Holmes: Binary Analysis Integration Through Datalog

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Matthew Maurer

B.S. in Computer Science, California Institute of Technology
M.S. in Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

Oct 2018

Acknowledgments

This thesis work was advised by David Brumley. Other members of the committee include Frank Pfenning (CMU CS), Manuel Egele (BU ECE), and Lujo Bauer (CMU ECE). Special thanks goes to Frank Pfenning for all his feedback given during the final revisions period.

I would like to thank Maverick Woo for our discussions on multi-valued logic which helped give rise to circumscription, and general support throughout my PhD career. Thanks to Kyle Soska and Mike Annichiarico, for listening to my ideas and acting as sounding boards.

I'd also like to thank in no particular order Patrick Xia, Michael Sullivan, Ben Blum, Michael Artzenius, Chris Lu, Mike Kaye, and Jasmine Sears for helping me remain sane throughout my time in grad school.

This work was funded by DARPA award N660011324040 and NSF award DGE1419362. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Abstract

We need tools and techniques for program analysis that address the unique problems faced when analyzing compiled code. During compilation, the compiler strips away source code information, such as control flow, types, and variable locations. As a result, analysis of compiled code faces a number of unique challenges. Previous work has proposed techniques to perform an individual analysis in isolation. However, previous work has not focused on the *co-dependencies* between these individual analysis. For example, value set analysis requires control flow analysis, which in turn requires value set analysis. We need techniques that can handle such co-dependencies in order to effectively check higher-level security properties.

In this thesis we propose using a logic-language based approach to encode co-dependent analysis and build a tool called Holmes based on our approach to demonstrate our ideas on real code. We demonstrate novel techniques for extending Datalog semantics to efficiently and effectively reason about real binary code. Our approach allows one to elegantly encode the co-dependence between value-set and control-flow analysis described above. We demonstrate Holmes and our approach by encoding real-world co-dependent analysis, and showing that Holmes can act as a framework to effectively check higher-level properties, such as use-after-free, on real compiled code.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We need program analysis techniques that can find security bugs in binary programs. Binary program analysis is essential because software is often only available in binary form. Even when source code is available, some security properties are more appropriately checked at the binary level. As an example, use-after-free bugs present in source may not be present at the binary level if the compiler chooses to elide a read when a register caches its value.

Authors designed existing tools [16, 35, 54, 70] as a sequence of analyses to perform with the results of each previous analysis being available to the next. In this model, an analysis author must ensure the framework calculates any information needed before their analysis, but after any analyses that could benefit their dependency's accuracy. Representing individual analyses as nodes with a directed edge between n_1 and n_2 indicating that n_1 must execute before n_2, this graph forms a sequence or line.

Compilers have moved from this sequence or *line* model to a dependency-based DAG model to avoid the cumbersome explicit staging inherent to the line model. The widely used compiler backend LLVM[47] instead uses a "pass" model. The line model is essentially an ad-hoc form of LLVM's pass model. In LLVM's pass model, each pass declares what analyses it depends upon, allowing the LLVM pass manager to dynamically schedule the analyses and allow a configurable pipeline. Considering the graph representation again, LLVM's model is a DAG. The execution order selected by LLVM's engine would be one of the valid sequences a careful programmer could have picked in the line model.[1]

This DAG model works well over source code, but falls short when applied to compiled code. The binary analysis tool Jakstab [45] focuses on the co-dependency of value analysis and lifting. It takes the approach of running these two analyses to a fixed point, and provides a hook for user-defined analyses to run on each iteration. The graph of execution dependencies looks like a single cycle because Jakstab does not explicitly encode analysis dependencies. The Jakstab approach is undoubtedly a step forwards. However, it still causes unnecessary recalculation and places a form of monotonicity requirements on analyses in order to remain correct. A natural extension is to allow the expression of dependencies, as in LLVM, but also cycles, as in Jakstab. We investigate this full graph model of analysis management in this thesis.

We often want co-dependent analysis to detect bugs. Jakstab [45] is one example where simply integrating two analyses (value analysis and control flow recovery/disassembly) lead to better results than IDA[35] (the industry standard for reverse engineering) for jump resolution. The result is that an evaluation of a security property done over the CFG presented by IDA would miss some true positives along the undetected control flow edges. Specifying these analyses as logical rules within the same environment would provide the same power Jakstab found by integrating them for free. Investigations via the Doop[14] system for program analysis found that integrating exception analysis and points-to analysis gave them candidate target sets which were half the size on average for each object. Extra precision of this sort translates into reduced false positives when evaluating security conditions over the program.

