Formally Verified Lifting of C-Compiled x86-64 Binaries

Freek Verbeek
freek@vt.edu
Virginia Tech & Open University
USA & The Netherlands

Joshua Bockenek
jabocken@vt.edu
Virginia Tech
USA

Zhoulai Fu
zhoulai.fu@sunykorea.ac.kr
State University of New York
Korea

Binoy Ravindran
binoy@vt.edu
Virginia Tech
USA

Abstract

Lifting binaries to a higher-level representation is an essential step for decompilation, binary verification, patching and security analysis. In this paper, we present the first approach to *provably overapproximative* x86-64 binary lifting. A stripped binary is verified for certain sanity properties such as return address integrity and calling convention adherence. Establishing these properties allows the binary to be lifted to a representation that contains an overapproximation of all possible execution paths of the binary. The lifted representation contains disassembled instructions, reconstructed control flow, invariants and proof obligations that are sufficient to prove the sanity properties as well as correctness of the lifted representation. We apply this approach to Linux Foundation and Intel's Xen Hypervisor covering about 400K instructions. This demonstrates our approach is the first approach to provably overapproximative binary lifting scalable to commercial off-the-shelf systems. The lifted representation is exportable to the Isabelle/HOL theorem prover, allowing formal verification of its correctness. If our technique succeeds and the proofs obligations are proven true, then – under the generated assumptions – the lifted representation is correct.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; Semantics; • **Security and privacy** → *Logic and verification.*

Keywords: Binary Analysis, Formal Verification, Disassembly

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523702

ACM Reference Format:
Freek Verbeek, Joshua Bockenek, Zhoulai Fu, and Binoy Ravindran. 2022. Formally Verified Lifting of C-Compiled x86-64 Binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523702

1 Introduction

Every technique applicable to binaries, whether it be decompilation [8, 16], binary verification [7, 20, 53], binary patching [27, 59] or security analysis [13, 33, 52, 57], needs to start with some form of binary *lifting*. Raw unstructured data needs to be lifted to a form where one can reason over behavior and semantics. Typically, binary lifting requires an answer to *at least* the following base questions: 1.) what instructions are executed by the binary (disassembly), and 2.) in what *order* can these instructions be executed (control flow recovery)?

These base questions are *intertwined*: neither of them can be answered in isolation. Disassembly requires knowledge of which instruction addresses are reachable from the entry point. Such reachability analysis requires knowledge on control flow, e.g., how jump targets are computed, to what address a `ret` statement returns or what the bounds on indices are when a jump table is read. Simply knowing what influence a `ret` statement has on the control flow already requires establishing that the return address is not overwritten. Any tool that performs such analysis, however, requires at least knowing what instructions are executed. This produces the chicken-and-egg problem of disassembly [49]. Something as ostensibly simple as disassembly requires, at a minimum, establishing absence of stack overflows (for returns), determining upper bounds on array indices (for resolving jump table accesses), dealing with pointer aliasing and adhering to calling conventions.

There is no existing tool that takes as input a binary and answers these two base questions. Both questions are indeed undecidable [25, 46]. The bulk of existing methods are either known to be unsound (e.g., misidentify code as

data or are underapproximative) [49] or are speculative or learning-based [6, 61]. The strength of those tools is their universality: they typically provide output for any binary, even in cases where guesses and non-validated assumptions have to be made. Their weakness is that their outputs are untrustworthy; thus, any analyses built on top of them are untrustworthy as well.

This paper proposes an approach to *trustworthy binary lifting* that simultaneously performs 1.) disassembly, 2.) control-flow recovery and 3.) generation of formal proofs that provide *assurance* of the output. Due to the undecidability of the problem, our approach is not universal: it may fail on certain binaries or annotate certain instructions with unsoundness warnings. However, our approach provides theoretical guarantee that *if* unannotated output is produced, that output is a sound overapproximative representation of the binary. To the best of our knowledge, *no existing work can provide scalable, formally overapproximative assurance between a binary and its lifted representation.*

Our approach verifies the following properties over functions in the binary, each of which is necessary for proving that the generated disassembly and control-flow are sound:

Return Address Integrity Functions do not overwrite their own return address.
Bounded Control Flow All indirect (i.e., dynamically computed) jumps transfer control flow to fixed, statically known, bounded sets of addresses.
Calling Convention Adherence All functions are proven to properly restore the set of registers indicated by the calling convention as non-volatile.

This paper presents a two-step approach to formally verified binary lifting: step 1 lifts a binary while verifying the above properties with algorithms proven correct with pencil-and-paper proofs, whereas step 2 validates that each and any inference made during step 1 can be proven formally correct in Isabelle/HOL [14, 42].

Step 1 consists of an algorithm for extracting a *Hoare graph* (HG) from an x86-64 binary. The vertices of an HG consist of 1.) *predicates* containing information on registers, memory locations and flags and 2.) *memory models* that provide pointer aliasing information. Edges are labeled with disassembled instructions. A key aspect is that the edges are *one-step-inductive*: each edge forms a Hoare triple [24]. Each vertex, i.e., each predicate & memory model, is sufficiently strong to prove that its outgoing edges are overapproximative, even in the case of non-trivial control flow such as indirect branches, jump tables and function calls/returns. In other words, each vertex provides an *invariant* that is sufficiently strong to prove what instructions are executed next.

An overapproximative relation requires that – besides all "normal" behavior – any "weird" behavior [18, 51] is represented as well. Normal behavior consists of, among other things, the intended control flow. "Weird" behavior is a term of art indicating control-flow transfers not intended by the program designers. Section 2 shows an example where instructions are overlapping, which is typically found in obfuscated code. This example exhibits a Return-Oriented-Programming (ROP) gadget that depends on whether two pointers alias or not. The overapproximative HG indeed contains an edge where the ROP gadget unexpectedly hijacks the control flow.

Step 2 exports the HG to the Isabelle/HOL theorem prover. Each edge individually forms a Hoare triple, and thus the formal verification effort consist of proofs of thousands of mutually independent theorems (generally, one per disassembled instruction). Each theorem pertains one Hoare triple, and each theorem can be verified automatically with tailored proof scripts. These proof scripts symbolically execute the formal semantics of the given instruction on the given precondition, and subsequently formally prove the postcondition. The mutual independence of all the theorems allows exploitation of the parallel proof techniques provided by the Isabelle/HOL theorem prover environment.

With regards to Step 1, the Xen hypervisor is used as case study. We lift 45 binaries and 2115 library functions, totaling 399 771 assembly instructions. Both Steps 1 and 2 are applied to some CoreUtils binaries (e.g., `gzip`, `tar`, `hexdump`). Applying Isabelle/HOL to the entire Xen case study requires formal semantics for a larger class of instructions than currently available. All source code, examples and case studies are available as open source.

Limitations, assumptions and scope. The complex nature of binary code necessitates making assumptions. We discuss two main assumptions here. Our approach nondeterministically tries out different memory relations (aliasing, separation, enclosed within, or encloses) when the relation between two pointers is unknown. Enumerating all cases where two pointers point to regions that are *partially overlapping* is infeasible, as this would quickly lead to a state space explosion. For example, two 8 byte regions can partially overlap in 14 ways, with the first byte of one region equal to any other byte of the other region, and the converse (the two other cases are covered by the aliasing case). In such cases, we do not generate a new memory model, but instead simply destroy all regions in memory that may partially overlap, meaning that reading from them always overapproximatively produces a symbolic expression representing *any* value. Note that in compiler generated code, this will typically only happen on the heap [2, 3]. The local stack frame is typically structured into regions that are accessed based on the above four relations. As soon as a memory write occurs to the local stack frame, and the relation between the write-destination and the region where the return address is stored is unknown, return address integrity cannot be proven and the function is rejected due to a verification error.

Second, binary analysis inherently suffers from dealing with *external functions*. Providing accurate models of the

behavior of all external functions called by a binary is generally infeasible, and thus our approach must be able to deal with *unknown* external functions. Theoretically, an overapproximative model should simply destroy the entire state after an unknown external function call. Such rigor would prevent any code to be lifted. We therefore make the assumption that external functions adhere to the 64-bit System V calling convention: the local stack frame is kept intact, as well as certain caller-saved registers. The heap and the global space, however, are destroyed. We generate proof obligations that state that these functions are assumed not to touch the local stack frame of the caller, and under these proof obligations the lifted representation can be shown to be a sound overapproximation of the binary.

In terms of scope, we 1.) target stripped commercial off-the-shelf (COTS) x86-64 binaries in the ELF format compiled with various levels of optimization, 2.) do not deal with multi-threaded code, 3.) do not deal with destructors executed after an exit, and 4.) limit the approach to binaries compiled from C code (specifically, throw-catch behavior and object initialization methods are unsupported). We also assume the existence of sound instruction semantics that express state changes *per instruction* (e.g., semantics that have been machine-learned from actual hardware [22, 47]). We assume the existence of a fetch function that, given an address, soundly retrieves *a single instruction* from the binary. Experimental results show that the majority of unsoundness annotations concern function callbacks. In order to gain scalability, we treat function calls as context free. That means that if a function pointer is passed as a parameter, its concrete value is unknown.

To summarize, this work presents a formal methods approach to disassembly and control-flow reconstruction. This provides assurance, where existing approaches are based on heuristics, machine-learning, or are known to be unsound (see Section 6). The key contributions of this paper are:

- Step 1: trustworthy binary lifting, providing an overapproximative relation between the binary and the output;
- Step 2: a method to formally verify output of Step 1;
- The demonstration that overapproximative binary lifting can be used to find "weird" edges in binaries;
- The application of binary lifting to all non-concurrent x86-64 executables of the Xen hypervisor.

2 Example

Figure 1 shows an example of a binary and (part of) its extracted HG. For the sake of presentation, the example uses 32-bit instructions, and address `a` is the base address of some jump table. First, the `cmp` and `ja` instructions compare the current value of register `eax` to constant value `0xc3`. If `eax` is less or equal than `0xc3`, the `mov` at address `0xb` reads a jump table with base address `a` and the value stored in register `eax` as the index. The pointer read from the jump table (referred to as $a_{\texttt{jt}}$) is stored in register `eax`. Subsequently, two memory writes happen: pointer $a_{\texttt{jt}}$ is written to memory at the address stored in register `edi` and the immediate value 1 is written to memory at the address stored in register `esi`. Finally, pointer $a_{\texttt{jt}}$ is used for an indirect branch. Essentially, this code reads an address from a jump table containing `0xc3` addresses and jumps to that address.

