_*`	3	46	159			3		3	3
`buddy_*`	5	67	225	1	1	1	3	3	3
`task_list_*`	3	43	128			3			3
`vring_*`	3	19	80			1			3
`string.h`	8	81	280	8		8			
`syscall.c`	23	293	857	5		19	7	17	
`tasks.c`	10	122	396	2		3	9	4	
`spinlock.h`	8	89	254	2		8	2	6	
Total	71	760	2379	18	1	46	21	33	12

† Non-optimized count

It has been tested against an actual x86-64 machine, increasing the model's reliability. It provides a formalization of large parts of the x86-64 ISA, including several modern instruction sets. Concurrency is not modeled.

Functions Analyzed. The selected functions (see Table 1) include functionality pertaining to a generic circular queue or ring buffer (the `dequeue_*` functions), the internals of HermitCore's `kmalloc` setup (the `buddy_*` functions), task management lists used by HermitCore's scheduler (`task_list_*`), and functions concerning virtual I/O (`vring_*`). We also verified standard string and memory related functions: `memcpy`, `memcmp`, `memset`, `strlen`, `strcpy`, `strncpy`, `strcmp`, and `strncmp`. This verification effort affirmed the well-known fact that some of those functions require an extra precondition, i.e., that the given string is null-terminated; failure to use null-terminated strings and/or using output buffers of too small a size can result in buffer overflows. Additional functions that were verified consist of some providing syscall support, more task- and scheduler-related functions, and functions for manipulating spinlocks.

Figures 3a and 3b show the CFGs for two of those functions, `dequeue_push` and `buddy_large_avail`. The former pushes a value onto a generic array-based queue while the latter checks for the smallest available reused memory block for a given allocation size. The former, lacking any loops, requires only pre- and postconditions (though additional invariants may be added). In contrast, the latter function requires a loop invariant in addition to the pre- and postconditions.

Discussion on Usability. In order to apply the method to a function in a binary, three steps require user interaction: *a*) defining a Floyd invariant, *b*) defining set R, and *c*) proving. Traditionally, defining invariants over software is a complicated matter. However, by restricting ourselves to memory preservation, invariants become significantly easier.

Section 5 provides requirements that define common parts of the invariants. For loops, one simply has to annotate each jump with a state predicate as in

$$\begin{tikzcd}
129 : \begin{array}{c} *[a,1] = v_0 \ \wedge\ \texttt{rsp} = rsp_0 \ \wedge \\ \texttt{rbp} = rbp_0 \ \wedge\ \texttt{rdi} = deq_{\text{ptr}} \ \wedge \\ *[rsp_0, 8] = \texttt{ret\_addr} \end{array} \arrow[d, "\cdots"] \\
\texttt{ret\_addr} : \begin{array}{c} *[a,1] = v_0 \ \wedge \\ \texttt{rsp} = rsp_0 + 8 \ \wedge \\ \texttt{rbp} = rbp_0 \end{array}
\end{tikzcd}$$

(a) `dequeue_push`

$$\begin{tikzcd}
0 : \begin{array}{c} *[a,1] = v_0 \ \wedge\ \texttt{rsp} = rsp_0 \ \wedge \\ \texttt{rbp} = rbp_0 \ \wedge *[rsp_0, 8] = \texttt{ret\_addr} \end{array} \arrow[d, "\begin{array}{l} \texttt{rsp} \coloneqq \texttt{rsp} - 8 \\ \texttt{rbp} \coloneqq \texttt{rsp} \end{array}"] \\
21 : \begin{array}{c} *[a,1] = v_0 \ \wedge\ \texttt{rsp} = rsp_0 - 8 \ \wedge \\ \texttt{rbp} = rsp_0 - 8 \ \wedge \\ *[rsp_0 - 8, 8] = rbp_0 \ \wedge \\ *[rsp_0, 8] = \texttt{ret\_addr} \end{array} \arrow[loop right] \arrow[d, "\begin{array}{l} \texttt{rbp} \coloneqq *[\texttt{rsp}, 8] \\ \texttt{rsp} \coloneqq \texttt{rsp} + 16 \end{array}"] \\
\texttt{ret\_addr} : \begin{array}{c} *[a,1] = v_0 \ \wedge \\ \texttt{rsp} = rsp_0 + 8 \ \wedge \\ \texttt{rbp} = rbp_0 \end{array}
\end{tikzcd}$$