Our main insight is that we need an analysis framework that allows for arbitrary cycles in the dependency graph. Further, we want a framework that is extensible so that adding new analyses is easier. Using a logic language is a natural approach to representing such a dependency graph.

Previous work in program analysis suggests that logic languages can help to structure

---

[1] Assuming a single evaluation thread.

this problem. Datalog has previously been successfully used to analyze programs [4, 14, 17, 46, 78]. Existing work has modeled a wide variety of properties from aliasing in compiled code [17] to security properties such as SQL injectability and cross site scriptability (XSS) [46] as facts in a deductive database. This work suggests a Datalog format as a potential common representation. Dataflow analyses are also representable [52] in this way.

We can even call out to code not written as logical rules[2]. This makes it possible to repurpose previously written analyses, or to write new analyses which may not be best represented as logic rules. There are still some restrictions on how such code can operate (for example, no visibly preserved state across calls) but taking this approach gives the flexibility required to be an integration system.

> **Thesis statement.** A Datalog derivative with extensions for negation and external callbacks can drive multiple potentially co-dependent analyses to reason about binary programs effectively.

Co-dependent Analyses. We examined the co-dependent analyses of control flow recovery (§ 2.1.1), value analysis (§ 2.1.2), and alias analysis (§ 2.1.4, § 6). Datalog fits well as a structuring tool for these analyses due to its incrementality and ability to handle mutual recursion.

Negation. In previous work, the framework runs an analysis "to completion" before proceeding. A later analysis can use the results of a prior analysis in two different ways. The first way is direct consumption of the facts provided by the analysis. In the case of a control flow recovery analysis, this would be treating a jump as possible because it is present in the returned CFG. The second way is a consumption of information on what was *not* returned by the analysis. Following the control flow example, the later analysis may assume that because the control flow analysis has completed, any edge not present in the recovered control flow graph cannot happen. In this way previous frameworks implicitly encoded *negated* information. Similarly, the a dataflow analysis producing an upper bound which increases throughout the analysis, such as a may-alias dataflow (§ 6.1.4), is not meaningful until it

---

[2] We use a system similar to external predicates using a common extension of logic language. §2.2.1

can no longer grow. As a result, the usability of a dataflow analysis implicitly depends on the *negated* information that no larger upper bound will be created, normally determined by waiting for the analysis to complete.

Co-dependence eliminates the ability to stage analyses since it may not be possible to know that an analysis will not continue later. As a result, the use of negated information must be made explicit (§ 4.2.4). Explicit negation also allows us to deal with more exotic control flow than "to completion". Our binary type reconstructor BiTR performs simultaneous constraint solving for type inference, with a goal of ignoring as few constraints as possible while still arriving at a consistent answer. Our form of negation is useful for describing that a particular solution is minimal (§ 3.3.6), and avoiding the computation of non-minimal solutions.

External Code. Datalog systems use external predicates to allow programmers to specify functionality outside the catalog programming language. Access to external code is important to allow the use of already existing analysis code, such as BAP [16] lifting and ELF parsing. External code also allows us to utilize traditional imperative data structures, such as hash tables, during the implementation of performance critical analyses (§ 6). We allow a more limited form of external predicates we call *callbacks*. The use of callbacks as opposed to external predicates allows us to more easily implement the engine (§ 4.3.2).

We investigated our thesis statement by designing and implementing Holmes, a logic language engine designed for the integration of binary analyses. BiTR (§ 3) inspired this design. For this purpose we invented a novel form of negation, based on circumscription [53] suited to this particular application (§ 4.2.4). This negation can further make progress from failed hypotheses (§ 4.2.5). This adaptation allows Holmes to resolve an output even in the case of cyclic negation. This is concretely the case in the use of VSA[11][3] to resolve function pointers or other indirect jumps. We also investigated different implementation approaches for Holmes (§ 4.3).

---

[3] Value-set-analysis is the current state of the art for performing range analysis for variables in binary code.

