2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Formally Verified Binary-level Pointer Analysis

Freek Verbeek
Open Universiteit & Virginia Tech
Heerlen, The Netherlands & Blacksburg, USA
freek.verbeek@ou.nl

Ali Shokri
Virginia Tech
Blacksburg, USA
ashokri@vt.edu

Daniel Engel
Open Universiteit
Heerlen, The Netherlands
daniel.engel@ou.nl

Binoy Ravindran
Virginia Tech
Blacksburg, USA
binoy@vt.edu

***Abstract*—Binary-level pointer analysis can be of use in symbolic execution, testing, verification, and decompilation of software binaries. In various such contexts, it is crucial that the result is trustworthy, i.e., it can be formally established that the pointer designations are overapproximative. This paper presents an approach to formally proven correct binary-level pointer analysis. A salient property of our approach is that it first generically considers what proof obligations a generic abstract domain for pointer analysis must satisfy. This allows easy instantiation of different domains, varying in precision, while preserving the correctness of the analysis. In the trade-off between scalability and precision, such customization allows "meaningful" precision (sufficiently precise to ensure basic sanity properties, such as that relevant parts of the stack frame are not overwritten during function execution) while also allowing coarse analysis when pointer computations have become too obfuscated during compilation for sound and accurate bounds analysis. We experiment with three different abstract domains with high, medium, and low precision. Evaluation shows that our approach is able to derive designations for memory writes soundly in COTS binaries, in a context-sensitive interprocedural fashion.**

***Keywords*—binary analysis, pointer analysis, formal methods**

I. Introduction

Pointer analysis is central to various forms of verification and analysis for software containing pointers, facilitating the construction of a state-based semantic model of software [10], [30], [1], [17], [38], [42]. It aims to statically resolve, for any pointer in a given program, which region of the memory it may point to. Specifically, given any two pointers, it must be known whether they are *aliasing*, always referring to *separate* regions in memory, or if they may possibly overlap. If a value is written to memory, and no pointer information is known, then one cannot accurately describe what the next state will be. This can lead to overapproximative *thrashing* parts of the memory state or *forking*, i.e., to conservatively considering both separation and aliasing possibilities. Both such cases are undesirable as they will quickly lead to unrealistic states and path explosions. In other words, pointer analysis is a necessity for building a state-based transition system that accurately models the software under investigation. Such a model, then, typically precedes a verification or analysis effort aimed at higher-level properties.

The necessity for pointer analysis immensely exacerbates when dealing with binaries (i.e., machine or assembly code) instead of source code. The reason is that at this level of abstraction *everything is a pointer*. There are no variables, and memory can be considered as a flat unstructured address space. In a typical x86-64 program, about 28% of all assembly instructions write to memory[1], producing tens of thousands of pointers even in medium-sized programs. Moreover, control-flow related information is stored in writable memory, such as the current return address and the currently caught exception stack. Theoretically, if the destination of even a single memory write cannot be resolved, then the effect of executing that memory write could result in either thrashing all memory (including such control-flow pertinent information) or forking into unrealistic states, such as when a memory write overlaps with the return address, even though in reality it did not. This may lead to a situation where it cannot even be established what instruction is to be executed, let alone what an accurate next state can be. This is one of the key challenges in dealing with binaries, preventing one from simply using techniques developed for source code analysis and applying them to low-level code found in binary executables [15].

This paper presents an approach to formally verified binary-level pointer analysis. Typically, such analyses are based on a form of abstract interpretation [12], where an abstract domain is defined that overapproximates concrete semantics [5], [35]. A fundamental challenge is choosing the "right" abstract domain, as this essentially boils down to balancing precision vs. scalability. This paper thus first leaves the abstract domain *polymorphic* and formulates a set of eight generic to-be-refined functions, as well as the proof obligations that these functions must satisfy. Over these generic functions, an executable algorithm for pointer analysis is formalized and proven correct. An *instantiation* thus defines an abstract domain and an implementation of the generic functions. Any instantiation that satisfies the proof obligations automatically constitutes a formally proven correct binary-level pointer analysis. As will be discussed later, all the formalism and proofs are carried out in Isabelle/HOL and have been shared with the readers.

We then provide three different instantiations of our generic functions, each of which strikes a different balance in the trade-off between precision vs. scalability. First, *pointer computations* form an abstract domain that keeps track of how pointers were computed: highly precise, but in practice one must cap the domain to a given size and produce top ($\top$) when that cap is exceeded. Second, *pointer bases* form an abstract domain where each pointer is represented only by its pointer base (e.g., a stack pointer, or the return value of a `malloc`): more coarse as it cannot be used for alias analysis, but still allows accurate separation analysis (for binaries compiled from source code, pointers based in different blocks can be assumed to be

[1]Measured over several CoreUtils binaries and Firefox libraries with different levels of optimization.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00231

separate, even if an out-of-bounds occurs [31]). Third, *pointer sources* are an abstract domain where a pointer is modeled by the set of sources (e.g., user inputs, initial parameter values) used in its computation. This is highly coarse, but is scalable and still allows a form of separation reasoning.

The pointer analysis presented in this paper is context-sensitive and compositional. Context-sensitivity is desirable, since pointers are passed through from function to function. We derive *function preconditions*, which states that invariably a certain function is always called within in a context where, e.g., register `rdi` contains a heap-pointer, or register `rsi` contains a pointer to within the stack frame of the caller. Per function, we can then derive a *function postcondition* that summarizes which regions were written to/read from by the function, and if after termination abstract pointers are left in return registers or global variables. Compositionally, these summaries can then be used for pointer analysis for callers of summarized functions. Due to space limitation, we focus our presentation on intraprocedural analysis and do not expand on the above technique for composition.

Bottom-up pointer analysis (i.e., binary analysis, in contrast to top-down source code analysis) can be useful in various use cases (see Section VI):

- It can be integrated into a disassembly algorithm [37]. A large facet of disassembly is assessing which instruction addresses are reachable (i.e., control flow recovery). A key challenge is resolving indirections, i.e., dynamically computed control flow transfers. Context-sensitive pointer analysis can assist, by providing information on which pointers are passed to a function.
- It can be a preliminary step to a decompilation effort [11]. Specifically, one of the steps in decompilation is to recover variables. Bottom-up pointer analysis provides information on which memory writes actually constitute variables.
- It can be the base of a bottom-up dataflow analysis. The function summaries already provide a form of dataflow analysis, by providing information on in- and output relations. They can be used to verify whether functions adhere to a calling convention and to see which state parts are overwritten or preserved by a given function. We also demonstrate by example that an overapproximative pointer analysis can be used for live variable analysis.

We emphasize the need for formally proven correct binary-level pointer analysis. Symbolic treatment of pointers and memory is notoriously difficult. Existing approaches typically make various assumptions *implicitly*, e.g., they may implicitly assume a return address cannot be overwritten, assume separation between pointers based on heuristics or best practices, or assume alignment of regions. This paper aims to reduce the trusted code base by explicitizing such assumptions and either proving them through invariants or reporting them explicitly otherwise. The trusted code base is thus reduced to the validity of explicit and configurable assumptions such as "regions based on stack pointers of different functions are assumed to be separate".

Limitations, assumptions and scope. A major assumption behind our approach is the treatment of *partially overlapping* memory accesses. Memory accesses (reads or writes) to partially overlapping regions may happen, but we assume that they do not concern pointers. More details can be found in Section III-A. Moreover, our approach has been implemented for the x86-64 architecture and does not deal with concurrency. A fundamental limitation is that not all indirections may be resolved, which may lead to unexplored paths.

The approach has been formally proven correct in Isabelle/HOL [36], [14], and has been mirrored in Haskell for experimental results. These confirm soundness relative to a ground truth obtained by observing executions. Moreover, they show precision comparable to or improved upon the state-of-the-art. We evaluate the effect of interprocedural bottom-up pointer analysis with respect to resolving indirections, identifying 135 cases where context-sensitive information allowed resolving a function callback. Finally, for all analyzed functions it has been verified whether the result is sufficiently precise to show that the return address has not been overwritten and that critical parts of the stack frame (e.g., storing non-volatile register values) are unmodified during execution of the function. This was successful for 99.6% of all analyzed functions.