The example is constructed as an example of "weird" control flow. Note the instructions do not contain a return statement. However, under specific circumstances, namely if the pointers in registers `esi` and `edi` alias, the *data* of the first instruction (`0xc3`) is interpreted as an *instruction*. Since this is a real concrete execution path, any overapproximative lifted representation must model such behavior.

We explain several of the points made in the introduction using this example. Also note that the notation at state `14` indicates that reading 4 bytes from address `edi` produces value $a_{\texttt{jt}}$. Respectively, $\equiv$ and $\bowtie$ denote aliasing and separation.

```
0x0:  3dc3000000   cmp eax,c3
0x5:  0f8718000000 ja  1c
0xb:  8b0485__a___ mov eax,DWORD PTR [eax*4+a]
0x12: 8907         mov DWORD PTR [edi],eax
0x14: c70601000000 mov DWORD PTR [esi],1
0x1a: ff27         jmp DWORD PTR [edi]
```

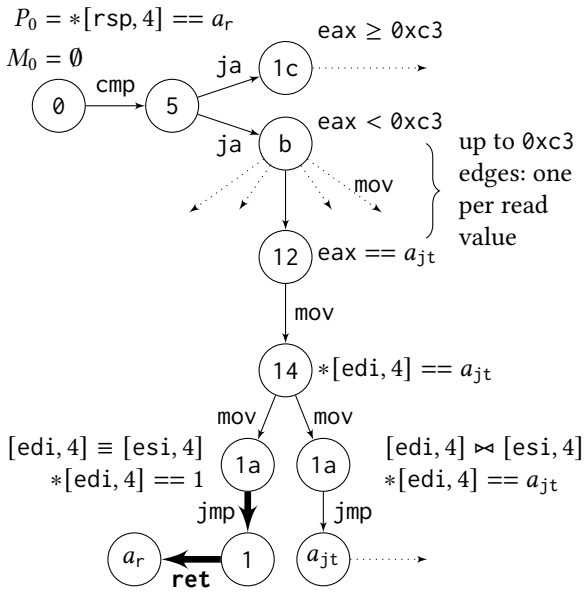


Figure 1. Hoare Graph Example. The bold arrows are "weird" edges leading to unexpected control flow.

The HG is provably overapproximative. Consider the set of outgoing edges at vertex `b`. The predicate associated with that vertex contains the information that register `eax` is bounded. It thus contains sufficient information to prove that reading the jump table provides at most `0xc3` possible values for $a_{\texttt{jt}}$. The bound can be proven only if the predicate associated with vertex `5` contains information on the flags read by the

`ja` instruction. That information is set by `cmp`. In other words, edges `0` → `5` → `b` each form a Hoare triple.

Disassembly requires pointer analysis. At state `14`, it is unknown whether the pointers in registers `edi` and `esi` alias or not. We must overapproximate by having one outgoing edge for each case. In the aliasing (≡) case, the `mov` at Line `14` overwrites the previous `mov`. The jump goes to address `1`, instead of the intended jump to pointer $a_{\texttt{jt}}$.

Disassembly requires bounds analysis. Address $a_{\texttt{r}}$ is the address initially stored at the top of the stack frame. The HG contains an edge to a final state where the instruction pointer is set to that address. To obtain this result, each vertex on the path from state `0` to that final state must contain enough information to show that the return address has not been modified and that frame and stack pointers are managed properly throughout function execution.

Weird edges are found. A jump to address `1` jumps into the middle of an instruction. Since byte `c3` corresponds to the `ret` instruction, this is actually ROP gadget. An unexpected "weird" edge [18, 51] has been found.

The Hoare Graph allows formal verification. The Hoare Graph is generated by the algorithms presented in Sections 3 and 4. Even though these algorithms are proven sound with pencil-and-paper proofs (see Theorem 4.7), one may still want to perform formal verification. The HG can be exported to Isabelle/HOL where each vertex becomes its own theorem. For example, vertex `14` is translated to a Hoare triple that states that the invariant associated to instruction address `14` ensures as postcondition the disjunction of the invariants associated to address `1a`. Essentially, this step removes the need for trusting the implementation of the algorithm presented in this paper.

At first glance, it may seem that a small piece of code leads to an exorbitant number of states and edges. However, typically the state space is close to the number of instruction addresses (see Section 5), as we apply joining of states to reduce the state space whenever possible.

3 Technical Formulation

We use $\mathbb{I}$, $\mathbb{P}$, $\mathbb{M}$ and $\mathbb{W}_n$, respectively, to denote the types for instructions, symbolic predicates, memory models and words of bit length n. Predicates and memory models will be defined and explained in the next section. A vertex of an HG is represented by a *symbolic state*, which is a tuple of type $\mathbb{P} \times \mathbb{M}$ We use σ (s) to denote symbolic (concrete) states. Notation $s \vdash X$ denotes that X holds in concrete state s, where X can be either a predicate or a memory model.

Definition 3.1. A binary is defined by a tuple of the form $\langle a_e, \text{fetch}, S, \rightarrow_B \rangle$, where a_e of type $\mathbb{W}_{64}$ is the entry point of the binary and fetch of type $\mathbb{W}_{64} \mapsto \mathbb{I}$ returns, given an address, *one* instruction. The behavior of the binary is modeled by some set of concrete states S and some black-box deterministic transition relation $\rightarrow_B$ over concrete states.

Definition 3.2. A Hoare Graph is defined as a tuple

$$\langle \Sigma, \sigma_I, \rightarrow_\Sigma \rangle$$

where symbolic states in $\Sigma = \mathbb{P} \times \mathbb{M}$ consist of predicates and memory models, $\sigma_I \in \Sigma$ is an initial symbolic state and $\rightarrow_\Sigma$ of type $\Sigma \times \mathbb{I} \times \Sigma \mapsto \mathbb{B}$ is a non-deterministic transition relation labeled with instructions.

The algorithm requires the definition of a *join* operation over symbolic states that soundly merges the information stored in two symbolic states. This is because the algorithm explores the state space by recursively adding new states and edges on the fly. Joining serves 1.) to prevent state space explosion and 2.) to ensure termination. If an instruction address is visited more than once, the *supremum* (result of joining) of all symbolic states associated with that address is computed until a least fixed point is reached.

We therefore define an algebraic *join-semilattice* over symbolic states. That is, we define our join operation such that it establishes a partial order over the symbolic states, allowing us to calculate a least upper bound state over any two symbolic states. This join-semilattice is depicted as the tuple $\langle \Sigma, \sqcup \rangle$, where Σ is the type of symbolic states and $\sqcup$ denotes the join operation. The desired partial ordering over Σ, $\sqsubseteq$, is then derived by defining $\sigma_0 \sqsubseteq \sigma_1$ as $\sigma_1 = \sigma_0 \sqcup \sigma_1$. Intuitively, $\sigma_0 \sqsubseteq \sigma_1$ denotes that σ_0 is "less abstract" then σ_1. The join must then satisfy the following soundness criterion for any concrete state s: $(s \vdash P \vee Q) \implies (s \vdash P \sqcup Q)$. The join must be sufficiently coarse to ensure that there exists no infinitely descending chain of symbolic states $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \cdots$. Since a symbolic state consists of a predicate and a memory model, we define a join for both.

3.1 Predicates

Predicates are assertions on state. A predicate consists of a set of *clauses*. A clause consists of two symbolic expressions and their relation. A predicate P holds in state s if and only if all clauses hold. An expression of type $\mathbb{E}$ consists of registers ($\mathbb{R}$), flags ($\mathbb{F}$), words, variables ($\mathbb{V}$), memory regions (modeled by an expression for the address and a natural number for the size) and the application of an operator to a list of expressions.

$$\mathbb{E} \coloneqq \mathbb{R} \mid \mathbb{F} \mid \mathbb{W} \mid \mathbb{V} \mid \mathbb{E} \times \mathbb{N} \mid \text{Op} \times [\mathbb{E}]$$

We identify a subset of these expressions called *constant expressions* ($\mathbb{C}$). These expressions cannot contain registers, flags, or memory regions. They represent constants or computations constructed using initial values. For example, $\texttt{rdi}_0$ is a variable denoting the initial value of register `rdi`.

Clauses are terms of the form $\mathbb{E} \square \mathbb{C}$, where $\square$ is an element of the following set of relations: $\{=, \neq, <, <_s, \geq, \geq_s\}$. The $\square_s$ relations treat their operands as signed, while their non-subscripted versions treat their operands as unsigned. There are two special predicates, $\top$ and $\bot$, that respectively indicate

being always true and always false. $\bot$ is also used to indicate an unknown $\mathbb{C}$.

Definition 3.3. The join of two predicates P and Q, notation $P \sqcup Q$, is provided by doing range abstraction for symbolic bit-vector values [48].

Example 3.4. Let $P = \{a = 3, a < \texttt{rdi}_0\}$ and $Q = \{a = 4, a < \texttt{rsi}_0\}$. As both predicates have equality clauses for a, those clauses are merged to produce a pair of clauses denoting that the value of a lies in the range $[3, 4]$. Since no maximum can be established between $\texttt{rdi}_0$ and $\texttt{rsi}_0$, these clauses are dropped. Thus, $P \sqcup Q = \{a \geq 3, a \leq 4\}$.

As required for a lattice, the join is associative, commutative and idempotent. Associativity is derived from the fact that set union and minimum/maximum are associative operations. The join is commutative and idempotent due to the commutativity and idempotency of the merge function. Finally, it satisfies that for any state s: $s \vdash P \vee Q \implies s \vdash P \sqcup Q$.