We present a type recovery mechanism for compiled code which inspired the Holmes design (§ 3). We define a *descriptive* type system, more powerful than direct C types for performing inference, wile being more flexible than a traditional prescriptive type system (§ 3.2). We address the problem of inconsistent type constraints by suggesting a resolution system which seeks a solution in which it drops the fewest constraints (§ 3.3). Finally, previous work in type recovery used a metric which minimized the importance of correct recovery of structure types. In § 3.4.1, we provide a novel metric for type recovery based on the probability of correctness of any individual query by a reverse engineer or downstream analysis. Throughout, we show how the design and implementation of BiTR motivates the features selected for Holmes.

We provide a specification for the Holmes logic language (§ 5). This enhances the ability to reason about the results of an analysis by constraining what derivations the analysis engine may follow. The specification also makes it possible to determine compliance for alternative implementations of the engine, potentially allowing programs to run on backends other than the initial one written for this thesis.

We show that Holmes can implement different sensitivities of alias analysis, and show how this affects practical use-after-free classification (§ 6). We implemented co-dependent alias analysis, use-after-free detection, and control flow recovery using Holmes. This implementation provides evidence for the feasibility of using a logic language as an integration layer for cyclically dependent analyses. This concrete bug finding tool demonstrates the applicability of the Holmes language to the domain of compiled-code program analysis. We translate analysis techniques from the compiler and program analysis communities into analysis of compiled code (§ 6.1). We measure the performance cost and precision benefits of different sensitivities of alias analysis by leveraging the modularity of the Holmes-based implementation (§ 6.3). We did not set out to invent a new or better alias analysis. Rather, we set out to demonstrate the suitability of Holmes as a platform for this type of analysis.

Chapter 2

Background

2.1 Binary Analysis

Questions of control flow, value analysis, alias analysis, and type recovery all have added challenges when analyzing compiled programs compared to analyzing source code.

Some of these challenges fall into the category of absent abstraction. Function locations, variable locations and bounds, variable types, and control flow graphs are all generally taken for granted even in the analysis of "low level" languages such as C, but are absent in binaries.

Another general class of added difficulty comes from the increased apparent state of a compiled program. A program with more state is more difficult to analyze. One of the features of functional code that makes it easier to analyze is its thorough reduction of state. Traditional procedural or imperative languages make this slightly more difficult through the use of state variables which expand the set of implicit inputs to each step greatly. Binary code is yet more difficult to analyze. It becomes quickly difficult to determine what parts of memory and registers are really live without abstractions like array bounds or variable scoping. It is difficult statically to know whether the process may access a given piece of state in the future. Execution effectively "leaks" state that existing analysis cannot determine will be unreachable. This increased state increases the problem size for alias analysis, value analysis, and type recovery, making it more difficult to find tractable approaches.

2.1.1 Control Flow Recovery

Recovering a complete control flow graph is a known difficult challenge in binary analysis. Algorithms for recovering any of the other missing abstractions depend strongly on having a pre-existing control flow graph[11, 33, 49, 51, 57, 60, 78]. However, determining the potential targets of an indirect (computed) jump becomes more difficult without the abstractions present in the source language. Despite this, real systems frequently use approaches which are both an over- and under-approximation due to the difficulty of this problem. These approaches usually produce representations which are useful when used directly by a human or UI. However, picking such an unsound approach can result in unsoundness in every downstream analysis

The goal of control flow recovery is to generate a complete graph of possible program counter transitions in the program while containing minimal number of spurious edges. The simple form of control flow recovery involves performing recursive descent disassembly from a given entry point and watching for a transition to a distinguished exit node (usually via a function return). However, indirect jumps due to function pointers, C++ virtual method calls, and even creative compilation of case statements can wreak havoc on this method since the value of the jump target to is not known ahead of time.

In each of the above cases, the compiler would be able to accurately determine the range of jump targets based on its own code generation information and information present in the source code. The type of a C++ object determines a small subset of legal function targets for a given method. The compiler can determine where jumps which came from switch statements will go as they have explicit targets in the source. Function pointers are the most difficult to resolve. At the source level, the compiler at least has the ability to restrict the possible targets to functions of the right type. Compilation strips away abstractions needed to perform any of these analyses. This renders them useless for direct use on raw compiled code.