In summary, we contribute:

- A *formally proven correct* approach to binary-level pointer analysis that leaves the abstract domain *generic*, allowing easy development of instantiations with different characteristics (e.g., different levels of preciseness);
- An evaluation over roughly 1.4 million assembly instructions, showing scalability and applicability of the approach.

Section II studies the related work. Sections III and IV provide details of our generic functions and their three different levels of instantiation, respectively. Section V, demonstrates the realization of interprocedural pointer analysis through our general functions. While Section VI looks at several use cases of the introduced approach, Section VII relates the experimental results to those produced by the state-of-the-art tools currently available. Section VIII concludes the paper.

II. Related Work

Source-level pointer analysis has been an active research field for decades [22]. Its typical use cases lie in *top-down* contexts: it takes as input source code, and provides information to the compiler for doing optimizations and data flow analyses. We here do not aim to provide an overview of this field, since our work focuses on *bottom-up* contexts: taking as input a binary, the result of pointer analysis provides information usable for decompilation and verification.

Generally, source-level approaches to program analysis cannot directly be applied to binary-level programs [6]. Various research therefore focuses on symbolic execution and/or abstract interpretation specifically tailored to the binary level. We distinguish *under*approximative techniques from *over*approximative ones. In this discussion, we specifically focus on how these techniques deal with pointers.

Underapproximative approaches:
SAGE combines symbolic execution of assembly code with fuzz testing, allowing exposure of real-life vulnerabilities in real-life software [21]. Initially, SAGE did "not reason about symbolic pointer dereferences", but it has been combined with Yogi allowing runtime behavior observed from test cases to

be used in refining abstractions to find whether pointers may be possibly aliasing [20]. S^{2}E is a platform for traversing binaries, allowing exploration of hundreds of thousands of paths using selective symbolic execution [9], [8]. S^{2}E provides an approach where dereferencing a symbolic pointer provides next states with possible concrete values based on the symbolic pointer and the current path constraints. A similar approach is taken by FUZZBALL, a symbolic execution framework for binaries, mainly concerned with improving the path coverage of binary fuzzers [3], [33]. It explores individual paths one by one and chooses concrete values for offsets in pointers. BINSEC is a code analysis tool with a focus on the security properties of binaries [16], [13]. It has been applied to find use-after-free bugs [18] and for reachability analyses [19]. Symbolic execution is based on forking, using an SMT solver to prune infeasible paths. BINSEC is based on *bounded* verification, making it underapproximative. Kapus et al. provide an interesting approach by concretizing and segmenting the memory model so that symbolic pointers can only refer to single memory segments [24]. Use of a test harness ensures both termination and that allocations always have a concrete size.

These methods underapproximate either by not exploring all paths, or by underapproximating pointer values. Underapproximation typically is very well suited for finding bugs and vulnerabilities in software: it leads to few false negatives and provides excellent scalability. It is suitable in the context of testing and binary exploration. In contrast, our pointer analysis is overapproximative, quantifying over all execution paths and all values. This makes it suitable in the contexts of verification and lifting to higher-level representations.

Overapproximative approaches:

Control Flow Reconstruction. Overapproximative approaches to binary analysis are generally based on a form of abstract interpretation. Many approaches are aimed at *control flow reconstruction* and resolving of indirect branches. By having abstract values represent a set of possible jump targets, an indirection can be resolved by concretizing the abstract value to all instruction addresses it represents. JAKSTAB performs binary analysis and control flow reconstruction based on this principle [27], [28]. The user manually provides a harness modeling the initial state, and relative to that initial state, the generated control flow graph overapproximates all paths in the binary. Pointer aliasing is dealt with by thrashing the symbolic state [29]. Verbeek et al. present an overapproximative approach to control flow reconstruction based on forking the state non-deterministically [43]. Their output can be exported to the Isabelle/HOL theorem prover where it can be formally proven correct. An important use case for overapproximative control flow reconstruction is *trimming* a binary by removing provably unreachable code. BINTRIMMER uses abstract interpretation to prove the reachability of dead code and trim the binary accordingly.

Using abstract interpretation for the purpose of control flow reconstruction (i.e., resolving of indirections) is different from using abstract interpretation for resolving designations of memory writes (i.e., pointer analysis). It can be seen as a specific form of the more generic technique *value set analysis* (VSA).

Value Set Analysis. Various approaches use abstract interpretation to do VSA: mapping state parts to abstract representations of a set of values that the state part may hold at a certain program point. CODESURFER/X86 utilizes as abstract domain a tuple storing a base and an offset [5], [39]. The offset is modeled by an abstract domain that combines intervals and congruences. Frameworks such as BAP [7] and ANGR [41] provide implementations of binary-level VSA. Abstract domains are typically a form of signed-agnostic intervals [35]. To our knowledge, BinPointer [26] is the work closest related to the contribution in this paper. Kim et al. provide binary-level context-insensitive pointer analysis in a sound and overapproximative fashion, while also targeting scalability and evaluating preciseness of their produced output. Their abstract domain is conceptually equivalent to the "pointer bases" domain presented in this paper, which is also similar to the abstract domain found in [5]. BinPointer reaches the conclusion of 100% soundness by running test-cases, while we reach 100% soundness through formal proofs.

Summary of relation to overapproximative approaches. Existing approaches use abstract interpretation to reconstruct control flow, or to do VSA. It is a well-known issue that in realistic, optimized and stripped COTS binaries, computations quickly become too complicated and obfuscated to be amenable for VSA sufficiently precise to enable reasoning over separation [45]. This paper presents the first approach to VSA that is formally proven correct, that is generic wrt. the abstract domain and that therefore allows different domains with different levels of preciseness to be used. To the best of our knowledge, there exists no approach that can overapproximatively assign pointer designations to virtually all memory writes in large COTS binaries, in a context-sensitive interprocedural fashion. In Section VII we aim to provide a more technical head-to-head comparison with existing tools.

III. OVERVIEW OF GENERIC CONSTITUENTS

There are three major constituents required to formulate the correctness theorem. First of all, a *concrete semantics* that provides a step function step over concrete states (denoted by s, s', $\ldots$). Second, an *abstract semantics*, defined by 1.) an abstract step function $\overline{\mathrm{step}}$ over abstract states (denoted by σ, σ', $\ldots$), and 2.) a *join* (denoted $\sqcup$). Third, a concretization function $\gamma_{\mathbb{S}}$ that maps abstract states to sets of concrete states.

We prove the following two theorems:

Theorem 1: Function $\gamma_{\mathbb{S}}$ is a *simulation relation* [4] between the concrete and abstract semantics:

$$s \in \gamma_{\mathbb{S}}(\sigma) \implies \mathrm{step}(s) \in \gamma_{\mathbb{S}}(\overline{\mathrm{step}}(\sigma))$$

This theorem shows that the abstract semantics overapproximate the concrete ones.

Second, we define an algorithm that performs symbolic execution, while maintaining a mapping ϕ from the visited instruction addresses to the abstract states. Whenever an instruction address is visited twice, the current abstract state is joined with the abstract state stored during the previous visit, and the algorithm proceeds if the joined state is unequal to the stored one. In essence, this is a fixed-point computation. We prove:

Theorem 2: The mapping ϕ produced by the algorithm provides *invariants*:

$$s \text{ is reachable} \implies s \in \gamma_{\mathbb{S}}(\phi(s.\texttt{rip}))$$

In words, any concrete reachable state s is included in the set of states represented by the abstract state stored in mapping ϕ associated to its instruction pointer `rip`. Reachability means that s is reachable from some unconstrained concrete state, with `rip` as the entry point, through a path of *resolved* control flow transfers.