3.2 Memory Models

Program analysis in programs with pointers requires efficient alias identification and classification. Alias information directs assignments to memory. We thus keep track of the read and written memory regions in a structured memory model. These memory models store *aliasing*, *separation* and *enclosure* relations for memory regions. A memory model is defined by the following data structure:

$$\mathsf{MemTree} \coloneqq \{\mathbb{C} \times \mathbb{N}\} \times \mathsf{Mem} \qquad \mathsf{Mem} \coloneqq \{\mathsf{MemTree}\}$$

That is, a memory model consists of a possibly empty *forest* of memory trees. Each memory tree has as a top-level node, a set of memory regions and a possibly empty sub-forest that holds its children. Two regions in the same node are aliasing. Children are enclosed in their parents. Siblings are separate.

Example 3.5. Consider the two memory models presented in Figure 2. These memory models involve three regions: $[\texttt{rdi}_0, 8]$, $[\texttt{rsi}_0, 8]$ and $[\texttt{rsi}_0 + 4, 4]$. The memory models depict the case where $\texttt{rdi}_0$ and $\texttt{rsi}_0$ alias and not alias.

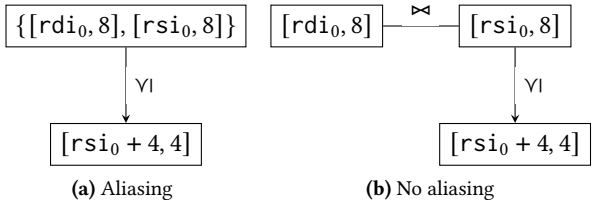
Definition 3.6. Let s be a concrete state and let $r_0 = \langle e_0, n_0 \rangle$ and $r_1 = \langle e_1, n_1 \rangle$ be two regions in memory. Respectively, *aliasing*, *separation* and *enclosure*, notations ($\equiv$, $\bowtie$, $\preceq$), are defined as:

$$r_0 \equiv r_1 \stackrel{\text{def}}{=} s \vdash e_0 = e_1 \wedge n_0 = n_1$$

$$r_0 \bowtie r_1 \stackrel{\text{def}}{=} s \vdash (e_0 + n_0 \leq e_1) \vee (e_1 + n_1 \leq e_0)$$

$$r_0 \preceq r_1 \stackrel{\text{def}}{=} s \vdash e_0 \geq e_1 \wedge e_0 + n_0 \leq e_1 + n_1$$

A relation holds *necessarily* if and only if it holds in all concrete states s. For example, $[\texttt{rsi}_0, 4] \bowtie [\texttt{rsi}_0 + 4, 4]$ denotes that the two regions are necessarily separate. The SMT solver/theorem prover Z3 [15] is used to establish whether these "necessarily"-relations hold for symbolic addresses given the current state predicate. This is done via expression translation directly to Z3's bit-vector representations, meaning no information is lost in the conversion and when querying the constructed logical formulas.



(a) Aliasing **(b)** No aliasing

Figure 2. Memory model examples. Siblings on the same level are separate, children are enclosed within their parents. Regions within the same node are aliasing. The one on the left shows a situation with two top-level regions aliasing and the child region they share. The one on the right shows a situation where the two top-level regions do not alias, and thus only one of those regions contains an enclosed child.

We further extend the above notation to memory trees, e.g., $t_0 \bowtie t_1$ denotes that all regions in t_0 are necessarily separate from all regions in t_1. Notation $t_0 \equiv t_1$ ($t_0 \preceq t_1$) denotes that some region in the top node of t_0 and some region in the top node of t_1 necessarily alias (enclosure).

Construction of a memory model is performed using the recursive *ins* function shown below. It takes as input a memory tree t and the current memory model M. The current predicate P is also supplied to assist in the region relationship analysis but is elided from the below presentation as it is a read-only value that is passed along through the function call chain. For output, function *ins* produces, nondeterministically, a set of new memory models based on all possible pointer relationships for the newly-inserted region. If no necessarily-relation can be established between t and any tree in M, then all trees possibly overlapping with t are destroyed (see Section 1). If a necessarily-relation can be established between tree t and some tree already in M, then only the relevant memory models need to be produced.

Definition 3.7. Let $t_0 = \langle R_0, M_0 \rangle$ and $t_1 = \langle R_1, M_1 \rangle$ be two trees. Function ins of type $\mathsf{MemTree} \times \mathsf{Mem} \times \mathsf{Pred} \rightarrow \{\mathsf{Mem}\}$ is defined as follows:

$$\mathrm{ins}(t_0, \varnothing) \stackrel{\text{def}}{=} \{t_0\}$$

$$\mathrm{ins}(t_0, t_1 : M) \stackrel{\text{def}}{=} \begin{cases} \mathrm{ins}_{\mathrm{AL}}(t_0, t_1, M) & \text{if } t_0 \equiv t_1 \\ \mathrm{ins}_{\mathrm{SEP}}(t_0, t_1, M) & \text{if } t_0 \bowtie t_1 \\ \mathrm{ins}_{\mathrm{ENC}}(t_0, t_1, M) & \text{if } t_0 \preceq t_1 \\ \mathrm{ins}_{\mathrm{CON}}(t_0, t_1, M) & \text{if } t_1 \preceq t_0 \\ \mathrm{destroy}(t_0, M) & \text{otherwise} \end{cases}$$

$$\mathrm{ins}_{\mathrm{AL}}(t_0, t_1, M) \stackrel{\text{def}}{=} \{(R_0 \cup R_1, M') : M \mid M' \in \mathrm{fold}(\mathrm{ins}, M_0 \cup M_1)\}$$
$$\mathrm{ins}_{\mathrm{SEP}}(t_0, t_1, M) \stackrel{\text{def}}{=} \{t_1 : M' \mid M' \in \mathrm{ins}(t_0, M)\}$$
$$\mathrm{ins}_{\mathrm{ENC}}(t_0, t_1, M) \stackrel{\text{def}}{=} \{\langle R_1, M' \rangle : M \mid M' \in \mathrm{ins}(t_0, M_1)\}$$
$$\mathrm{ins}_{\mathrm{CON}}(t_0, t_1, M) \stackrel{\text{def}}{=} \{\mathrm{ins}(t', M) \mid t' \in \{\langle R_0, M' \rangle \mid M' \in \mathrm{ins}(t_1, M_0)\}\}$$

Notation $a : X$ denotes $\{a\} \cup X$. Let $t_0 = \langle R_0, M_0 \rangle$ be the tree to be inserted and let $t_1 = \langle R_1, M_1 \rangle$ be a tree already in the memory model. If t_0 and t_1 alias, then they are combined by taking the union of their nodes. All subtrees are then reinserted using a fold. If trees t_0 and t_1 are separate, then tree t_0 is recursively inserted into the remainder of the memory model and t_1 is added without modification. If t_0 is enclosed in t_1, it is recursively inserted into the sub-forest of t_1. For each memory model M' thus obtained, a memory model is produced with a tree $\langle R_1, M' \rangle$. The remainder of memory model M is unmodified. Finally, if t_1 is enclosed in t_0, then t_1 is recursively inserted into the sub-forest of t_0. For each memory model M' thus obtained, a tree t' is produced that consists of $\langle R_0, M' \rangle$. That tree is then recursively inserted in memory model M.

Example 3.8. Consider the three-instruction assembly snippet below. This snippet first stores the value 1000 in the eight-byte memory region pointed to by `rdi`, then stores the value 1001 in the four-byte region pointed to by `rsi+4`. Finally, it stores the value 1002 in the eight-byte region pointed to be `rsi`. If the current state allows aliasing and separation between $[\mathtt{rdi}, 8]$ and $[\mathtt{rsi}, 8]$, then these three instructions will result in the two memory models in Figure 2. Note that region $[\mathtt{rsi} + 4, 4]$ is necessarily enclosed in region [rsi,8].

```
mov qword ptr [rdi], 1000
mov dword ptr [rsi+4], 1001
mov qword ptr [rsi], 1002
```

A memory model M *holds* in concrete state s if all siblings are separate and all trees hold. A tree holds if its node contains aliasing regions and all trees in its sub-forest are enclosed.

Definition 3.9. A memory model M *holds* in state s, notation $s \vdash M$, if and only if:

$$(\forall t_0, t_1 \in M \cdot t_0 \neq t_1 \implies s \vdash t_0 \bowtie t_1) \wedge (\forall t \in M \cdot s \vdash t)$$

A memory tree $t = \langle R, M \rangle$ *holds* in state s, notation $s \vdash t$, if and only if:

$$(\forall r_0, r_1 \in R \cdot s \vdash r_0 \equiv r_1) \wedge (\forall t' \in M \cdot s \vdash t' \sqsubseteq t) \wedge (s \vdash M)$$

Example 3.10. Consider again the memory models in Figure 2 for the assembly snippet in Example 3.8. The aliasing memory model in Figure 2a is only consistent in states where $\mathtt{rdi}_0 = \mathtt{rsi}_0$. Meanwhile, the non-aliasing memory model in Figure 2b is only consistent in states where $\mathtt{rdi}_0 + 8 \leq \mathtt{rsi}_0$ or $\mathtt{rsi}_0 + 8 \leq \mathtt{rdi}_0$.

The insertion function must be *complete*: the produced memory models should cover any possible relation between inserted region r and any region r' already present in the memory model. To formulate completeness, we use $R(M)$ to denote the set of regions in memory model M and $\mathcal{R}(M)$ to denote the set of relations. For example, we have $([\mathtt{rdi}_0, 8] \equiv [\mathtt{rsi}_0, 8]) \in \mathcal{R}(M)$ for the memory model in Figure 2a.

Lemma 3.11. *Let r_0 and M resp. be a region and a memory model. Let f of type $\mathbb{C} \times \mathbb{N} \mapsto \{\equiv, \bowtie, \sqsubseteq, , \sqsupseteq\}$ be some mapping that provides for any region r' currently in memory model M a relation between r_0 and r'. Assume that f is possibly true:*

$$\exists s \cdot s \vdash M \wedge (\forall r' \in R(M) \cdot s \models r_0 \; f(r') \; r')$$

Then, insertion of region r_0 into M will produce at least a corresponding memory model:

$$\exists M' \in \mathrm{ins}(\langle r_0, \varnothing \rangle, M) \cdot \{(r_0 \; f(r') \; r') \mid r' \in R(M)\} \subseteq \mathcal{R}(M')$$

In words, there exists some memory model that contains all relations of mapping f.