(b) `buddy_large_avail`

Fig. 3. Example Floyd invariants

Fig. 3b. However, for recursion the invariant becomes more complicated. Generally, it has to be shown that both the stack and frame pointers are preserved throughout the recursion. Moreover, it has to be shown that return addresses are pushed correctly. A second interaction in defining a Floyd invariant is finding the right precondition. Such preconditions need not be derived from a reference manual or from source code annotations; instead, users can run symbolic execution until non-determinism occurs. At that point, Isabelle/HOL provides the exact condition under which exceptional behavior happens. It is then up to the user to strengthen the precondition based on that condition. This means that the proof methodology may expose implicit or undocumented preconditions.

The proof effort consists of defining parent relations and running symbolic execution. After symbolic execution, it must be proved that the resulting state satisfies the invariant. In most of the cases, those proofs could be handled by Isabelle/HOL using standard off-the-shelf tools. The exception is again recursion. The proof that the stack and frame pointers preserve their values requires interactive theorem proving with a large focus on word arithmetic.

7 Related Work

Going back to the late 80's and early 90's, Yu and Boyer [3, 28] provided semantics and mechanized reasoning for a subset of instructions of the MC68020 microprocessor in the Boyer-Moore theorem prover (Nqthm) [2], a precursor to ACL2 [12]. This work also utilized symbolic execution and even covered many of the same string functions we did, such as `strcpy` and `strcmp`. Similarly, Clutterbuck and Carré performed formal verification of low-level code using SPACE-8080 [5], a verifiable subset of the Intel 8080 ISA that is analyzable and formally verifiable using the Southampton Program Analysis Development Environment (SPADE) [4]. Another usage of SPADE for verification of assembly was in the correctness proof of fuel control code for a Rolls-Royce jet engine [24].

Decompilation into logic allows formal verification of assembly and machine code [21]. Developed in the HOL4 theorem prover, that work uses operational semantics of machine code to lift programs into a functional form, which can then be used in a Hoare logic framework for program analysis. It has been successfully used with machine models of the ARM ISA. This work builds upon decompilation to derive highly automated proofs for a specific property.

Matthews et al. [18] used the theorem prover ACL2 [12] to target a simple machine model called TINY as well as Java virtual machine (JVM) bitcode using the M5 operational model. Both of these assembly-style languages feature a stack for handling scratch variables rather than a register file as x86, ARM, and most other mainstream ISAs do. They utilize symbolic execution of code annotated with invariants on specific instructions. While they proved functional correctness, they did not show effective scalability due to the restricted models and small amount of code verified.

In contrast to the bottom-up approach presented in this paper, top-down approaches have been studied extensively. The CompCert project [16] provides a compiler that has been verified to produce assembly or machine code with the same semantics as the source, thus removing them from the TCB. A top-down approach requires verification of the original source code as well. One such verification project is AutoCorres [8], part of the seL4 verified microkernel project [13]. This tool parses C code into a shallowly-embedded monadic representation. It produces proofs of the semantic equivalence between the original code and the monadic version and can be used to prove properties via Hoare logic. Another top-down project is CakeML [14], a full-toolchain project for proof synthesis and in-logic execution. It utilizes a subset of Standard ML modeled with big-step operational semantics.

8 Conclusion

Formal verification of binaries can produce highly reliable claims over software. By eliminating trust in a compiler or in the semantics of a source language, the TCB is drastically decreased. It is, however, fundamentally a harder problem than source code verification.

This paper targets formal verification of memory usage in x86-64 binaries, showing that functions in a binary restrict themselves to certain regions of memory. It aims to automate verification as much as possible while still allowing user interaction wherever necessary. This semi-automated methodology requires setting up an invariant, which traditionally is a hard problem in itself. Requirements for memory preservation invariants are provided. For recursive functions, more involved invariants are required, plus interactive theorem proving to show preservation of the stack and frame pointers. Invariants include preconditions necessary for excluding exceptional behavior. Such preconditions are exposed by applying the methodology to a binary, instead of deriving them from documents or source code annotations.

The approach was applied to functions of HermitCore, a unikernel OS. We formally proved memory preservation for functions with loops, recursion, C structs and unions, and dynamic memory operations. Both optimized and non-optimized versions were verified.