A. *Concrete Semantics*

The concrete semantics are largely straightforward, except for the treatment of partially overlapping memory accesses. At the binary level, any memory access occurs through pointer computations. We formulate the assumption that *any region in memory storing a pointer is from there on not accessed in a partially overlapping fashion*. For example, consider a scenario where region $\langle \texttt{rsp}_0 - 16, 8\rangle$ has been accessed (denoting the 8-byte region 16 bytes below the original value of the stackpointer `rsp`). From that point on, regions $\langle \texttt{rsp}_0 - 16, 4\rangle$ and $\langle \texttt{rsp}_0 - 12, 4\rangle$ are still considered valid accesses. Region $\langle \texttt{rsp}_0 - 12, 8\rangle$ is not, as it partially overlaps with a previously accessed region. As a consequence of this assumption, any partially overlapping access is assumed not to produce a pointer. Since we are interested in pointer analysis, we therefore allow the concrete semantics to make values read from or written to by partially overlapping access to become *tainted*. Overapproximation then concerns untainted values *only*.

The concrete states stores *concrete values*, denoted $\mathbb{V}$. A concrete value is either an immediate bitvector or the special value $\top$ (tainted). A concrete state contains an assignment of registers to concrete values. Concrete memory assigns concrete values to concrete regions: tuples of type $\mathbb{V} \times \mathbb{V}$ containing the address and size of the region. A priori, no memory alignment information is available and thus concrete memory must store concrete values as well as the current alignment. Every read/write updates the alignment information and taints values in memory accordingly. The result is a type $\mathbb{S}$ modeling *concrete states* and function step: a formal but fully executable semantics, in which small assembly programs can be executed on concrete initial states.

B. *Abstract Semantics*

The abstract state stores *abstract values*. The datatype for abstract values, denoted $\overline{\mathbb{V}}$, is left completely polymorphic. We assume existence of a special $\overline{\top}$ element. Over this datatype, the following generic (i.e., *undefined*) functions are assumed to be available:

$\gamma_{\overline{\mathbb{V}}}$	Concretization	$\overline{\mathbb{V}} \mapsto \{\mathbb{V}\}$
$\overline{S}$	Abstract semantics	Operation $\times\, [\overline{\mathbb{V}}] \mapsto \overline{\mathbb{V}}$
$\sqcup$	Join	$\overline{\mathbb{V}} \times \overline{\mathbb{V}} \mapsto \overline{\mathbb{V}}$
$\overline{\bowtie}$	Separation	$\langle \overline{\mathbb{V}} \times \overline{\mathbb{V}}\rangle \times \langle \overline{\mathbb{V}} \times \overline{\mathbb{V}}\rangle \mapsto \mathbb{B}$
$\overline{\sqsubseteq}$	Enclosure	$\langle \overline{\mathbb{V}} \times \overline{\mathbb{V}}\rangle \times \langle \overline{\mathbb{V}} \times \overline{\mathbb{V}}\rangle \mapsto \mathbb{B}$
$\overline{==}$	Aliassing	$\langle \overline{\mathbb{V}} \times \overline{\mathbb{V}}\rangle \times \langle \overline{\mathbb{V}} \times \overline{\mathbb{V}}\rangle \mapsto \mathbb{B}$

For sake of presentation, we omit functions used for getting initial abstract values. Function $\overline{S}$ takes as input the name of an operation executed by an assembly instruction (e.g, `add`). Note that a single assembly instruction may execute several such operations (e.g., `imul`). Function $\overline{S}$ symbolically executes that operation on abstract values. The separation, enclosure and aliassing relations are over abstract regions. Function $\gamma_{\overline{\mathbb{R}}}$ concretizes an abstract region $\langle a, si\rangle$ by applying $\gamma_{\overline{\mathbb{V}}}$ to both the address and the size.

The above generic functions must satisfy a set of 26 proof obligations. Most of these are trivial (e.g., the join must be a commutative semigroup over $\overline{\mathbb{V}}$; enclosure is transitive and reflexive). We here provide some examples of interesting proof obligations. Join, separation, enclosure, aliasing, and abstract semantics must be overapproximative. For separation, this means that concretizing separate abstract regions must produce separate concrete regions; note that concrete separation $\bowtie$ can be expressed in terms of linear equalities for non-tainted values. For the semantics, overapproximation is formulated by stating that the result of applying some concrete operation $\square$ must be overapproximated by the semantics provided by $\overline{S}$ for the corresponding symbolic operation $\overline{\square}$. Here, $\square$ can, e.g., be an arithmetic, logical or bitvector operation. The presentation shows binary operators, but this easily extends to n-ary operators. Finally, there is a set of algebraic properties concerning the relations over regions, such as "Enclosure in separate region".

IV. Overview of Instantiations

We provide three different instantiations, each of which has been formally defined in Isabelle/HOL and for each of which all proof obligations have been proven. All three domains are represented by sets of elements from a different universe: $\mathcal{C}$, $\mathcal{B}$ and $\mathcal{S}$. Elements of these domains satisfy the following syntax:

$$\underbrace{\{c_0, c_1, \ldots\}^{\mathcal{C}}}_{\substack{\mathcal{C}\text{onstant} \\ \text{Computation}}} \quad \mid \quad \underbrace{\{b_0, b_1, \ldots\}^{\mathcal{B}}}_{\mathcal{B}\text{ases}} \quad \mid \quad \underbrace{\{s_0, s_1, \ldots\}^{\mathcal{S}}}_{\mathcal{S}\text{ources}}$$

The abstract points first of all concern symbolic expressions, denoted by type $\mathbb{E}$. As *operators*, symbolic expressions have arithmetic and logical operations, bit-level operations such as sign-extension or masking, and other operations related to x86-64 assembly instructions. There is a dereference operator $*\langle e, si\rangle$ – where e and si are symbolic expressions – that models reading si bytes from address e. As *operands*, symbolic expressions have *immediate values*, *state parts*, *constants* or *heap-pointers*. Immediate values are words of fixed size. State parts can be registers (e.g., `rax`, `edi`, ...) or flags (`ZF`, `CF`, ...). Constants are values of state parts relative to the initial state. For example, $\texttt{rax}_0$ denotes a constant: the initial value stored in register `rax`. Constants thus represent initial values of state parts when the current function was called. A heap-pointer is an expression of the form $\texttt{alloc}[id]$ and models the return value of an allocation function such as `malloc`. The id is an identifier to distinguish different mallocs. A *constant computation* is a symbolic expression with as operands only immediate values and constants.

Pointers with constant computations. An expression of the syntax $\{c_0, c_1, \ldots\}^{\mathcal{C}}$ is based on a non-empty set of constant computations. It non-deterministically represents any value from the given set. For example, expression

The join must be overapproximative:	$\gamma_{\mathbb{V}}(a_0) \subseteq \gamma_{\mathbb{V}}(a_0 \sqcup a_1)$
Separation must be overapproximative:	$r_0 \mathrel{\overline{\bowtie}} r_1 \wedge r_0 \in \gamma_{\mathbb{R}}(r_0) \wedge r_1 \in \gamma_{\mathbb{R}}(r_1) \implies r_0 \bowtie r_1$
Semantics must be overapproximative:	$v_0 \in \gamma_{\mathbb{V}}(a_0) \cup \{\top\} \wedge v_1 \in \gamma_{\mathbb{V}}(a_1) \cup \{\top\} \implies v_0 \square v_1 \in \gamma_{\mathbb{V}}(\overline{S}(\overline{\square}, a_0, a_1)) \cup \{\top\}$
The join respects separation:	$r \mathrel{\overline{\bowtie}} \langle a_0 \sqcup a_1, si \rangle \implies r \mathrel{\overline{\bowtie}} \langle a_0, si \rangle$
The join respects enclosure:	$\langle a_0, si \rangle \mathrel{\overline{\sqsubseteq}} \langle a_0 \sqcup a_1, si \rangle$
Enclosure in separate region:	$r_0 \mathrel{\overline{\sqsubseteq}} r_1 \wedge r_1 \mathrel{\overline{\bowtie}} r_2 \implies r_0 \mathrel{\overline{\sqsubseteq}} r_2$

Fig. 1: Examples of Proof Obligations

$\{\texttt{rsp}_0 - 8, \texttt{rdi}_0 + 16\}^{\mathcal{C}}$ simply represents any of the two constant computations. This domain is the most concrete.