Proof. The proof is by induction. The base case is trivial. For the inductive case, we insert region r into $\{t_1\} \cup M$. Four cases are possible:

1. Region r_0 necessarily aliases with t_1. In this case, since mapping f is possibly true, it must assign $\equiv$ to all top-level regions of t_1, $\geq$ to all other regions in t_1 and $\bowtie$ to all regions in M. The created memory model contains these relations.
2. Region r_0 is necessarily separate from t_1. In this case, since mapping f is possibly true, it must assign $\bowtie$ to any region in t_1. Thus, tree t_1 is not modified and region r_0 is recursively inserted into M. The induction hypothesis (IH) then finishes the proof.
3. Region r_0 is necessarily enclosed by a top-level region of t_1. Since mapping f is possibly true, it must assign $\bowtie$ to all regions of M. Therefore, the insertion function does not modify M. Since r_0 and r_1 do not alias, the top-level regions R_1 of tree t_1 can remain unmodified as well. Region r_0 is recursively inserted into a child of t_1, proof follows form IH.
4. Tree r_1 is necessarily enclosed into region r_0. In this case, since mapping f is possibly true, it must assign $\geq$ to all regions of t_1. Therefore, tree t_1 is recursively inserted as subtree of r_0, producing a set of trees. For the remaining regions not in t_1, f can hold arbitrary relations. Therefore any t' in the produced set is recursively inserted into M. Again, the IH then finishes the proof.

□

Definition 3.12. The join of two memory models M_0 and M_1, notation $M_0 \sqcup M_1$, is recursively defined as:

$$M_0 \sqcup M_1 \stackrel{\text{def}}{=} \left\{ \mathrm{join}_{\mathrm{t}}(T) \;\middle|\; T \in {}^{M_0 \cup M_1}\!/_{\mathbf{C}^+} \right\}$$

$$\langle R_0, . \rangle \; \mathbf{C} \; \langle R_1, . \rangle \stackrel{\text{def}}{=} R_0 \cap R_1 \neq \varnothing$$

$$\mathrm{join}_{\mathrm{t}}(T) \stackrel{\text{def}}{=} \left\langle \bigcap \{R \mid \langle R, . \rangle \in T\}, \bigsqcup \{M \mid \langle ., M \rangle \in T\} \right\rangle$$

This operation partitions the memory trees in M_0 and M_1 based on equivalence relation $\mathbf{C}^+$. This equivalence relation is the transitive closure of relation $\mathbf{C}$, which determines if its two memory trees have any top-level regions in common. In other words, all memory trees that have one or more top-level regions in common are put in an equivalence class and are thus joinable. The function join_t then performs the join operation for each equivalence class of memory trees, taking the intersection of all their region sets and the supremum of their child memory models.

Example 3.13. Consider two memory models M_0 and M_1 both with as top node $[\texttt{rdi}_0, 8]$, where M_0 has an enclosed child $[\texttt{rdi}_0, 4]$ and M_1 has an enclosed child $[\texttt{rdi}_0 + 4, 4]$. The join of M_0 and M_1 is one memory model with as top node $[\texttt{rdi}_0, 8]$ and the two subsregions as separate sibling-children.

We prove that the join over memory models is sound.

Lemma 3.14. *Let s be a state and M_0 and M_1 be memory models. Then:*

$$(s \models M_0 \vee M_1) \implies (s \models M_0 \sqcup M_1)$$

Proof. Let $r_0 \square r_1$ be a relation in $\mathcal{R}(M_0 \sqcup M_1)$. If $\square$ is $\equiv$, then both regions r_0 and r_1 must have been present in all trees in the corresponding equivalence class. The relation thus held in either M_0 or M_1. If $\square$ is $\bowtie$, then the two regions are from trees generated from different equivalence classes. Since they are from trees that do not share a top-level region, the original trees in either M_0 or M_1 are separate as well. Similar reasoning applies for the other cases. □

Definition 3.15. The join of some two symbolic states $\sigma_0 = \langle P_0, M_0 \rangle$ and $\sigma_1 = \langle P_1, M_1 \rangle$, notation $\sigma_0 \sqcup \sigma_1$, is:

$$\sigma_0 \sqcup \sigma_1 \stackrel{\text{def}}{=} \langle P_0 \sqcup P_1, M_0 \sqcup M_1 \rangle$$

We would like to remark that this join loses information. It can thus only be applied in a sound fashion for postcondition weakening [24]. In other words, dropping clauses and performing state cleanup serve only to reduce state constraints; they never add additional ones. In practice, this loss of information means that we may produce a state that would not actually be encountered during program execution, or we may be unable to resolve some indirections/prove some return addresses (which would result in annotations/tool failure). Even in such cases, given successful completion and no annotations produced, we will always produce all states that would be encountered during concrete execution.

4 Algorithm

Algorithm 1 provides the base functionality of Hoare Graph extraction. This base functionality is extended in two ways: 1.) a context-free approach to function calls (see Section 4.2) and 2.) preventing states from being joined when they are incompatible (see below).

The algorithm maintains two objects. First, a *bag* of symbolic states that is to be explored. Function EXPLORE is repeatedly called until the bag is empty. Second, the current Hoare Graph *HG*.

The algorithm requires a function τ modeling instruction semantics. Our implementation supports a wide range of x86-64 instructions, including (conditional) moves and jumps as well as arithmetic, logical and bit-vector operations (sufficient to deal with all Xen binaries). Given a supported instruction and a suitable memory model, function τ transforms its supplied predicate into a set of predicates by symbolically executing the single instruction. The memory model allows τ to take into account information on pointer relations when performing symbolic execution using destination operands that reference memory locations.

The algorithm requires an expression evaluation function $\text{eval} : \mathbb{E} \times \mathbb{P} \mapsto \mathbb{C}$, which maps an expression (over registers, flags, and dereferenced memory regions) to a constant-expression. To this end, it uses the predicate of the current symbolic state. No memory model information is required for this evaluation; that information is only required when writing to a location in memory.

Definition 4.1. Given predicate P, the evaluation of an expression e is defined as follows:

$$\text{eval}(e, P) = \begin{cases} v & \text{if } e = v \text{ is a clause in } P \\ \bot & \text{otherwise} \end{cases}$$

The algorithm then executes steps according to the following step function:

Definition 4.2. The *symbolic state step function* for symbolic state $\sigma = \langle P, M \rangle$, notation $\text{step}_\Sigma(\sigma)$, is defined as:

$$\text{step}_\Sigma(\sigma) \stackrel{\text{def}}{=} \{\langle P', M' \rangle \mid P' \in \tau(P, M') \wedge M' \in \text{ins}(R, M)\}$$
where
$$R \stackrel{\text{def}}{=} \{[\text{eval}(a, P), s] \mid [a, s] \text{ used by instruction } i\} - \{\bot\}$$
$$i \stackrel{\text{def}}{=} \text{fetch}(\text{eval}(\texttt{rip}, P))$$

Given the current symbolic state σ, the set of next symbolic states is obtained by applying predicate transformation to the current predicate and by inserting regions into the current memory model. The set of regions is obtained by considering the operands of the current instruction. For example, the instruction `mov qword ptr [rax + 4*rdi], rax` results in one region $[\texttt{rax} + 4 * \texttt{rdi}, 8]$. That region is – given the current predicate – evaluated to a constant. For example, if the current predicate contains $\texttt{rax} = 0x100$ and $\texttt{rdi} = \texttt{rsi}_0$, then evaluation produces the constant $0x100 + 4 * \texttt{rsi}_0$. The evaluated region is inserted. If the current predicate does not contain sufficient information to evaluate the region, evaluation produces $\bot$ and the region is not inserted. The latter overapproximates any relation (e.g., aliasing, separation) the new region may have with the current memory model. Finally, if no bounded set of next states can be determined

(such as for unresolved indirect jumps or calls), we produce an annotation and stop further exploration from that state.

Definition 4.3. Two symbolic states σ and σ' are *compatible*, notation $\sigma_0 \cong \sigma_1$, if and only if their instruction pointers (`rip`) are equal.

States will only be joined when they are compatible. A second extension to the base algorithm modifies this definition so that states are not considered compatible when registers contain different immediate values that directly influence control flow (e.g., when the immediates are loaded from a jump table). In general, it is impossible to know whether a stored value will influence future control flow. However, it suffices to detect situations in which values will *likely* influence future control flow. If a value was erroneously deemed to influence future control flow, then we have unnecessarily explored paths that could have been joined earlier, but this only affects run-times and not soundness. If we join states that contain immediate values that turn out to be necessary to assess future control flow, this will lead to unresolved indirections or a verification error, but does not affect soundness. Concretely, if two states assign a certain state part with immediate pointers to instructions (i.e., pointers that fall in the range of text sections in the binary), then we do not join them to an abstract value but instead, continue exploration from both states. This is because these immediate values will highly likely influence future control flow. This causes less abstraction, and more preciseness in some very specific cases, but does allow us to resolve more indirections.

4.1 Base Algorithm

Algorithm 1 Base of Hoare Graph Extraction

```
1:  function EXPLORE
2:      pop σ from bag
3:      if ∃σ_c ∈ HG · σ_c ≅ σ then
4:          if σ ⊑ σ_c then return
5:          σ_j := σ ⊔ σ_c
6:          HG[σ_c := σ_j]
7:      else
8:          σ_j := σ
9:      end if
10:     for all σ' ∈ step_Σ(σ_j) do
11:         HG += (σ_j, σ')
12:         if eval(rip, pred(σ')) is not immediate then
13:             annotate, stop further exploration
14:         else
15:             bag += σ'
16:         end if
17:     end for
18: end function
```

Let σ be some symbolic state from the bag (Line 2). Function EXPLORE first searches for a current symbolic state σ_c already in the current *HG* that is compatible (Line 3). If such a state exists and it is more abstract (based on ordering $\sqsubseteq$) than state σ, no further exploration is necessary (Line 4). Otherwise, σ and σ_c are joined (Line 5). The *HG* is modified by replacing the current state with the joined one. This replacement maintains all current edges: only the state is modified. Symbolic state σ_j is the state to be explored further. If no compatible state exists in the current *HG*, then σ is the state to be explored further (Line 8). Exploration occurs at Lines 10 to 17. For every successor σ' (possibly none), an edge is added to the *HG*. If, for some successor evaluation of the instruction pointer, `rip` does not produce an immediate concrete value, then this is due to either 1.) a return statement (after which `rip` is set to the symbol pushed to the top of the stack in the initial state), or 2.) the current symbolic state does not provide sufficient information to resolve the computation of `rip` (because of an indirect branch, for example). In the second case, the state is annotated with an unsoundness warning (Line 13) and the algorithm terminates early. Otherwise, the successor is added to the bag.