2.1.2 Value Analysis

It follows that recovering over-approximations of the range of potential values for pieces of state in a program would be useful. One common way to do this is to perform abstract interpretation of the program over a domain designed to model the range of values well [11, 29, 57]. A program with more state is more difficult to analyze. One of the features of functional code that makes it easier to analyze is its thorough reduction of state. Traditional procedural or imperative languages make this slightly more difficult through the use of state variables which expand the set of implicit inputs to each step greatly. Binary code is yet more difficult to analyze. It becomes quickly difficult to determine what parts of memory and registers are really live without abstractions like array bounds or variable scoping. It is difficult statically to know whether the process may access a given piece of state in the future. Execution effectively "leaks" state that existing analysis cannot determine will be unreachable. This increased state increases the problem size of analyses such as alias analysis, value analysis, and type recovery, making it more difficult to find tractable approaches.

Unfortunately, Value Set Analysis[1] (VSA) assumes that the control flow graph has already known in order to sequence its transformations. Additionally, it can lose nearly all its precision by loops whose transformations are not well represented by its particular domain. The state of the art approach to the co-dependence of the control flow recovery and value analysis problems is to iterate control flow recovery followed by value analysis. This process allows the creation of new paths from new values, which in turn re-inform the value analysis [45].

2.1.3 Type Recovery

Another important static analysis is type recovery. A binary program may not necessarily have a traditional assignment of types to variables. However, source language requirements and programmer volition make it likely that there is a useful assignment of types. Even

---

[1] Value Set Analysis is the current state of the art for value analysis over compiled code.

assembly programmers and dynamic language users [36] usually use consistent types to some extent.

There are two primary sources for type information after the compiler strips abstractions: propagation from API/ABI boundaries and how the program interacts with the structure of the data. Propagation [67] of type information from code boundaries entails knowing a priori that a particular library or service in use by the program matches some signature, then following definitions of its arguments and uses of its results to annotate the program with this information. This technique is straightforward and keeps useful information like type and field names. Unfortunately, it encounters difficulties with any kind of internal data structure. Usage based analysis [49, 51] to perform better at recovering internal structures, and works by examining how the code defines and uses state and generating constraints for types based on that.

2.1.4 Alias Analysis

An alias analysis endeavors to answer the question "Do these two pointers point to the same thing?" There are two basic varieties: may and must alias. Must alias means that two values will definitely point to the same thing, but the lack of such a relationship means nothing. May alias means that two values might point to the same thing, and the lack of such a relationship means they definitely will not.

Additionally, alias analyses have different degrees of sensitivity. The sensitivity of an alias analysis refers to what additional parameters the analysis examines when asking whether two pointers alias. A flow-sensitive analysis uses the program counter or statement location as a parameter.[2] A context-sensitive analysis uses program call stack as a parameter. This term is also sometimes used to discuss the granularity of the memory model in use, e.g. a "field sensitive analysis" is one in which the model distinguishes between writes to `s->x` vs `s->y`. In our case, a lack of type information and the presence of pointer arithmetic adds

---

[2] This implicitly encodes the location within the control flow graph since the analysis is also conditioned on the particular compilation of the source program.

complexity to field sensitivity, so it differs from the traditional presentation (§ 6.1.8).

Analysis which is both context and flow insensitive is generally efficient, in nearly linear time [71]. However, their lack of sensitivity makes the drawing of variable boundaries important and removes all ability to reason about a variable being overwritten as a safety property. Common examples of this are Steensgaard [71] and Andersen's [6].

Flow-sensitive analyses are still more seldom used due to their longer run times, but modern techniques are beginning to allow them to scale to larger codebases [38]. Flow sensitivity is the most important sensitivity for our use case because it helps us to reason about overwritten variables. For example, if the program frees a variable, then overwrites it with a fresh pointer (for example, `free` followed by `strdup`), this avoids leaving the variable as potentially free. It is additionally specifically important to the binary domain due to the repeated re-use of registers. Over the course of a function, the register `RAX` probably maps to multiple different variables, depending on the current program counter. Flow sensitivity helps to keep these relationships separated by parameterizing the alias relationship accordingly. The implementation of a flow sensitive analysis generally follows the pattern of performing dataflow on a points-to relationship to a fixpoint.