Pointers with bases. Expressions of the form $\{b_0, b_1, \ldots\}^{\mathcal{B}}$ *partly* abstract away from how the pointer is computed. It keeps track only of positive *addends* in the computation that can be recognized as a basis for a pointer. The given set is a non-empty set of *bases*. A base is defined by the following datastructure:

$$\mathsf{Base} \equiv \mathsf{StackPointer}\ f \mid \mathsf{Global}\ a \mid \mathsf{Alloc}\ id \mid \mathsf{Symbol}\ name$$

Four types of pointer bases are recognized. The base may be the stackpointer (in x86-64 this is register `rsp`) pointing to somewhere in the stackframe of a certain function f. The base may be a global address a. At the binary level, the global address space is pointed to using immediate values and thus a is a 64-bit immediate word. The base may be the result of some dynamic memory allocation. Finally, a base can be a named symbol. At the binary level, external variables are named symbols with immediate addresses.

Note that any pointer will have at most one base. For example, no pointer computation would allow addition of the stack pointer and a heap address. The datastructure allows a set of bases to allow the abstract pointer to non-deterministically represent different pointers with bases.

Pointers with sources. Expressions of the form $\{s_0, s_1, \ldots\}^{\mathcal{S}}$ *fully* abstract away from how the pointer is computed. Whereas pointers with identifiable bases contain sufficient information at least for roughly establishing a memory designation, a pointer parameterized with sources only concerns what information has been used in the computation of the pointer. In other words, the pointer has been computed by some expression with operands from the given set of sources. The following sources are possible:

$$\mathsf{Source} \equiv \mathsf{Constant}\ c \mid \mathsf{Base}\ b \mid \mathsf{Fun}\ name$$

A source can be a constant, a base or the return value of some function. For example, expression $\{\texttt{rdi}_0, \texttt{rsi}_0, \mathsf{Fun}\ getc\}^{\mathcal{S}}$ indicates a pointer that has been computed using *only* the initial values of registers `rdi` and `rsi` and the return value of function `getc`.

Example 1: Consider the running example in Figure 2. The example allocates memory and performs some memory writes. Two regions (at addresses `0x3006` and `0x3008`) are relative to the initial value of the stackpointer $\texttt{rsp}_0$. The abstract pointers corresponding to these memory writes may be represented in $\mathcal{C}$: respectively $\{\texttt{rsp}_0 - 16\}^{\mathcal{C}}$ and $\{\texttt{rsp}_0 - 8\}^{\mathcal{C}}$. The memory write at address `0x3005` occurs on the heap. Even if *offset* is some convoluted dynamically computed offset, the abstract pointer will be representable in $\mathcal{B}$: $\{\mathsf{Alloc}\ \texttt{0x3003}\}^{\mathcal{B}}$. Finally, the memory write at address `0x3007` writes to the address initially stored in register `rdx` plus the return value of function `getc`. The abstract pointer is representable in $\mathcal{S}$: $\{\texttt{rdx}_0, \mathsf{Fun}\ \texttt{getc}\}^{\mathcal{S}}$.

A. Separation over Abstract Pointers

Each of the three domains allows a different type of reasoning over separation. Intuitively, separation between two pointers implies that *necessarily* – i.e., for any address represented by the two pointers – two memory writes commute. Separation is denoted by the infix operator $\bowtie$.

Consider two pointers $p_0 + \mathit{offset}$ and p_1. If both pointers are defined by constant computations, separation can be decided by an SMT solver. For the other domains, however, one may argue that separation cannot be guaranteed under any circumstances. We argue this is too strict. For example, if one pointer is based on the heap and the other on the local stack, then it is safe to assume that even if *offset* is chosen nefariously, writes to the two pointers will either cause a segmentation fault or they will commute. Assuming separation is thus safe. On the other hand, we argue that if both pointers are based on local stackframes of different functions, then assuming separation is dangerous as a stack overflow may cause the pointers to overlap.

It may be the case that separation is not necessarily true, but *desirable*. Consider a pointer with source set $\{\texttt{rdi}_0\}$. The only way for that pointer to overlap with the stackframe of the current function, is if initially – at the time the current function was called – register `rdi` contained a pointer to below its own stackframe. That is invalid source code and can therefore be considered undesirable. For various use cases, it can be useful to distinguish "necessarily" separation from "desirable" separation (see Section VI).

For domains $\mathcal{B}$ and $\mathcal{S}$, whether two abstract pointers are separate is in essence domain specific knowledge on what assumptions can validly be made when dealing with an x86-64 binary. In Figure 3, this knowledge will be encoded in algebraic definitions. Depending on ones view, one may easily add or remove cases of separation, and move cases from "desirable" to "necessarily" or the other way around.

Pointers with constant computations. For two constant computations, separation can be formulated as a linear programming problem, solvable by an SMT solver [34]. This requires the size of the regions pointed to, to be known. Typically, these sizes can syntactically be inferred from the current instruction. We define function $\mathsf{SMT}[\bowtie]$ as a function that takes as input two constant computations c_0 and c_1,

```
0x3000 : mov rbp, rsp
0x3001 : call getc
0x3002 : mov rcx, rax
0x3003 : call malloc
0x3004 : lea rsi, [rbp - 8]
0x3005 : mov qptr [rax + offset], rsi
0x3006 : mov qptr [rsp-16], 0x2000
0x3007 : mov qptr [rdx + rcx], rax
0x3008 : mov dptr [rsi], 0
```

(a) x86-64 Assembly

$$
\begin{aligned}
\sigma.\texttt{rsp} &= \{\texttt{rsp}_\emptyset\}^{\mathcal{C}}\\
\sigma.\texttt{rbp} &= \{\texttt{rsp}_\emptyset\}^{\mathcal{C}}\\
\sigma.\texttt{rcx} &= \{\mathsf{Fun}\ \mathit{getc}\}^{\mathcal{S}}\\
\sigma.\texttt{rax} &= \{\texttt{alloc[0x3003]}\}^{\mathcal{C}}\\
\sigma.\texttt{rsi} &= \{\texttt{rsp}_\emptyset - 8\}^{\mathcal{C}}\\
\sigma.*\langle\{\mathsf{Alloc}\ \texttt{0x3003}\}^{\mathcal{B}}\rangle &= \{\texttt{rsp}_\emptyset - 8\}^{\mathcal{C}}\\
\sigma.*\langle\{\texttt{rsp}_\emptyset - 16\}^{\mathcal{C}}, 8\rangle &= \{\mathsf{Global}\ \texttt{0x2000}\}^{\mathcal{B}}\\
\sigma.*\langle\{\texttt{rdx}_\emptyset, \mathsf{Fun}\ \texttt{getc}\}^{\mathcal{S}}\rangle &= \{\texttt{alloc[0x3003]}\}^{\mathcal{C}}\\
\sigma.*\langle\{\texttt{rsp}_\emptyset - 8\}^{\mathcal{C}}, 4\rangle &= \top
\end{aligned}
$$

(b) Abstract State

Fig. 2: Assembly code. For sake of presentation, pseudo code is given on the right. *offset* denotes some dynamically computed offset. Instruction `lea` loads an address into register `rsi` without reading from memory. The binary has a data section with address range `0x2000` to `0x2040`.

and two sizes si_0 and si_1 and returns true if and only if the following SMT problem is unsatisfiable: $c_0 \leq a < c_0 + si_0 \wedge c_1 \leq a < c_1 + si_1$. In words, an address a that is in both regions should be unsatisfiable. Constant computations typically consist of arithmetic operations containing only $+$, $-$ and $*$, and bit-level operations supported by SMT-theory QF_BV.