Soundness. To formulate soundness and present a proof, we first define a relation **R** between the concrete transition system and the Hoare Graph. We then prove Lemma 4.5, which shows that this relation is a simulation. As a direct result of this lemma, any concrete path can be simulated by a path consisting of symbolic steps produced by function step_Σ.

Definition 4.4. A concrete state s is *related to* symbolic state $\sigma = \langle P, M\rangle$, notation $s\ \mathbf{R}\ \sigma$, if and only if:

$$s\ \mathbf{R}\ \sigma \stackrel{\text{def}}{=} (s \vdash P) \wedge (s \vdash M)$$

Lemma 4.5. *Assume that predicate transformation τ is correct:*

$$\forall s\ s' \cdot s \rightarrow_B s' \wedge (s \vdash P) \implies \exists Q \in \tau(P, M) \cdot s' \vdash Q$$

Then relation **R** *is a simulation between the concrete transition system and the transition system obtained by abstract step function* step_Σ*:*

$$\forall s\ s' \cdot s \rightarrow_B s' \wedge s\ \mathbf{R}\ \sigma \implies \exists \sigma' \in \text{step}_\Sigma(\sigma) \cdot s'\ \mathbf{R}\ \sigma'$$

Proof. Let s and σ be two related states. Hence $(s \vdash P) \wedge (s \vdash M)$. By correctness of τ, we obtain a predicate $Q \in \tau(P, M)$ such that $s' \vdash Q$. By Lemma 3.11 (completeness of the insertion function), the memory model that holds in state s' is generated. Since the step function overapproximates by taking any combination of predicates in $\tau(P, M)$ and generated memory models, there exists at least one symbolic state that is related to s'. □

Definition 4.6. Hoare Graph $H = \langle \Sigma, \sigma_I, \rightarrow_\Sigma\rangle$ is *sound* with respect to binary $B = \langle a_e, \text{fetch}, S, \rightarrow_B\rangle$, notation $\text{sound}(H, B)$, if and only if:

$$\text{sound}(H, B) \equiv \forall s_0 \rightarrow_B^* s \rightarrow_B s' \cdot \exists \sigma \rightarrow_\Sigma \sigma' \cdot s\ \mathbf{R}\ \sigma \wedge s'\ \mathbf{R}\ \sigma'$$

In words, for every reachable transition from s to s' in the binary, there must exist a related transition in the Hoare Graph.

Theorem 4.7. *Algorithm 1 constructs a sound Hoare Graph.*

Proof. The structure of the algorithm is close to a depth-first search (DFS). For that reason, the white-path lemma is used to prove soundness [10]. For a normal DFS, the white-path lemma states that the DFS will eventually explore some state s' if and only if there exists some state s currently in the bag *and* there exists a "white" path from s to s'. A key difference between Algorithm 1 and a normal DFS is that states are joined. For the sake of this proof, a state is therefore considered "white" if the current HG contains no compatible state that is equal or more abstract (under $\sqsubseteq$). We reformulate the white-path lemma as follows:

$$\begin{aligned}&\sup(\sigma') \text{ is explored} \iff\\ &\exists \sigma \in \mathit{bag} \cdot \exists \pi = [\sigma, \ldots, \sigma'] \cdot \mathsf{white}(\pi)\\ &\textbf{where}\\ &\sup(\sigma') \equiv \bigsqcup\{\sigma'' \mid \sigma'' \cong \sigma' \wedge \exists \pi = [\sigma, \ldots, \sigma''] \cdot \mathsf{white}(\pi)\}\end{aligned}$$

In words, $\sup(\sigma')$, the supremum of all compatible states that are currently reachable through white paths, is explored by the algorithm if and only if there exists a white path from some σ currently in the bag to σ'. Given this version of the white-path lemma, it directly follows that if the bag initially contains the initial state only:

$$\sup(\sigma') \text{ is explored} \iff \sigma' \text{ is reachable from } \sigma_0$$

Now let s be a reachable concrete state and s' be a successor. Lemma 4.5 shows that the path from s_0 to s can be simulated by a path of related symbolic states. Let σ be the symbolic state related to concrete state s, i.e., $s \mathbf{R} \sigma$. Since σ is reachable, $\sup(\sigma)$ is explored. We thus have $s \mathbf{R} \sigma \implies s \mathbf{R} \sup(\sigma)$. This is a direct implication of Lemma 3.14: since joining makes the states more abstract, it makes the set of related concrete states larger. Line 10 will then explore some state $\sigma' \in \mathrm{step}_\Sigma(\sigma_j)$. By Lemma 4.5, we have $s' \mathbf{R} \sigma'$. □

4.2 Extension: Function Calls

The base algorithm as presented in Algorithm 1 does not treat function calls as special instructions. This is unsatisfactory for two reasons: first, for *external* function calls, a function τ that transforms the predicate may not be available. External function calls are dynamically linked and thus the assembly instructions are not available during static analysis. Second, even though *internal* function calls theoretically pose no problem, simply unfolding every function call prevents scalability. We present an extension to the algorithm that treats internal function calls compositionally. That is, it ensures that each function is explored only once.

4.2.1 External Functions. The function name is matched against a list of hard-coded function names that are known to be terminating, such as `exit` and `stack_chk_fail`. In case of a terminating function, function step_Σ will produce the empty set, stopping further exploration from the current state. Otherwise, the function is some unknown external function. We make the assumption that this unknown function adheres to the 64-bit System V calling convention. Function step_Σ therefore modifies the current state by assigning $\bot$ to all registers, flags and heap regions currently in the state that may not be assumed to be preserved by a function call. In other words, only the clauses concerning the stack frame and callee-saved non-volatile registers are kept. Similarly, all relations in the memory model concerning the heap are removed. We call this *cleaning* the current symbolic state. As with the join operation, this usage is sound as the end result is always a weakening of the postcondition.

4.2.2 Internal Functions. If the operand of a `call` can be resolved to an address inside the executable range of the binary, it is recognized as an internal call. Consider the following assembly code:

Function Call	Return	Exit
`100: call 400`	`400: ...`	`400: ...`
`105: ...`	`450: ret`	`450: call exit`

Intuitively, exploration from address `0x100` can proceed both at addresses `0x400` (entering the function) and at `0x105` (after the function). The latter, however, may not safely be assumed, as it is not known whether the called function returns normally. A function may always exit, in which case address `0x105` is never visited. Other issues, such as buffer overflows, can prevent a normal return as well.

We therefore introduce the notion of *reachability*. Each symbolic state has a Boolean field that is set to true only if the state is known to be reachable. States in the bag whose reachability field is false are not selected. Line 3 of the algorithm becomes:

3: **if** $\exists \sigma_c \in HG \cdot \sigma_c \cong \sigma \wedge \mathsf{reachable}(\sigma_c)$ **then**

Moreover, after Line 15, any symbolic state with the same `rip` as the newly explored σ' is marked as reachable:

14: mark all $\sigma \in \mathit{bag}$ with $\mathtt{rip}(\sigma) = \mathtt{rip}(\sigma')$ as reachable

Secondly, we treat internal function calls as *context free*. In the example above, exploration of address `0x400` is done in a fresh empty symbolic state. In that state, instead of pushing the concrete return address `0x105`, a symbol $\mathcal{S}_{\mathtt{0x400}}$ is pushed. As a result, wherever the internal function is called, it will always start in the exact same state and therefore exploration happens only once.

A global mapping is maintained that remembers that symbol $\mathcal{S}_{\mathtt{0x400}}$ is linked to return address `0x105`. It may be the case that the internal function is called from different call sites, in which case the mapping is updated accordingly: one symbol may be mapped to multiple return addresses. As

soon as the instruction pointer is set to symbol $\mathcal{S}_{\texttt{0x400}}$ (e.g., by a `ret` instruction), all mapped return addresses are set to reachable.

5 Experimental Results

5.1 Hoare Graph Extraction

We have applied HG extraction to: 1.) several stripped binaries of CoreUtils as found in a standard Ubuntu distribution; 2.) a binary with a manually induced buffer overflow, confirming that no HG is extracted; and 3.) all 63 x86-64 binaries and all 2151 functions from the 25 shared objects we identified in the Xen Hypervisor.

All results are publicly available; we report here on the Xen case study. The Xen Project is a mature, industrial-strength hypervisor used in many production systems such as Amazon's cloud platforms [9]. Hypervisors provide a method for managing multiple virtual instances of operating systems (guests) on a physical host. Xen is a suitable case study because of its complexity and wide range of programs and shared libraries produced by its build process.

The analysis was performed on a machine running Linux Mint 20.1 Cinnamon with a 6-core, 2.9 GHz Intel Core i9-8950HK CPU. The machine had 31 GiB of RAM and 32 GiB of swap space on a KXG50PNV1T02 NVMe SSD. The version of Xen used was 4.12.

Table 1 shows an overview. The upper part of the table shows binaries. Lifting one binary means starting the extraction algorithm at the entry point and exploring all reachable assembly instructions in the binary, including internal function calls. The lower part shows library functions in a shared object. For every `.so` file, all externally exposed functions as reported by the `nm` utility are considered. Lifting one such function means starting the extraction algorithm at the function's address and exploring all reachable assembly instructions from that point, including calls to other internal functions.

Three issues may prevent lifting a binary to an HG, shown in the second column of Table 1.