Context-sensitive analyses are frequently used in analysis of Java and other object oriented programs because it helps to reason about which class of objects may have been the argument to a function, and thus which methods may be the target of the call. The two primary ways to accomplish this are to either take a call-site approach (tracking a stack of return addresses), or an object based approach where method calls take as a parameter which objects they may have occurred on. We will focus on the call-site approach in this thesis as not all the code under examination is object oriented, and we are not performing object recovery on the code which is.

The call-site approaches are generally distinguished from one another based on the domain used for the stack tracking. With an unbounded stack, it reduces to inlining every function call. This can result in an expansion of problem size, and cannot terminate in the case of recursion. If an analysis can handle the larger control flow graph, it can special case out recursion by either contracting strongly connected components of the function call graphs to single nodes, or by truncating stacks on recursive calls to the last time this call site occurred. Another option is to limit how much of the context the domain tracks. A common approach is to limit the domain by tracking only the most recent k calls for some fixed k. In this case, a strategy to deal with recursion is not required, but may still prove useful to put available precision to the best use possible.

One of the most focused on analyses for alias analysis over binaries is VSA [11]. VSA integrates the problem of alias and value analysis by doing abstract interpretation over a domain called a strided interval, with dynamic allocations appearing as free variables. This formulation has produced useful results in the past, but its relative expense [66] has limited its applications, especially with regards to whole program analysis. The authors of VSA also applied to variable recovery [12].

2.1.5 Dynamic Analyses

Dynamic analyses are those which focus on properties of individual executions, or statements of the form "there exists a trace with the property…". These analyses are frequently good at pinpointing problems or increasing the accuracy of approximations. However, being path oriented limits these techniques to describing what could happen, rather than what must always be the case.

Fuzzing

One common modern way to find issues in code is to simply perform a large number of executions with varied inputs. This technique is known as "fuzzing", and is widely used in the search for security flaws. Fuzzing is especially useful for analyzing compiled code as it does not rely on source-level abstractions. The primary two defining axes of fuzzing are blackbox [25, 39, 62] vs whitebox [10, 19, 32] and generational [25] vs mutational [39].

Symbolic Execution

A slightly more sophisticated (though more expensive) approach is to keep inputs "symbolic" while executing. Essentially, the evaluator uses expressions based on input variables in the place of concrete values unless the evaluator requires a concrete value to take a step. When a the evaluator encounters a conditional branch, it extends path constraint by conjunction with the condition. This restricts the range of feasible inputs for this execution. When the next instruction is a potentially dangerous operation (such as jumping to or dereferencing a symbolic value), a SMT solver can check a formula for the plausibility of the "bad" scenario[3]. The execution engine constructs the formula by taking the conjunction of the path constraint with whatever condition suffices to cause the badly behaved event. If the SMT solver returns a satisfying assignment to the formula, that solution describes a bug-causing input[9, 20]. If the formula passed is complete (the operation is *only* dangerous if the formula is satisfiable), an UNSAT response from the SMT solver give safety for *this path only.* Since loops may create an infinite number of paths, it is difficult to demonstrate that any region is safe this way. The major benefit of the symbolic execution and SMT approach is that it results in a concrete trace which exercises a bug condition.

2.2 Logic Language

Logic languages are a family of declarative programming languages which express computation as a series of rules used to derive a conclusion. This makes them well suited to the task at hand because it allows us to describe the provenance of information and to separate the execution strategy for a suite of analyses from the expression of said analyses. We will focus our review on Datalog and Prolog as exemplars of two styles of logic programming.

---

[3] SMT stands for Satisfiability Modulo Theory. It is a SAT solver whose variables have extra domains and additional operations. The SMT solvers discussed here operate over the theory of bitvectors.