Pointers with bases. Separation for pointers with bases can be decided algebraically. Figure 3 presents relation $\bowtie^{\mathsf{B}}$, i.e., separation over bases. The local stack frame of any function f is assumed to be separate from the global address space and the heap. As argued above, separation over different stackframes is desirable, but not necessarily. The global address space is assumed to be separate from the local address space and the heap. The global address is not necessarily assumed to be separate from the symbols in the binary, i.e., external variables. Both are typically constant immediates, within the range of addresses within the binary. Separation is considered to be desirable. If two global addresses are based on immediates that fall within the range of different data sections of the binary, then separation is considered desirable.

Two allocations with different ids are assumed to be separate, as different ids ensure these were different calls to `malloc`. A pointer based on `malloc` should not lead to a write overlapping with the region of a different `malloc`. Whether this separation should be considered necessary or desirable is debatable. Heap overflows are common and critical vulnerabilities, and if these are part of the attacker model, then this separation should be considered desirable. Note that two pointers based on allocations with the same *id*, are not necessarily *not* separate.

Pointers with sources. Since pointers with sources are more abstract, there are fewer cases that allow deciding separation necessarily. For example, consider two pointers with source sets $\{\texttt{rdi}_\emptyset\}$ and $\{\texttt{rsi}_\emptyset\}$. Whether these can be considered separate depends on the initial state of the current function, i.e., whether registers `rdi` and `rsi` initially were separate. However, there still are cases when separation can be decided. First, when two sources are actually bases, the above relation $\bowtie^{\mathsf{B}}$ can used as decision procedure. Second, consider a pointer computed using constant $\texttt{rdi}_\emptyset$ and a newly allocated pointer. As long as the allocation happened within the current function, separation can safely be assumed: essentially this assumes that the initial state cannot predict where `malloc` will allocate memory. Since sources of the form $\mathsf{Fun}\ f$ do not represent pointers, separation between these sources and all other sources can be assumed. Finally, consider a source $\texttt{rdi}_\emptyset$, i.e., the initial value of register `rdi` when the current function was called. It is possible that the caller initialized `rdi` with a pointer to the stackframe of the callee, but we consider it undesirable.

Example 2: Revisiting Example 1, we address the memory accesses in order of execution. First a write happens to pointer $p_0 = \{\mathsf{Alloc}\ \texttt{0x3003}\}^{\mathcal{B}}$. Then, a write happens to pointer $p_1 = \{\texttt{rsp}_\emptyset - 16\}^{\mathcal{C}} = \{\mathsf{StackPointer}\ \texttt{0x3000}\}^{\mathcal{B}}$. These two regions are necessarily separate for all three domains. Then, a write happens to pointer $p_2 = \{\texttt{rdx}_\emptyset, \mathsf{Fun}\ \texttt{getc}\}^{\mathcal{S}}$. Separation between p_0 and p_2 follows necessarily for domains $\mathcal{B}$ and $\mathcal{S}$. However, separation between p_1 and p_2 is not necessarily true. It is, however, *desirable*. Finally, a write happens to pointer $p_3 = \{\texttt{rsp}_\emptyset - 8\}^{\mathcal{C}}$. Similar reasoning applies for separation with p_0 and p_2. For domain $\mathcal{C}$, separation between p_1 and p_3 is decided by an SMT solver.

In similar fashion, enclosure and aliassing are instantiated. The join is defined as set-union. Instantiating abstract semantics and concretization is straightforward.

V. Intraprocedural Pointer Analysis

Algorithmically, intraprocedural pointer analysis can be achieved using a standard abstract interpretation algorithm [12], [23]. Essentially, starting at some initial abstract state with the instruction pointer set to the entry point of the binary (or a function of interest), symbolic execution traverses the assembly instructions step-by-step using function $\overline{\text{step}}$. It runs until a return statement, or an exiting function call. Whenever an instruction address is visited twice, the current state σ_{curr} is compared to the state σ_{stored} belonging to the previous visit. If state σ_{stored} is more abstract then state σ_{curr}, exploration can stop. Otherwise, the two states must be *joined* to a state σ_{join}. That state is stored and exploration continues.

Necessarily:				
StackPointer f	$\bowtie^{\mathsf{B}}$	Alloc id		
StackPointer f	$\bowtie^{\mathsf{B}}$	Global a		
StackPointer f	$\bowtie^{\mathsf{B}}$	Symbol $name$		
Global a	$\bowtie^{\mathsf{B}}$	Alloc id		
Alloc id_0	$\bowtie^{\mathsf{B}}$	Symbol $name$		
Alloc id_0	$\bowtie^{\mathsf{B}}$	Alloc id_1	if	$id_0 \neq id_1$
Base b_0	$\bowtie^{\mathsf{S}}$	Base b_1	$\equiv$	$b_0 \bowtie^{\mathsf{B}} b_1$
Constant c	$\bowtie^{\mathsf{S}}$	Base Alloc id	if	id belongs to current function
Fun f	$\bowtie^{\mathsf{S}}$	s_1		
Desirable:				
StackPointer f	$\bowtie^{\mathsf{B}}$	StackPointer f'	if	$f \neq f'$
Global a	$\bowtie^{\mathsf{B}}$	Global a'	if	a and a' are from different data sections
Global a	$\bowtie^{\mathsf{B}}$	Symbol $name$		
Constant c	$\bowtie^{\mathsf{S}}$	Base StackPointer f	if	f is current function

Fig. 3: Separation over abstract pointers: the smallest symmetric relation such that the above holds.

Effectively, this algorithm provides a (partial) mapping from instruction addresses to stored symbolic states. A post-state σ_{post} is computed by taking the supremum of all terminal non-exiting states. Thus, we define *state* and a *symbolic step function* on top of the generic constituents.

Abstract States. A state stores values for registers, flags and memory. An abstract *state part* is defined by the following datatype:

$$\overline{\mathbb{SP}} \equiv \mathsf{Register}\ \texttt{r} \mid \mathsf{Flag}\ \texttt{f} \mid \mathsf{Memory}\ (\overline{\mathbb{V}} \times \mathbb{N}?)$$

A memory statepart is represented as a region, a tuple with an abstract pointer and optionally a size. Whenever clear from context, we will omit the constructors. For example, $sp = \texttt{rax}$ simply means that statepart sp equals register $\texttt{rax}$. An abstract state is a partial mapping from stateparts to abstract values: $\overline{\mathbb{S}} \equiv \overline{\mathbb{SP}} \rightharpoonup \overline{\mathbb{V}}$.Notation $\sigma.\texttt{r}$ denotes the current value stored in register $\texttt{r}$. Memory partitions the address space into *separate* abstract regions, and assigns stored values to each of these regions. Notation $\sigma.*\langle r\rangle$ denotes the value stored in region r, where the size is omitted when not available.

Example 3: Revisit the running example in Figure 2. Based on the separation relations decided in Example 2, the state σ in Figure 2b can model the state after execution of the assembly snippet. Note that this state is based on *desirable* separation, as otherwise regions would need to get joined. Intuitively, the following claims are modeled by this state:

- Any region based on a heap-pointer allocated at line `0x3003` contains no pointers other than the pointer $\texttt{rsp}_0 - 8$.
- The 8-byte region at address $\texttt{rsp}_0 - 16$ stores a value that may point to the global data section.
- A memory write has occurred that is determined by the initial value of register `rdx` as well as input provided via function `getc`. That memory write was assumed to be separate from all others.
- If the value stored in region $\langle \texttt{rsp}_0 - 8, 4\rangle$ would be dereferenced, then no information on its designation in memory is known.