Unprovable Return Addresses: as explained in Section 2, when a `ret` instruction is encountered, the current precondition must be sufficiently strong to prove that the return address at the top of the stack frame has not been modified. Moreover, the current precondition must show that the value of the stack pointer has been properly restored to its initial value. If the current precondition is not strong enough, the algorithm does not produce an HG since it cannot prove where control flow will lead to.

Concurrency: binaries that contain function calls to multithreading functions (such as `pthread_*`) are declared out of scope. We include them in Table 1 so that we account for all x86-64 Xen binaries.

Timeout: the timeout was set to 4 hours per binary/function.

In total, for 45 out of 63 binaries and 2115 out of 2151 library functions, the basic sanity properties (return address integrity, bounded control flow and calling convention adherence) could be proven and an HG could be generated.

The third and fourth columns of Table 1 show the number of instructions lifted out of the binary and the number of states of the HG. Taking both the binaries and shared objects into account, 399 771 instructions were lifted. Since states belonging to the same address are joined whenever compatible, the number of states is close to the number of instructions.

Column A shows the number of resolved indirections, i.e., the indirections where the effect of the instruction on the instruction pointer could be overapproximatively established. Columns B and C show the *annotations*, i.e., the numbers of unresolved indirect jumps and calls, respectively. Unresolved indirect calls are often caused by function callbacks: a function pointer is passed as a parameter (or through a global variable) from function to function. Programmer-supplied function arrays are another source of non-resolution. Since function calls are handled without context, the function pointer is unknown at the time it is actually called.

Figure 3 relates the sizes of functions (in numbers of instructions) to the verification time. The largest function successfully verified was `libxl_domain_suspend` from `libxenlight.so.4.12.0`, with 3925 instructions and 4207 symbolic states. The analysis took 49 minutes and 10 seconds to complete. The second-largest function verified, `libxl_domain_suspend_only`, had 3713 instructions with 4100 symbolic states and took 16 minutes 34 seconds to complete. The longest verification time was around 2 hours for function `libxl_domain_build_info_gen_json` with 1584 instructions. For the 1907 functions, we had 4 timeouts (not included in the 15:28 hrs of verification time). These functions generally had a large number of states that could not be joined (causing explosion in the number of states to be explored). Figure 3 shows that there is very little correlation between verification times and instruction count.

In total, we lifted an HG for 2115 out 2151 functions (98%). We can account for why this number is relatively high:

- For many functions, any pair of pointers *to the local stack frame* abided by any of the four relations for which we accurately model memory relations (aliasing, separations, enclosed within, encloses). As a result, even if the heap and the global memory space were grossly overapproximated, the local stack frame was modelled accurately and return address integrity could be proven.
- In case of an unresolved function call, we treated the function overapproximatively as an unknown external function. Typical reasons for unresolved indirections include callbacks: a function pointer is set by some function f and is retrieved and called back in function g. A context-sensitive approach would be able to increase the number of supported indirect calls, but this would need to be done sufficiently scalable.

Table 1. Xen Case Study Statistics Summary

Directory		Instrs.	Symbolic States	A	B	C	Time (h:m:s)
	Binaries						
. . . /bin	15 = 12 + 2 + 1 +0	6751	6829	21	19	0	0:15:54
. . . /xen/bin	17 = 7 + 1 + 8 +1	2433	2468	8	3	3	0:01:17
. . . /libexec	1 = 1 + 0 + 0 +0	82	87	1	0	0	0:00:10
. . . /sbin	30 = 25 + 1 + 4 +0	8858	9178	26	4	8	0:18:39
Total	63 = 45 + 3 +13+1	18 124	18 562	56	26	11	0:35:59
	Library functions						
. . . /lib	1907=1874+29+ 0 +4	353 433	362 635	1	244	600	15:28:17
. . . /xenfsimage	109 = 106 + 3 + 0 +0	17 184	17 683	0	0	27	1:58:36
. . . /dist-packages	16 = 16 + 0 + 0 +0	379	407	0	0	3	0:00:06
. . . /lowlevel	119 = 119 + 0 + 0 +0	10 651	10 799	0	0	90	0:08:43
Total	2151=2115+32+ 0 +4	381 647	391 524	1	244	720	17:35:42

$w + x + y + z$: w lifted, x unprovable return address, y concurrency, z timeout
A = Resolved indirection B = Unresolved jump(s) C = Unresolved call(s)

• Some of the rejections constitute functions that do not adhere to the calling convention. Manual analysis of these cases shows that these are all compiler-generated functions that are not required to follow a calling conventions.
• Other rejections were caused by a precondition insufficient to derive an overapproximative bounded set of concrete values for the next instruction pointer. This may occur when an array or struct is stored on the stack and accessed via variable offset. Such constructs may lead to complicated pointer arithmetic *within* the stack frame. The result is that the algorithm cannot prove that a memory region was separate from the top of the stack frame, storing the return address.
• Even though not all instructions of the x86 ISA are supported, all instructions occurring in the case study are, so this is not a reason why functions were rejected.

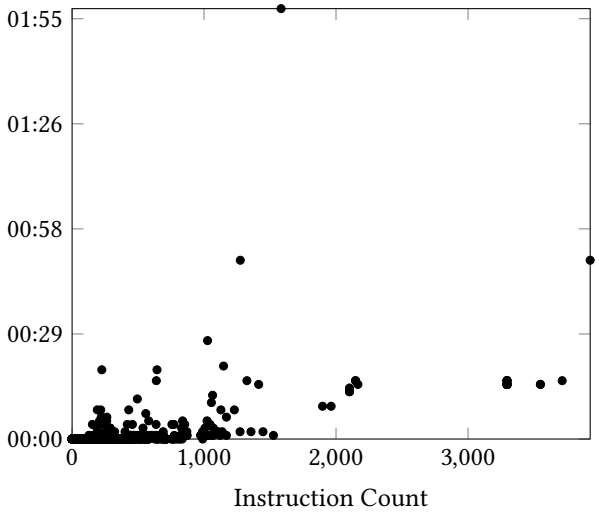


Figure 3. Verification times vs. instruction counts, sorted by instruction count.

5.2 Formal Proofs in Isabelle/HOL

For several CoreUtils binaries, we extracted an HG and exported it to the Isabelle/HOL theorem prover. The binaries are closed-source, taken from a standard MacOS 11.5.2 distribution. Table 2 provides an overview of the binaries, the number of instructions and Hoare triples, and the number of resolved indirections (there are no unresolved indirections). Without exception, all Hoare triples could be proven automatically.

We have developed a formal model of the semantics of roughly 120 different x86-64 assembly instructions. These instructions include various moves, arithmetic/logical operations, jumps, and call/return. Floating-point operations are mapped to uninterpreted functions. The model provides semantics for register aliasing and a byte-level little-endian memory model. Moreover, we have developped a symbolic execution engine that applies the formal semantics of an instruction to a symbolic state, and matches that to a given postcondition. This engine is based on a library of formally proven correct simplification theorems, as well as theorems that prove separation properties over different memory writes. Finally, we support automatic generation of implicit assumptions necessary for formal proofs. The informal algorithm can implicitly make assumptions that, e.g., regions in the global memory space are not overlapping with regions from the stack frame. A formal proof must explicitly assume that.

Table 2. Overview of binaries exported to Isabelle/HOL.

Binary	#Instructions	#Indirections
hexdump	2515	11
od	3040	11
wc	445	0
tar	5730	5
du	883	3
gzip	3465	7
Total	16 078	37

Effectively, each and any implicit assumption made during HG generation is formalized and exported to Isabelle/HOL.

5.3 Examples of Failures

Stack Overflow. ROP emporium (https://ropemporium.com) provides pedagogical examples that contain an exploitable stack overflow. For the `ret2win` example, the exploit simply amounts to ensuring that a call to `memset` writes 48 bytes to its given pointer. For our tool, `memset` is an unknown external function, and thus it is annotated with the assumptions needed to ensure return address integrity. The annotation states that:

$$\texttt{@400701} : \texttt{memset}(\texttt{RDI} := \texttt{RSP}_0 - 40)\ \texttt{MUST PRESERVE}\ [\texttt{RSP}_0 - 8\ \texttt{TO}\ \texttt{RSP}_0 + 8]$$

At address 0x400701, a call to memset occurs with as first parameter a pointer into the stack frame of the caller ($\texttt{RSP}_0 - 40$). The algorithm needed to assert that this call did not overwrite the memory region $[\texttt{RSP}_0 - 8\ \texttt{TO}\ \texttt{RSP}_0 + 8]$, where, among other things, the return address is stored. In other words, the algorithm asserted and noted as proof obligation that the write executed by `memset` did not exceed 32 bytes. In this example, the algorithm did not produce a verification error, but generated proof obligations that can be violated. Such a violation constituted an exploit candidate.

Stack Probing. In the binary `/usr/bin/zip` – as available in a standard MacOS distribution – a certain function executes the following function call:

```
100009fe6:  mov eax, 0x1400
100009feb:  call 0x10000a6a0
100009ff0:  sub rsp, rax
```

Register `rax` gets the value `0x1400`, then an internal function is called, and then the number of bytes in `rax` is allocated locally on the stack frame. The called function executes a compiler-generated technique called *stack probing*. That function traverses the stack and reads-then-discards individual bytes below the current stack pointer at intervals of `0x1000` bytes. The instruction at address `0x100009ff0` eventually causes a verification error, since the tool cannot establish whether register `rax` has been modified during that function call.

Non-standard Stackpointer Restoration. Normally, a function restores the stackpointer `rsp` to its initial value, plus eight due to the pushing of the return address. That is, after a `ret` statement the symbolic state is verified for:

$$\texttt{RSP} == \texttt{RSP}_0 + 8$$

In the binary `/usr/bin/ssh` – as available in a standard MacOS distribution – a function returns with the stackpointer `rsp` set to the following symbolic value:

$$\begin{aligned}&\texttt{RSP} == *[\\&\quad((\texttt{RSP}_0 - (48 - (((-4) - \texttt{R9}_0) * 8))) \,\&\, (-400))\\&\quad + ((\texttt{udiv64}(\texttt{R9}_0, 4) * 4) * 8) + 8\\&\quad] + 56\end{aligned}$$

This complicated symbolic value shows that the stackpointer is not normally restored, but instead read from a region in memory whose address is based on the initial value of register R9 (notation $*[a]$ denotes reading from address a). This function leads to a verication error, since as the stackpointer cannot be proven to be normally restored, no accurate memory relations over the local stack frame can be formulated.