2.2.1 Datalog

Basics

Predicates form the concept of a relation between multiple values. For example,

$$\mathbf{parent}(\cdot, \cdot)$$

represents a relation in which the left argument is the parent of the right argument. A “fact” is a particular member of a relation. We use italics to denote a constant. As an example,

$$\mathbf{parent}(alice, bob)$$

represents the fact that *alice* is *bob*’s parent. The set of facts known is sometimes referred to as the database. The database has two parts: extensional and intensional. The extensional portion of the database is those facts which the program author or user provides to the system prior to execution; the intensional database consists of those derived from the extensional database. In order to actually perform computation with this system and create such an extensional database logic languages use “rules” which function similarly to inference rules from traditional proof writing. To write rules, I’ll use **X** to represent a variable named x. Following in the vein of the previous examples,

$$\mathbf{sibling}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{parent}(\mathbf{P}, \mathbf{X}), \mathbf{parent}(\mathbf{P}, \mathbf{Y})$$

expresses that if for some assignment to **P**, **X**, and **Y**, we know that **X** shares parent **P** with **Y**, then we can derive that **X** and **Y** are siblings. The left hand side is the *head* and represents the template used to build new facts with this rule. The right hand side is the *body* and describes the search or match which must have a result on the database. An evaluation engine applies rules“to saturation”. This means that it applies all rules repeatedly until the body no longer has any new matches in the database.[4] Given this, rule declaration order is irrelevant. [5]

---

[4] Without external code, this is equivalent to “the head is in the database, or the body does not match“. With external code however, the head is not necessarily predictable, so we instead describe each rule triggering at most once per assignment of variables which matches the database.

[5] Rule declaration order is irrelevant to results. It may cause different performance in some engines.

Example: Uncle

If we ignore the genderedness of the term "Uncle", calculating from a series of parent/child relations all of the uncles is a straightforward example. We begin with an extensional database consisting of:

$$\mathbf{parent}(\mathit{alice}, \mathit{bob}) \quad \mathbf{parent}(\mathit{alice}, \mathit{charlie}) \quad \mathbf{parent}(\mathit{bob}, \mathit{deb}) \quad \mathbf{parent}(\mathit{bob}, \mathit{ed})$$

We take the sibling rule from the earlier example, and also

$$\mathbf{uncle}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{sibling}(\mathbf{X}, \mathbf{P}), \mathbf{parent}(\mathbf{P}, \mathbf{Y})$$

which indicates that someone is an uncle if they are that person's parent's sibling.

We can compute the complete database from these rules and the extensional database. The uncle rule cannot apply, since there are no facts for the sibling predicate. Applying the sibling rule to saturation yields a database of

$$\mathbf{parent}(\mathit{alice}, \mathit{bob}) \quad \mathbf{parent}(\mathit{alice}, \mathit{charlie}) \quad \mathbf{parent}(\mathit{bob}, \mathit{deb}) \quad \mathbf{parent}(\mathit{bob}, \mathit{ed})$$
$$\mathbf{sibling}(\mathit{bob}, \mathit{charlie}) \quad \mathbf{sibling}(\mathit{charlie}, \mathit{bob}) \quad \mathbf{sibling}(\mathit{deb}, \mathit{ed}) \quad \mathbf{sibling}(\mathit{ed}, \mathit{deb})$$

The right hand side of the sibling rule no longer applies, so the only option now is to execute the uncle rule. Doing so to saturation yields

$$\mathbf{parent}(\mathit{alice}, \mathit{bob}) \quad \mathbf{parent}(\mathit{alice}, \mathit{charlie}) \quad \mathbf{parent}(\mathit{bob}, \mathit{deb}) \quad \mathbf{parent}(\mathit{bob}, \mathit{ed})$$
$$\mathbf{sibling}(\mathit{bob}, \mathit{charlie}) \quad \mathbf{sibling}(\mathit{charlie}, \mathit{bob}) \quad \mathbf{sibling}(\mathit{deb}, \mathit{ed}) \quad \mathbf{sibling}(\mathit{ed}, \mathit{deb})$$
$$\mathbf{uncle}(\mathit{charlie}, \mathit{deb}) \quad \mathbf{uncle}(\mathit{charlie}, \mathit{ed})$$

At this point, the program has finished because neither rule applies, and so the program has finished. It has also derived the only two uncle facts possible.