Major additions to our framework would be handling of concurrency and related instruction variants. Additionally, proper modeling of virtual machine and hypervisor calls in logic would allow verification of a wider range of functions from the HermitCore library.

Acknowledgments. 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. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: International Conference on Computer Aided Verification. pp. 171–177. Springer-Verlag (2011)
2. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, Inc. (1979)
3. Boyer, R.S., Yu, Y.: Automated proofs of object code for a widely used microprocessor. Journal of the ACM **43**(1), 166–192 (1996)
4. Carré, B.A., ONeill, I.M., Clutterbuck, D.L., Debney, C.W.: SPADE–the southampton program analysis and development environment. In: Software Engineering Environments. Peter Peregrinus, Ltd. Stevenage (1986)
5. Clutterbuck, D.L., Carré, B.A.: The verification of low-level code. Software Engineering Journal **3**(3), 97–111 (May 1988). https://doi.org/10.1049/sej.1988.0012
6. Dawson, J.: Isabelle theories for machine words. Electronic Notes in Theoretical Computer Science **250**(1), 55–70 (2009)
7. Floyd, R.W.: Assigning meanings to programs. Mathematical Aspects of Computer Science **19**(1), 19–32 (1967)
8. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: Beringer, L., Felty, A. (eds.) International Conference on Interactive Theorem Proving. pp. 99–115. ITP 2012, Springer-Verlag, Berlin, Heidelberg (Aug 2012)
9. Heule, S., Schkufza, E., Sharma, R., Aiken, A.: Stratified synthesis: Automatically learning the x86-64 instruction set. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 237–250. PLDI '16, ACM, New York, NY, USA (2016)
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (Oct 1969)
11. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Pearson (2006)
12. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)

13. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 207–220. SOSP '09, ACM Press, New York, NY, USA (2009), https://sel4.systems/
14. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 179–191. POPL '14, ACM, New York, NY, USA (2014), https://cakeml.org/
15. Lankes, S., Pickartz, S., Breitbart, J.: HermitCore: A unikernel for extreme scale computing. In: ROSS '16. pp. 4:1–4:8. ACM, New York, NY, USA (2016)
16. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert - a formally verified optimizing compiler. In: Embedded Real Time Software and Systems, 8th European Congress. ERTS 2016, SEE, HAL, Toulouse, France (Jan 2016), http://compcert.inria.fr/
17. Madhavapeddy, A., Scott, D.J.: Unikernels: The rise of the virtual library operating system. Communications of the ACM **57**(1), 61–69 (Jan 2014)
18. Matthews, J., Moore, J.S., Ray, S., Vroon, D.: Verification condition generation via theorem proving. In: International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 362–376. Springer-Verlag (2006)
19. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer-Verlag (2008)
20. Myreen, M.O., Gordon, M.J.C.: Hoare logic for realistically modelled machinecode. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 568–582. Springer-Verlag, Berlin, Heidelberg (2007)
21. Myreen, M.O., Gordon, M.J.C., Slind, K.: Machine-code verification for multiple architectures - an application of decompilation into logic. In: 2008 Formal Methods in Computer-Aided Design. pp. 1–8. IEEE (Nov 2008)
22. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, vol. 2283. Springer Science & Business Media (2002)
23. Obdržálek, J., Trtík, M.: Efficient loop navigation for symbolic execution. In: Bultan, T., Hsiung, P.A. (eds.) Automated Technology for Verification and Analysis. pp. 453–462. Springer-Verlag, Berlin, Heidelberg (2011)
24. ONeill, I.M., Clutterbuck, D.L., Farrow, P.F., Summers, P.G., Dolman, W.C.: The formal verification of safety-critical assembly code. In: IFAC Symposium on Safety of Computer Control Systems 1988. SAFECOMP '88, vol. 21, pp. 115–120 (Nov 1988). https://doi.org/10.1016/S1474-6670(17)54540-1
25. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on. pp. 55–74. IEEE (2002)
26. Roessle, I., Verbeek, F., Ravindran, B.: Formally verified big step semantics out of x86-64 binaries. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 181–195. CPP 2019, ACM, New York, NY, USA (2019)
27. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. pp. 225–236. ISSTA '09, ACM, New York, NY, USA (2009)
28. Yu, Y.: Automated Proofs of Object Code for a Widely Used Microprocessor. Ph.D. thesis, University of Texas at Austin (1992)