6 Related Work

We relate our work to existing approaches for disassembly, binary decompilation, binary verification, and abstract interpretation [11]. To the best of our knowledge, the only existing work that has similar focus on disassembly based on formal methods is Jakstab [28–30], which we therefore discuss in detail.

Jakstab. Jakstab performs binary analysis and control flow reconstruction utilizing abstract interpretation. Jakstab's main analysis was designed for binaries with potentially handcrafted, obfuscated behavior (such as device drivers and malware). The two main differences are:

(1) Jakstab often requires usage of manually-coded harnesses for binaries. A harness provides property specification and additional intermediate operations not found in the actual binary, and possibly external call modeling. They may additionally be used to provide pointer initialization for library/driver code. For COTS binaries, such as the binaries of the Xen case study, that kind of harness is impossible to create precisely. An imprecise harness leads either to false positives requiring manual investigation, or to the necessity of unsound heuristics (page 168 of [28]).

(2) Additionally, we argue that Jakstab is not overapproximative. The instructions and states that are reached by Jakstab are *relative to the harness*; that is, relative to some initialization and external information. Jakstab reaches, on average, 15% of the instructions that are present in a binary (based on the results presented in Table 6.2 of [28]). This percentage is computed exclusively over the case studies where Jakstab reports a complete and successful result (no

counterexamples found for the property being checked). As Jakstab explores roughly 15% of the instructions in a binary when their analysis is reported as successful, we argue that it is not an overapproximative approach.

Disassembly. Disassemblers are tools that take a binary and lift it to an assembly language. Traditionally, there are two main methods of disassembly: *linear sweep* and *recursive traversal* [49]. Modern disassemblers may combine the two or use other techniques like probabilistic [36, 60, 61] or conflict analyses [6]. Linear sweep algorithms are easy to implement but well-known to be unsound [49]. Recursive traversal algorithms are more complex than linear sweep. They operate by starting from some initial instruction and then trace the possible paths of execution, interpreting instructions as they proceed [34, 49]. Our work is an example of a recursive traversal disassembler. This allows higher accuracy than linear sweep. The major challenge of recursive traversal is properly dealing with indirect calls and jumps. A tool that primarily uses recursive traversal is IDA [23], intended for interactive debugging and reverse engineering. Typically, existing approaches to recursive traversal use heuristics or guesses to approximate indirect branches.

Probabilistic approaches to disassembly [6, 36] include machine learning techniques, such as BYTEWEIGHT [4] or the works of Wartell et al. [60, 61]. In general, these techniques attempt to identify sequences of bytes as instructions based on their context, using large amounts of *training sets* as a guide. The main disadvantage of probabilistic/speculative techniques is that they inherently cannot provably overapproximate the behavior of the binary. Although they typically have very few cases of underapproximation, such cases are not impossible.

The key difference between our work and existing disassemblers is that: 1.) no existing disassembler aims at providing a guarantee that the lifted representation is a sound overapproximation of the binary; and 2.) our approach goes beyond disassembly, providing both control flow and invariants that are sufficiently strong enough to prove control flow. The cost of our approach is that it may fail, whereas other approaches are able to guess or use heuristics to continue.

Binary Decompilation. A decompiler takes a binary as input and lifts to a higher-level representation. Ghidra [43], RetDec [1], Phoenix [8] and FoxDec [56] all aim at lifting a binary to C code. SmartDec [19] lifts to C++ code, whereas McSema [16] lifts to LLVM. Ramblr [58] lifts a binary to *symbolized* assembly, where concrete addresses are replaced with symbolic labels. CodeSurfer/x86 [2, 3] provides a graphical interface for lifting binaries to an intermediate representation and interactively analyzing them. Decompilers are often integrated into reverse engineering and program exploration tools such as IDA Pro, Binary Ninja [62] and Ghidra.

A key factor in decompilation approaches, and also in other approaches that aim at producing control flow graphs or dataflow analyses, is that many of them assume that disassembly has been done by an external tool that is assumed to be sound. Our algorithm thus *complements* these works.

Decompilation-into-Logic (DiL) [38–40], uses operational semantics of machine code to lift binaries into a *functional* representation in Higher Order Logic. It is a technique that can be used in formal verification contexts. DiL, does not deal with indirect branching and *assumes* that return addresses are not overwritten. In their own words, their "heuristic is easily confused by computed branches" [37].

Binary Verification. Binary verification techniques aim at proving properties on the machine code level [35]. Typically, binary verification aims at proving that the binary is correct with respect to some higher-level artifact (source code or a specification). Sewell et al. used a refinement-based approach to verify the binary of the seL4 microkernel [31, 32, 50]. Kamkin et al. developed a methodology for verifying that the machine code of RISC-V binaries satisfy annotations in the binaries' source code [26]. For a top-down approach, proof-carrying code [41] integrates a proof into the binary that is verified at runtime.

In contrast, our approach aims at a context where a higher-level artifact such as source code or a specification is not available. In such contexts, most approaches are interactive [20, 21, 54]. Verbeek et al. provided a binary verification methodology tailored to memory preservation properties, with a "manual effort vs. instruction count ratio" of roughly 1 to 11 [55]. Tan et al. provide a fully automated method, AUSPICE, that takes about 6 hours for a 533-instruction string search algorithm [53]. Our approach *complements* formal verification that generate invariants for proving functional correctness. Our approach aims at removing the lifting process from the trusted code base.

Abstract Interpretation. The general problem of lifting binary code from a string of 0s and 1s to a higher-level form is known to be undecidable. Approximate solutions have been extensively studied, especially in the framework of abstract interpretation, which gives mathematical foundations to reason about approximations and their computations. Bardin et al. provide Binsec that uses, among others, abstract interpretation for information flow analysis in cryptographic implementations [5, 12]. Zhang et al. provide a path sampling algorithm and use abstract interpretation to prune infeasible paths. Reinbacher et al. use abstract interpretation in similar fashion for binary-level test case generation [45]. These works aim at orthogonal usages of abstract interpretation with respect to this paper, and *assume* availability of a tool that provides overapproximative control flow. Our approach is thus highly complementary to all these works.

BinTrimmer, developed by Redini et al., use abstract interpretation to refine CFGs from binaries for debloating [44]. They deal with indirect branches in an overapproximative fashion. A key difference is that BinTrimmer – in contrast to our work – is *solely* focused on indirections, i.e., is not

concerned with proving that a return address is not overwritten, with stack overflows, jump-in-the-middle behavior, or providing invariants sufficiently strong to serve as evidence for overapproximation. Moreover, BinTrimmers' case studies are six hand-picked binaries containing up to 555 LoC, in contrast to our Xen Hypervisor case study.

7 Discussion

The approach taken in this paper necessarily makes assumptions (see Section 1). We provide here a high-level discussion on how the assumptions affect the usability of overapproximative binary lifting in various application domains.

Security Analysis The central claim in this paper is that *if* all assumptions and proof obligations are met, *then* the lifted representation is a sound overapproximation of the binary. Section 5.3 shows an example where an assumption can be violated: `memset` may not preserve the indicated region. The negation of assumptions required for "normal" behavior may lead to "weird" behavior. In other words, the negation of the generated assumptions may be useful in the generation of exploits. A key challenge here is to filter out the relevant (exploitable) assumptions from the irrelevant ones.

Binary Verification We argue that the majority of existing work on binary verification *assumes* the existence of a trustworthy disassembler. This work exposes and makes explicit assumptions that otherwise may remain implicit. We argue that basing a verification effort on an a verified HG instead of on the output of any of-the-shelf disassembler reduces the trusted code base of the verification effort.

Decompilation Similarly, we argue that the majority of existing decompilation tools *assume* the existence of a reliable disassembler. A verified HG is a reliable base for decompilation. For example, the provably correct assembly and control flow inferred by our approach could be the input to McSema [16], in order to produce provably correct LLVM code. The assumptions then may be translated to higher-level `assert`-statements: the decompiled code is correct as long as no assert is triggered.

Patching Binary patching typically either involves some stages of decompilation, or replacing snippets of assembly instructions with different ones [17]. We argue that lifting both an original binary and its patched version to HGs would increase the trustworthiness of the patch effort. Both the HGs – but also the assumptions required for lifting the binaries – could be mutually compared, and this comparison may expose unexpected effects of the patch.

8 Conclusions

This paper presents the first *provably overapproximative* lifting mechanism for x86-64 binaries. Any overapproximative representation of a binary must include both all its "normal" as well as all its "weird" behaviors. A method is proposed that takes a stripped binary as input (no debugging information or address labeling is required). It produces a Hoare Graph as output that contains: 1.) the assembly instructions found in the binary; 2.) the control flow; and 3.) evidence, in the form of inductive invariants that are sufficiently strong to prove soundness. Our approach can deal with overlapping instructions and aims at providing overapproximative bounds to indirect branches (e.g., when a `jmp` is based on a computation instead of on a constant). In some cases, unsoundness annotations are used to indicate possible issues. Also, assumptions are enumerated explicitly in the form of proof obligations asserting requirements over external functions. If our technique succeeds and the proof obligations are proven true, then under these assumptions, the lifted representation is a provable overapproximation of the binary. We have applied our approach to binaries and shared objects of the Xen Hypervisor, covering 399 771 instructions in total. This case study shows that our methodology is scalable and applicable to commercial off-the-shelf software written without verification in mind. The Hoare Graph can be exported to the Isabelle/HOL theorem prover, where it can be formally verified. This second step essentially validates any inference made by the algorithms during Step 1.

In future work, we aim to provide support for concurrency. Moreover, we find that the context-free nature of our approach limits the number of function callbacks that are properly dealt with. We will study passing around statefull information between functions to find a midpoint between scalability and better support for function callbacks.

Finally, we aim to combine the lifted Hoare Graphs with existing approaches to binary analysis. Provably sound binary lifting can be the base for *any* trustworthy binary-level technique, including decompilation, binary verification and binary patching.