Example: Related

Alternatively, perhaps we only wish to know if two people are related. Consider the rules

$$\mathbf{related}(\mathbf{X},\mathbf{Y}) \leftarrow \mathbf{parent}(\mathbf{X},\mathbf{Y})$$

$$\mathbf{related}(\mathbf{Y},\mathbf{X}) \leftarrow \mathbf{related}(\mathbf{X},\mathbf{Y})$$

$$\mathbf{related}(\mathbf{X},\mathbf{Z}) \leftarrow \mathbf{related}(\mathbf{X},\mathbf{Y}),\mathbf{related}(\mathbf{Y},\mathbf{Z})$$

We could structure the previous rule set as a two-step procedure: find the siblings, then find siblings of parents. This rule set is potentially more interesting because it is a recursive build up of information. However, this is not any more difficult than the previous query since Datalog applies its rules to saturation. The first rule translates a parent/child relationship into a "related" relationship. The second provides symmetry. The third provides transitivity. This is sufficient to compute the actual related set. Execution proceeds much the same as the previous example, simply firing each rule until no rules are able to fire. The only real difference is that the second and third rule will fire recursively.

Termination

It might be surprising to realize that the language as described thus far is actually terminating despite allowing recursive rules. Heads construct facts with variables or directly with values. If constructed with a variable, the body must bind that variable. If a body binds a variable to a value, that value must have been in the database. As a result, the only possible values are those which were in the extensional database and those mentioned concretely in the rules. With a finite domain of values and a finite list of predicates, there is a finite list of facts which any execution could instantiate.

Evaluation is a series of steps. Each step fires the first rule it can find which matches, or if no rule matches, terminates. Since this evaluation strategy is always productive or terminating, and we have established a finite upper bound on the database, we have a bound on the length of such an evaluation. This evaluation strategy matches the saturating/fixpoint semantics of Datalog because it only produces facts according to the rules and only stops when no rules apply.

There are some important caveats to termination as a language feature. Termination directly implies a limit to the language's power. Datalog's inability to create new symbols can be limiting. Unmodified Datalog can still implement a number of different kinds of closure or search procedures, but even simple arithmetic is outside of its purview if not within a bounded domain. Some language modifications (including those needed for Holmes) will introduce the ability to generate new symbols. When this happens, the resulting language loses its termination guarantees.

External Predicates

One example of a case where introducing new symbols might be desirable is the ability to perform arithmetic. It is possible to encode arithmetic over some finite domain (e.g., $0 - (2^{64} - 1)$) by adding a predicate for each binary operation and prepopulating the extensional database with all the values. However, having such a table is massively impractical.

One way to deal with this issue is to introduce external predicates: predicates whose truth is not defined by presence in the database. Instead, the engine computes or queries its truth from some external system when needed. This leads to the need for a mode system. Consider an external predicate $\mathbf{sum}(\cdot, \cdot, \cdot)$, where the first entry is the sum of the other two. When trying to match the body of a rule, it becomes important that at the match algorithm sends at least two values to the external code; if only it only sends one, the resulting match set will have the enormous size of $|\mathbf{D}|^2$ where $\mathbf{D}$ is our domain set. In the case of functions designed to be hard to invert (e.g., a hash function), this becomes not only a result size concern and an actual implementability concern. To deal with this, external predicates can have "mode signatures". These signatures explain which entries are logical inputs and which are outputs. For example, we could have

$$\mathbf{sum}(-, +, +) \quad \mathbf{hash}(-, +)$$

A mode checking phase on the rule can then guarantee that matches against the sum predicate only needs an implementation of

$$\text{add} : \text{uint64} \to \text{uint64} \to \text{uint64}$$

rather than needing to assume the presence of subtraction and possible-addends operations.

Unfortunately, adding external predicates adds all possibilities for their result values to the upper bound for the termination. Adding an external predicate which only produces a small type such as a boolean has little effect on our ability to bound execution. Adding a finite-but-large type like `uint64` makes any numerical approximation of runtime from value count irrelevant, but still technically gives termination. Adding a type with non-finite size like lists or strings *breaks termination.* As a simple example, consider an external predicate which appends the character 'a' called 'extend'. Then

$$\mathbf{infinite}(\mathbf{X}) \leftarrow \mathbf{infinite}(\mathbf{Y}), \mathbf{extend}(\mathbf{Y}, \mathbf{X})$$

$$\mathbf{infinite}(``")$$

provides a simple infinite loop by extending the string with more 'a's forever.

Backwards Chaining

The evaluation model discussed thus far is what is commonly known as forward chaining, or bottom-up search. This approach searches for a proof by taking known facts and deriving anything they can hoping to reach the goal. However, there are some applications for which this approach does not work well — as an example, any system that would have a potentially infinite intensional database would never complete execution with such a semantics.