The claims in the above example can be formalized by *concretization* of abstract states to concrete ones. Concretization of state parts is provided by function $\gamma_{\mathbb{SP}}$ of type $\overline{\mathbb{SP}} \mapsto 2^{\mathbb{SP}}$:

$$\begin{array}{lcl} \gamma_{\mathbb{SP}}(\mathsf{Register}\ \texttt{r}) & \equiv & \{\ \mathsf{Register}\ \texttt{r}\ \} \\ \gamma_{\mathbb{SP}}(\mathsf{Flag}\ \texttt{f}) & \equiv & \{\ \mathsf{Flag}\ \texttt{f}\ \} \\ \gamma_{\mathbb{SP}}(\mathsf{Memory}\ \overline{r}) & \equiv & \{\ \mathsf{Memory}\ r \mid r \in \gamma_{\mathbb{R}}(\overline{r})\ \} \end{array}$$

Definition 1: The *concretization* of states is function $\gamma_{\mathbb{S}}$ of type $\overline{\mathbb{S}} \mapsto 2^{\mathbb{S}}$, and is defined as:

$$\gamma_{\mathbb{S}}(\sigma) \equiv \{s \mid \forall \overline{sp} \cdot \sigma.\overline{sp} = \overline{v} \implies \forall sp \in \gamma_{\mathbb{SP}}(\overline{sp}) \cdot \exists v \in \gamma_{\mathbb{V}}(\overline{v}) \cdot s.sp = v\}$$

In words, abstract state σ is mapped to any concrete state s in which both all stateparts and all stored values have been concretized.

Abstract Step Function. Every assembly instruction can be written as a sequence of micro-instructions of the following form:

$$\texttt{dst} := f(\texttt{in}_0, \texttt{in}_1, \ldots)$$

A single destination – be it a register, a memory region or a flag – gets assigned the return value of a pure operation f that is based on zero or more input-operands (registers, memory regions, flags, or immediate values). The operation typically corresponds to a mnemonic, e.g., `add` or `imul`. In x86-64, many (but not all) instructions have one destination operand (register or memory) and may set a list of flags. Typically, it cannot be the case that both destination and sources are memory. We here make no assumptions, and generalize the symbolic step function over any sequence of micro-instructions. Transformation from basic assembly to sequences of micro-instructions can be done, e.g., using the Ghidra decompiler which translates assembly instructions of various architectures into *low P-code* [40].

The semantics of executing a micro-instruction μ consists of resolving the input-operands, applying operation f, and writing the produced value to the state. This is implemented in function $\overline{\mathsf{step}}(\mu, \sigma)$. The abstract semantics $\overline{S}$ are used to apply operation f on abstract values. Functions $\mathsf{read}(r, \sigma)$ and write(r, a, σ) are defined that take care of memory accesses in abstract states.

Correctness and Termination. Correctness – the abstract semantics overapproximate the concrete semantics – has been defined at the start of Section III and has been formally proven correct in Isabelle/HOL. Note that it may be the case that paths are unexplored due to unresolved indirections.

Termination has also been proven, but it requires the proof obligation that there does not exist an infinite chain of different states such that $\sigma_0 \sqcup \sigma_1 \sqcup \ldots$. We thus ensure termination by putting bounds on the sizes of the sets of the abstract pointers. These bounds are chosen manually. They do not affect correctness. Making them larger makes the pointers more precise, but increases running times. For bounds $\mathcal{C}$, $\mathcal{B}$, and $\mathcal{S}$ the bounds are resp. 10, 5 and 250. Any pointer with more elements is shifted to $\overline{\top}$.

VI. Use Cases

We here discuss several use cases with small pedagogical examples.

Integration into Disassembly: A fundamental problem in disassembly is resolving indirections. Typically, indirect jumps can be analyzed through *intra*procedural analysis. For example, they are the result of reading a jump table induced by a `switch` statement. For indirect calls, however, often *inter*procedural analysis is necessary. An indirect call is often the result of a function callback. These typically happen across function boundaries: a function pointer is passed from function to function until it is called. This scenario mandates context-sensitive interprocedural analysis where the context provides sufficient information to resolve indirect calls.

```
Function f:
0x6000 :  mov qword ptr [0x2010], 0x6050
0x6001 :  call 0x6500
0x6002 :  call exit
. . .
0x6050 :  push rbp
0x6051 :  . . .
. . .
Function g:
0x6500 :  mov rax, qword ptr [0x2010]
0x6501 :  call rax
0x6502 :  ret
```

Fig. 4: Example of Indirect Call

Figure 4 provides an example. Function pointer `0x6050` is written to a global variable (at address `0x2010`). Function h will be analyzed in a context where the pre-state provides:

$$\sigma_{\text{pre}}.*\langle\{\text{Global 0x2010}\}^{\mathcal{B}}\rangle = \{\text{0x6050}\}^{\mathcal{C}}$$

When symbolically executing the indirect call at Line `0x6501`, it is overapproximatively known that any pointer stored in the global address space based on address `0x2010` points to instruction address `0x6050`. The recursive traversal can therefore consider `0x6050` as a reachable instruction address. Note that in a stripped binary, functions are not delineated: it is not known what addresses are function entries. Without context, the indirect call at Line `0x6501` cannot be resolved and the entire function at entry `0x6050` would have been missed. Section VII-B provides data on how many indirections can be resolved in practice.

Preliminary to Decompilation: At the assembly level, there are no variables. Bottom-up analysis, such as decompilation, has no ground truth as to what regions in memory constituted variables. A variable in source code typically is compiled to a memory region, with the characteristic that this region is separate from any memory write not to the same variable. As example consider Figure 5. It shows assembly with some hypothetical pointer analysis result on the right. Based on the derived pointers, a decompilation tool can assign a variable `x` to lines `0x7000` and `0x7002`.

Bottom-up Dataflow Analysis: The post-state produced by analysis of a function provides overapproximative insight into what parts of the state are modified by a function. One application of this, is that it can be used to verify whether a certain function abides by a calling convention. The calling convention designates certain registers to be non-volatile, i.e., they must be preserved by a function. This can be observed from the post-state directly. If the post-state provides that, for a register `r`: $\sigma_{\text{post}}.\texttt{r} = \{\texttt{r}_0\}^{\mathcal{C}}$,then this indicates that register `r` has been properly preserved. Many compilers use a push/pop pattern to achieve such preservation: a register value is initially pushed to the local stack, and popped just before return. Calling convention adherence requires abstract pointer analysis to be sufficiently precise throughout symbolic execution that the local region into which a register value is pushed is not overwritten. A push/pop pattern is not necessary though: our approach is transparent to the means of register preservation a function may use.

We provide an additional example demonstrating this overapproximative form of pointer analysis can be used for live variable analysis. Consider a function returning with the state in Example 3. The pointer allocated at line `0x3003` is stored in the return register `rax` and can therefore considered to be live. However, if the state overapproximately indicates that the pointer is not returned and not written to global memory, the pointer can be considered as "not live" after return of the function.

Finding suspect patterns for automated exploit generation: Pointer analysis can enhance real-world downstream analyses. As an example, we consider automated exploit generation: automatically finding bugs and generating working exploits [2], [44]. One of the many challenges in this field, is to deal with state space explosion. Our pointer analysis can be used to find patterns in the binary that may lead to vulnerabilities, thereby pruning the state space to be explored.

As a concrete case study, we can enumerate all instructions in a binary that do a function call to an external function, *and* that pass a pointer to the current stackframe as parameter to that function. This is a suspect pattern, as the external function has been given the opportunity to overwrite the return address. We have applied our pointer analysis to the `ret2win` challenge provide by ROP emporium[2]. The above heuristic finds an instruction at address `0x400701`: `call memset` where register

[2]https://ropemporium.com

```
0x7000 :  mov qword ptr [rdi], 42
0x7001 :  mov qword ptr [rsi], 43
0x7002 :  mov qword ptr [rbp-40], 44
```

(a) x86-64 Assembly

```
x     :=  42
*rsi  :=  43
x     :=  44
```

(b) Decompiled

$$\begin{array}{lcl} \texttt{rdi} & = & \{\texttt{rsp}_0 - 48\}^{\mathcal{C}} \\ \texttt{rsi} & = & \{\mathsf{Alloc}\ id\}^{\mathcal{B}} \\ \texttt{rbp} & = & \{\texttt{rsp}_0 - 8\}^{\mathcal{C}} \end{array}$$

(c) Abstract Pointers

Fig. 5: Assembly code, decompiled to code with variables based on abstract pointers.

rdi (the first parameter under the System V ABI calling convention) contains a local pointer. Indeed, the example is (purposefully) exploitable, and the exploit leverages exactly this particular instruction.