Availability

We provide the complete source code of our implementation and the case study results at:

https://doi.org/10.5281/zenodo.6330573

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Laure Gonnord for their insightful comments and suggestions, which helped to greatly improve the paper. Zhoulai Fu's work on this paper was partly done while he was at the IT University of Copenhagen, Denmark. This work is supported by the Defense Advanced Research Projects Agency (DARPA) under contract N6600121C4028 and Agreement No. HR.00112090028, the US Office of Naval Research (ONR) under grant N00014-17-1-2297, NAVSEA/NEEC under grant N00174-16-C-0018, and US Naval Surface Warfare Center Dahlgren Division under grant N00174-20-1-0009.

References

[1] Avast Software. [n.d.]. RetDec :: Home. https://retdec.com/ Accessed 2020-07-31.

[2] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86—A Platform for Analyzing x86 Executables. In *Compiler Construction*, Rastislav Bodik (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 250–254. https://doi.org/10.1007/978-3-540-31985-6_19

[3] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–23. https://doi.org/10.1007/978-3-540-24723-4_2

[4] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *23rd USENIX Security Symposium (USENIX Security 14)*. 845–860.

[5] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. 2011. The BINCOA framework for binary code analysis. In *International Conference on Computer Aided Verification*. Springer, 165–170. https://doi.org/10.1007/978-3-642-22110-1_13

[6] Mohamed Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2016. Speculative Disassembly of Binary Code. In *Proceedings of the 2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. 1–10. https://doi.org/10.1145/2968455.2968505

[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *International Conference on Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 463–469. https://doi.org/10.1007/978-3-642-22110-1_37

[8] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*. 353–368.

[9] David Chisnall. 2008. *The Definitive Guide to the Xen Hypervisor*. Pearson Education.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[11] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL 77)*. ACM, 238–252. https://doi.org/10.1145/234528.234740

[12] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1021–1038. https://doi.org/10.1109/SP40000.2020.00074

[13] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2009. Dynamic Integrity Measurement and Attestation: Towards Defense against Return-Oriented Programming Attacks. In *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing* (Chicago, Illinois, USA) *(STC '09)*. Association for Computing Machinery, New York, NY, USA, 49–54. https://doi.org/10.1145/1655108.1655117

[14] Jeremy Dawson. 2009. Isabelle Theories for Machine Words. *Electronic Notes in Theoretical Computer Science* 250, 1 (2009), 55–70. https://doi.org/10.1016/j.entcs.2009.08.005

[15] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[16] Artem Dinaburg and Andrew Ruef. 2014. McSema: Static translation of x86 instructions to LLVM. In *ReCon 2014 Conference, Montreal, Canada*.

[17] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 151–163. https://doi.org/10.1145/3385412.3385972

[18] Thomas F Dullien. 2017. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE Transactions on Emerging Topics in Computing* (2017). https://doi.org/10.1109/TETC.2017.2785299

[19] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: Approaching C++ Decompilation. In *2011 18th Working Conference on Reverse Engineering*. 347–356. https://doi.org/10.1109/WCRE.2011.49

[20] Shilpi Goel. 2016. *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. Ph.D. Dissertation. The University of Texas at Austin. http://hdl.handle.net/2152/46437

[21] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. 2014. Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls. In *Proceedings of the 2014 Formal Methods in Computer-Aided Design (FMCAD)*. 91–98. https://doi.org/10.1109/FMCAD.2014.6987600

[22] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 237–250. https://doi.org/10.1145/2908080.2908121

[23] Hex-Rays SA. 2020. IDA Pro – Hex Rays. https://www.hex-rays.com/products/ida/ Accessed 2020-07-06.

[24] Charles Antony Richard Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

[25] R. Nigel Horspool and Nenad Marovac. 1980. An approach to the problem of detranslation of computer programs. *Comput. J.* 23, 3 (1980), 223–229. https://doi.org/10.1093/comjnl/23.3.223

[26] Alexander Kamkin, Alexey Khoroshilov, Artem Kotsynyak, and Pavel Putro. 2020. Deductive Binary Code Verification Against Source-Code-Level Specifications. In *Tests and Proofs*, Wolfgang Ahrendt and Heike Wehrheim (Eds.). Springer International Publishing, Cham, 43–58. https://doi.org/10.1007/978-3-030-50995-8_3

[27] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2017. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 412–424. https://doi.org/10.1145/3134600.3134627

[28] Johannes Kinder. 2010. *Static Analysis of x86 Executables*. Ph.D. Dissertation. Technische Universität, Darmstadt. http://tuprints.ulb.tu-darmstadt.de/2338/

[29] Johannes Kinder. 2012. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *2012 19th Working Conference on Reverse Engineering*. 61–70. https://doi.org/10.1109/WCRE.2012.16

[30] Johannes Kinder and Dmitry Kravchenko. 2012. Alternating Control Flow Reconstruction. In *Verification, Model Checking, and Abstract Interpretation*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 267–282. https://doi.org/10.1007/978-3-642-27940-9_18

[31] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 207–220. https://doi.org/10.1145/1629575.1629596

[32] Gerwin Klein, Thomas Sewell, and Simon Winwood. 2010. Refinement in the Formal Verification of the seL4 Microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, David S. Hardin (Ed.). Springer Science+Business Media, LLC, Boston, MA, 323–339. https://doi.org/10.1007/978-1-4419-1539-9_11

[33] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2005. Automating Mimicry Attacks Using Static Binary Analysis. In *USENIX Security Symposium*, Vol. 14. 11–11.

[34] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *USENIX Security Symposium*, Vol. 13. 18–18.

[35] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. 2018. Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB. In *International Conference on Interactive Theorem Proving*. Springer, 362–369. https://doi.org/10.1007/978-3-319-94821-8_21

[36] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1187–1198. https://doi.org/10.1109/ICSE.2019.00121

[37] Magnus O Myreen, Michael JC Gordon, and Konrad Slind. 2008. Machine-code verification for multiple architectures – an application of Decompilation into Logic. In *2008 Formal Methods in Computer-Aided Design*. IEEE, 1–8. https://doi.org/10.1109/FMCAD.2008.ECP.24

[38] Magnus O. Myreen and Michael J. C. Gordon. 2007. Hoare Logic for Realistically Modelled Machine Code. In *Tools and Algorithms for the Construction and Analysis of Systems*, Orna Grumberg and Michael Huth (Eds.). Springer-Verlag, Berlin, Heidelberg, 568–582. https://doi.org/10.1007/978-3-540-71209-1_44

[39] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. 2008. Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic. In *2008 Formal Methods in Computer-Aided Design* (Portland, OR, USA). IEEE, 1–8. https://doi.org/10.1109/FMCAD.2008.ECP.24

[40] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. 2012. Decompilation into Logic – Improved. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 78–81.

[41] George C. Necula. 1997. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 106–119. https://doi.org/10.1145/263699.263712

[42] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. Springer Science & Business Media. https://doi.org/10.1007/3-540-45949-9

[43] NSA. 2019. Ghidra. https://ghidra-sre.org/ Accessed 2020-07-06.

[44] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards static binary debloating through abstract interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 482–501. https://doi.org/10.1007/978-3-030-22038-9_23

[45] Thomas Reinbacher, Jörg Brauer, Martin Horauer, Andreas Steininger, and Stefan Kowalewski. 2011. Test-case generation for embedded binary code using abstract interpretation. In *Sixth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10)–Selected Papers*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/OASIcs.MEMICS.2010.101

[46] Henry Gordon Rice. 1953. Classes of Recursively Enumerable Sets and their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[47] Ian Roessle, Freek Verbeek, and Binoy Ravindran. 2019. Formally Verified Big Step Semantics out of x86-64 Binaries. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. ACM, New York, NY, USA, 181–195. https://doi.org/10.1145/3293880.3294102

[48] Radu Rugina and Martin Rinard. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Sigplan Notices* 35, 5 (2000), 182–195. https://doi.org/10.1145/358438.349325

[49] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. 2002. Disassembly of Executable Code Revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering*. 45–54. https://doi.org/10.1109/WCRE.2002.1173063

[50] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 471–482. https://doi.org/10.1145/2491956.2462183

[51] Rebecca Shapiro, Sergey Bratus, and Sean W Smith. 2013. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*.

[52] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper*. Hyderabad, India. https://doi.org/10.1007/978-3-540-89862-7_1

[53] Jiaqi Tan, Hui Jun Tay, Rajeev Gandhi, and Priya Narasimhan. 2015. AUSPICE: Automatic Safety Property Verification for Unmodified Executables. In *VSSTE*. Springer, 202–222. https://doi.org/10.1007/978-3-319-29613-5_12

[54] Freek Verbeek, Joshua Bockenek, Abhijith Bharadwaj, Binoy Ravindran, and Ian Roessle. 2019. Establishing a refinement relation between binaries and abstract code. In *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*. 1–5. https://doi.org/10.1145/3359986.3361215

[55] Freek Verbeek, Joshua Bockenek, and Binoy Ravindran. 2020. Highly Automated Formal Proofs over Memory Usage of Assembly Code. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020)*. 98–117. https://doi.org/10.1007/978-3-030-45237-7_6

[56] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. 2020. Sound C Code Decompilation for a subset of x86-64 Binaries. In *Proceedings of the 18th International Conference on Software Engineering and Formal Methods* (Amsterdam, The Netherlands) *(SEFM 2020)*. https://ssrg-vt.github.io/FoxDec/

[57] Fish Wang and Yan Shoshitaishvili. 2017. Angr – The Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9. https://doi.org/10.1109/SecDev.2017.14

[58] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17)*. https://doi.org/10.14722/ndss.2017.23225

[59] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 157–168. https://doi.org/10.1145/2382196.2382216

[60] Richard Wartell, Yan Zhou, Kevin W. Hamlen, and Murat Kantarcioglu. 2014. Shingled Graph Disassembly: Finding the Undecideable Path. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 273–285. https://doi.org/10.1007/978-3-319-06608-0_23

[61] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating Code from Data in x86 Binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 522–536.

[62] Jordan Wiens. 2020. Binary Ninja. https://binary.ninja/2020/05/11/decompiler-stable-release.html Accessed 2020-07-06.