VII. Evaluation

In addition to the formal proofs of the soundness of our approach, we provide a prototype implementation and conducted a series of experimental studies to evaluate *soundness* and *preciseness*. We run 1.) a *comparative* evaluation against the state-of-the-art, and 2.) a more in-detail evaluation *per instantiation*.

A. Comparative Evaluation

The closest related works to our binary pointer analyzer are BinPointer [26] and BPA [25]. Unfortunately, even after contacting the authors of the papers, either their source code was not available or the code was not runnable on our examples. Therefore, to stay fair in our comparison, we used the exact same dataset of binaries that BinPointer and BPA used (SPEC_2006_V1) and we employed the same definition of soundness and precision they used in their paper.

We map all abstract pointers to a subset of $\{L, G, H\}$ (for: local, global, heap). Domains that are provably within the current stackframe are mapped to $\{L\}$, domains that are relative to the stackpointer but that may possibly be above the current stack frame (e.g., point to the stackframe of a caller) are mapped to $\{L, H\}$. Global bases are mapped to $\{G\}$. Top is mapped to $\{L, G, H\}$, the rest is considered heap and mapped to $\{H\}$. This produces function PA that maps instruction addresses to observations.

To assess the ground truth, we built an instrumentation tool based on the dynamic binary instrumentation tool PIN [32]. Each memory write observed at run-time to some address a is mapped to an element of $\{L, G, H\}$. If $\texttt{rsp}_0 - 0x10000 \leq a \leq \texttt{rsp}_0$, address a is considered local (here $\texttt{rsp}_0$ denotes the value of the stackpointer when the current function was called). We cannot know how large the stackframe is, and overapproximate the size with the constant $0x10000$ for sake of observation. If address a is a memory address covered by any of the sections where the binary is located at run-time, it is considered global. In all other cases, address a is considered heap. By running multiple executions, each memory write is mapped to a subset of $\{L, G, H\}$ as well. This produces function GT that maps instruction addresses to observations. The total set of (instruction addresses of) observed memory writes is denoted with W.

Soundness: We consider a static pointer analysis to be sound if its results support the ground truth, i.e., the observations of PIN. In other words, the pointer analysis must predict a superset of the ground truth. Soundness is computed using the *recall*: the percentage of supported memory writes to the total number. If the recall is 100%, the pointer analysis is sound.

Precision: Soundness does not imply usefulness. For example, if the pointer analysis returns $\top$ for all the instructions, it will be consider sound but useless. We therefore measure *precision* as well. In words, precision measures the specificity of the returned pointer domains. The precision is computed as the total average of the percentage of domains that PA overapproximated but were not observed by GT.

$$\begin{array}{lcl} \mathsf{recall} & \equiv & 100 * \frac{|\{a \in \mathsf{W} \cdot \mathsf{GT}(a) \subseteq \mathsf{PA}(a)\}|}{|W|} \\ \mathsf{precision} & \equiv & 100 * \mathsf{avg}_{a \in \mathsf{W}}(1 - \frac{|\mathsf{PA}(a) \setminus \mathsf{GT}(a)|}{3}) \end{array}$$

Table I shows results. The authors of BinPointer have reported that the recall of their approach on the SPEC dataset is 100%. Since we also observed the same recall for our approach on this dataset, we do not include it in the table. As the results show, when it comes to heap and global memory accesses, our approach achieves over a 29% and 15% higher precision compared to BinPointer and BPA. For the local memory accesses, our approach shows almost similar precision compared to that of BinPointer. All other larger case studies reported in [26] are not publicly available. Interpreting and comparing their published results subjectively, we conclude that our work achieves at least similar precision. The largest reported binary for BinPointer is 161K instructions. In the next section, we show that our tool scales upto 507K instructions, and we thus argue we are at least as scalable as well.

B. Per-instantiation Evaluation

We also evaluate per instantiation soundness and precision. As case studies, we consider a set of binaries covering over 1M instructions (see Table II). Overall recall is 100%.

We discuss precision in more detail (see Figure 6). Instantiation $\mathcal{C}$ is always 100% precise – as no abstraction is applied – unless it assigns $\top$ to a pointer. It may assign, e.g., $\{\texttt{rsp}_0 - 8, \texttt{0x3000}\}^{\mathcal{C}}$, in which case on different paths the pointer can be local or global (0.79% of the overall memory writes). Overall, instantiation $\mathcal{C}$ assigns a non-$\top$ pointer to 85.22% of all memory writes.

Instantiation $\mathcal{B}$, then, only marginally improves that number. It assigns pointers to 0.95% more memory writes, and for those only the base of the pointer is known. However, the value of this instantiation fluctuates per binary: for some it did not improve on $\mathcal{C}$ at all, but for others up to 5% more memory

Binary	#instructions	Local			Global			Heap		
		BPA	BinPointer	This paper	BPA	BinPointer	This paper	BPA	BinPointer	This paper
mcf	2.4 k	26.3	100.0	100.0	27.0	85.7	91.5	N/A	N/A	57.8
lbm	2.2 k	22.3	99.5	100.0	73.1	100.0	66.7	N/A	N/A	33.3
libquantum	9.6 k	47.9	100.0	100.0	100.0	100.0	100.0	6.9	6.9	50.0
bzip2	11.0 k	16.9	93.2	89.2	51.7	51.7	100.0	3.6	21.8	40.0
sjeng	22.2 k	32.7	97.5	79.1	55.1	55.6	99.8	N/A	N/A	56.1
milc	23.1 k	49.4	99.4	93.3	81.2	88.9	95.2	23.7	23.7	51.1
hmmer	60.4 k	38.0	99.9	98.9	76.4	76.4	100.0	7.6	11.5	65.4
h264ref	100.3 k	35.3	97.3	97.1	6.2	65.5	90.6	24.2	40.8	47.6
Average		33.6	98.4	94.7	58.8	78.0	93.0	13.2	20.9	50.2

TABLE I: Comparison study of the precision (%) of BPA, BinPointer, and this paper on the SPEC dataset.

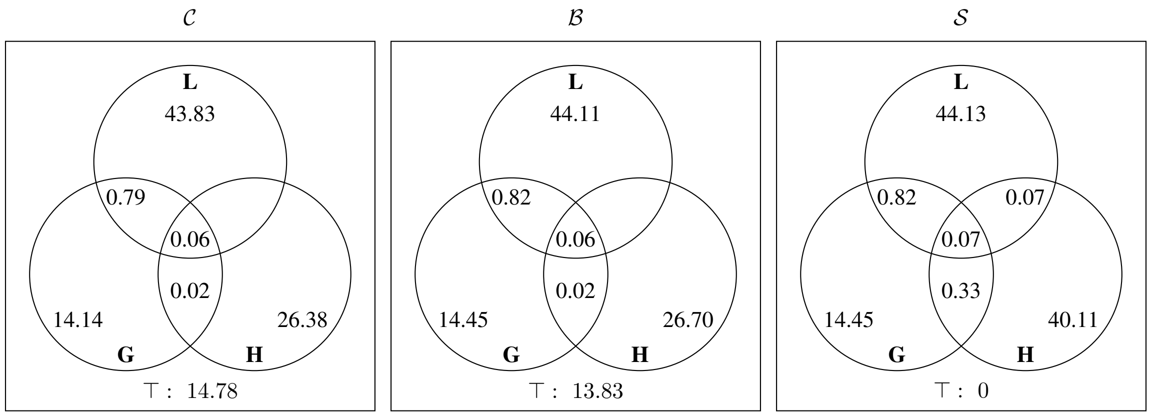

Fig. 6: Precision results per instantiation. The numbers are percentages. For example, instantiation $\mathcal{C}$ assigned a local pointer to 43.83% of all memory accesses, and to 0.79% a pointer set with both a local and a global pointer.

writes got assigned a non-$\top$ pointer. These are typically the hard cases, e.g., a local pointer to an array for which only a base could be established.

Instantiation $\mathcal{S}$, finally, assigns a non-$\top$ pointer to all memory writes. Figure 6 shows that this mostly concerns heap-pointers. From this we can conclude that the cap on the number of sources is never hit, since otherwise this instantiation would assign $\top$. For those writes where both $\mathcal{C}$ and $\mathcal{B}$ assign $\top$, instantiation $\mathcal{S}$ at least provides us with information on which sources (e.g., function inputs or `malloc`s) were used to compute the pointer value.

A holistic way of looking at precision, is 1.) to see whether the pointer analysis is sufficiently precise such that for each function all its pointers are assigned a domain separate from the top of the stack frame (where the return address is stored), and 2.) that it enables resolving indirections. Table II shows that for 94.2% of all functions this was the case (**OK**). For 5.41% of the functions, not all indirections could be resolved (**UN**), but all resolved paths ending in a return were **OK**. Finally, for 0.39% of the functions, our pointer analysis was not sufficiently precise to show that the return address was not overwritten (**ERR**). This typically happens for functions with complex stackpointer manipulation, such as stack probing or dynamic stack allocation.

Binary	#instrs	Time (s)	#functions		
			OK	UN	ERR
`du`	30 k	9	173	7	0
`gzip`	14 k	4	101	5	0
`host`	12 k	4	62	8	2
`sha512sum`	10 k	5	36	3	0
`sort`	18 k	8	146	8	0
`spec/*`	150 k	82	942	41	11
`sqlite3`	319 k	93	1687	144	8
`ssh`	124 k	61	523	32	8
`tar`	91 k	15	300	17	0
`vim`	507 k	266	2922	86	1
`wc`	6 k	1	46	4	0
`wget2`	61 k	10	578	81	2
`xxd`	2 k	1	13	0	0
`zip`	24 k	69	103	2	0
Total	1.4 m		94.2%	5.41%	0.39%

TABLE II: Evaluation on an Apple M1 Pro with 32 GB of memory. Memory usage was at most roughly 15GB for `vim`.

VIII. Conclusion

This paper presented an approach to formally proven correct binary-level pointer analysis, that aims to assign a designation to each memory write in a binary. The designations are provably overapproximative: the write provably cannot occur to any region in memory outside of its designation. Evaluation confirms this soundness, and shows that precision is comparable to or improves upon the state-of-the-art.

Many existing approaches to binary analysis, whether it is disassembly, decompilation, binary patching or security analysis, are unsound. State-of-the-art tools apply heuristics, incorporate best practices, and are generally based on extensive human-in-the-loop knowledge. Decompilation becomes a form of art rather than an algorithm. We envision an overapproximative – provably sound – approach as an alternative. This requires provably sound disassembly, control flow reconstruction, decompilation, and type inference, to begin with. At the heart of all of these overapproximative techniques lies a proper understanding of the semantics of each individual instruction. Binary-level pointer analysis aims to aid these future endeavors in overapproximative binary analysis, by indicating what the effect of each memory write in a binary can be.

Data-Availability Statement

The implementation of the pointer analysis in this paper, all case studies, and the formal Isabelle/HOL proofs are available anonymously at: https://doi.org/10.5281/zenodo.14223108.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions, which helped to greatly improve the paper. This work is supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4028, by DARPA and the Army Contracting Command Aberdeen Proving Grounds (ACC-APG) under Prime Contract #W912CG23C0024, and by DARPA under Prime Contract #HR001124C0492.

References

[1] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, University of Copenhagen, 1994.

[2] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[3] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 12–22. [Online]. Available: https://doi.org/10.1145/2001420.2001423

[4] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[5] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004, pp. 5–23.

[6] ——, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 1–84, 2010.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469.

[8] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective symbolic execution," 2009. [Online]. Available: http://infoscience.epfl.ch/record/139393

[9] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, feb 2012. [Online]. Available: https://doi.org/10.1145/2110356.2110358

[10] J.-D. Choi, M. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1993, pp. 232–245.

[11] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.

[12] P. Cousot and R. Cousot, "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*, 1977, pp. 238–252.

[13] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1021–1038.

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

[15] S. Debray, R. Muth, and M. Weippert, "Alias analysis of executable code," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998, pp. 12–24.

[16] A. Djoudi and S. Bardin, "BINSEC: binary code analysis with low-level regions," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, ser. Lecture Notes in Computer Science, C. Baier and C. Tinelli, Eds., vol. 9035. Springer, 2015, pp. 212–217. [Online]. Available: https://doi.org/10.1007/978-3-662-46681-0_17

[17] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 242–256, 1994.

[18] J. Feist, L. Mounier, S. Bardin, R. David, and M. Potet, "Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free," in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, SSPREW@ACSAC 2016, Los Angeles, California, USA, December 5-6, 2016*, M. D. Preda, N. Stakhanova, and J. T. McDonald, Eds. ACM, 2016, pp. 2:1–2:12. [Online]. Available: https://doi.org/10.1145/3015135.3015137

[19] G. Girol, B. Farinier, and S. Bardin, "Not all bugs are created equal, but robust reachability can tell the difference," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12759. Springer, 2021, pp. 669–693. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_32

[20] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin, "Automating software testing using program analysis," *IEEE software*, vol. 25, no. 5, pp. 30–37, 2008.

[21] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008. [Online]. Available: https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[22] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 54–61.

[23] S. Horwitz, A. Demers, and T. Teitelbaum, "An efficient general iterative algorithm for dataflow analysis," *Acta Informatica*, vol. 24, no. 6, pp. 679–694, 1987.

[24] T. Kapus and C. Cadar, "A segmented memory model for symbolic execution," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 774–784. [Online]. Available: https://doi.org/10.1145/3338906.3338936

[25] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level." in *NDSS*, 2021.

[26] S. H. Kim, D. Zeng, C. Sun, and G. Tan, "BinPointer: towards precise, sound, and scalable binary-level pointer analysis," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, 2022, pp. 169–180.

[27] J. Kinder, "Static analysis of x86 executables," Ph.D. dissertation, Technische Universität, Darmstadt, Nov. 2010. [Online]. Available: http://tuprints.ulb.tu-darmstadt.de/2338/

[28] J. Kinder and D. Kravchenko, "Alternating control flow reconstruction," in *Verification, Model Checking, and Abstract Interpretation*, V. Kun-

cak and A. Rybalchenko, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 267–282.

[29] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *International Conference on Computer Aided Verification*. Springer, 2008, pp. 423–427.

[30] W. Landi, B. G. Ryder, and S. Zhang, "Interprocedural modification side effect analysis with pointer aliasing," *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 56–67, 1993.

[31] X. Leroy and S. Blazy, "Formal verification of a c-like memory model and its uses for verifying program transformations," *Journal of Automated Reasoning*, vol. 41, no. 1, pp. 1–31, 2008.

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[33] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," *SIGPLAN Not.*, vol. 47, no. 4, p. 337–348, mar 2012. [Online]. Available: https://doi.org/10.1145/2248487.2151012

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

[35] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Signedness-agnostic program analysis: Precise integer bounds for low-level code," in *Asian Symposium on Programming Languages and Systems*. Springer, 2012, pp. 115–130.

[36] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[37] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 833–851.

[38] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, p. 1467–1471, sep 1994. [Online]. Available: https://doi.org/10.1145/186025.186041

[39] T. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *International Conference on Compiler Construction*. Springer, 2008, pp. 16–35.

[40] R. Rohleder, "Hands-on Ghidra tutorial about the software reverse engineering framework," in *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019, pp. 77–78.

[41] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[42] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996, pp. 32–41.

[43] F. Verbeek, J. Bockenek, Z. Fu, and B. Ravindran, "Formally verified lifting of c-compiled x86-64 binaries," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 934–949.

[44] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2139–2154.

[45] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, "BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–31, 2019.