Prolog uses the opposite approach: backwards chaining or top-down search. In backwards chaining, a particular goal match directs the search rather than enumerating all possible derivations from available information. The logic engine examines the goal and the available rules and breaks it down via unification to a series of potential subgoals, then recurses.

2.2.2 Negation

Stratified Negation

One interpretation of negation is as the inability of the system to prove a particular fact. For example, this sort of negation could be useful to describe routing. We can describe network reachability as:

$$\mathbf{reachable}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{link}(\mathbf{X}, \mathbf{Y})$$

$$\mathbf{reachable}(\mathbf{Y}, \mathbf{X}) \leftarrow \mathbf{reachable}(\mathbf{X}, \mathbf{Y})$$

$$\mathbf{reachable}(\mathbf{X}, \mathbf{Z}) \leftarrow \mathbf{reachable}(\mathbf{X}, \mathbf{Y}), \mathbf{reachable}(\mathbf{Y}, \mathbf{Z})$$

Next, our machine has a local network and a gateway — a common configuration. Traffic serviceable by the local network should be directly sent, while the remainder of routable traffic should go to the gateway.

$$\mathbf{route}(\mathit{local}, \mathbf{T}) \leftarrow \mathbf{reachable}(\mathit{local}, \mathbf{T})$$

$$\mathbf{route}(\mathit{gw}, \mathbf{T}) \leftarrow \neg\mathbf{reachable}(\mathit{local}, \mathbf{T}), \mathbf{reachable}(\mathit{gw}, \mathbf{T})$$

In the case of this set of rules, negation has a clear interpretation. First, generate all reachability predicates. Then, if reachable was not derivable thus far, matches against not reachable should succeed. However, recursive rules and negation can mix to cause termination and stability issues. Consider the set of rules:

$$\mathbf{p}(\mathbf{X}) \leftarrow \neg\mathbf{q}(\mathbf{X})$$

$$\mathbf{q}(\mathbf{X}) \leftarrow \neg\mathbf{p}(\mathbf{X})$$

There is no interpretation of this which terminates with the fixpoint semantics, or which provides a minimal model [72]. To deal with this, Datalog imposes a requirement on negation known as stratification. It is possible to phrase this requirement as a graph property:

Create a graph where the predicates are nodes with an edge between two nodes iff there is a rule in the program which uses the source node predicate as a premise to derive the destination node predicate. Additionally, label all those edges where the with a negated premise predicate. There must be no cycles which include one or more negated edges.

2.2.3 Circumscription

Circumscription is a concept in logic related to including Occam's Razor into deriving a model from logic statements. We discuss it here because it will be relevant to the Holmes treatment of reasoning from incomplete information.

As an example of why circumscription might be useful, consider the following: There are three cups marked a, b, and c. There are balls in cup a, and at least one of cups b and c. This encodes to:

$$a \wedge (b \vee c)$$

We know the above formula is true, but not the truth values for a, b, and c, there are three possible satisfying assignments:

$$(\text{true}, \text{true}, \text{false}), (\text{true}, \text{false}, \text{true}), (\text{true}, \text{true}, \text{true})$$

Circumscription is a mode of reasoning that in this situation allows us to examine the first two options rather than the third. It is about picking the minimal situation that could correspond to what we know.

McCarthy's paper introducing the idea [53] gives more examples. A slightly simpler (if imprecise) way of thinking about circumscription is to consider it as taking a specification of truth which is irrefutable, while interpreting a minimal set of facts as true — conclusions are not necessarily grounded in fact, but rather are statements without refutation.

Original Definition

Originally, McCarthy defined circumscription with respect to first order logic. Specifically, the circumscription of a predicate P to a predicate Q in a formula A is

$$(A[Q/P] \wedge \forall x.Q(x) \rightarrow P(x)) \rightarrow (\forall x.P(x) \rightarrow Q(x))$$

McCarthy calls this a “sentence schema”. Essentially, this says that for a given formula A we know to be true, and some predicate P we have incomplete information about, if some other (usually artificially defined) predicate Q could replace P in A without changing its truth value, and P is true whenever we know Q to be true, then we can have the other half of the implication, making Q a legitimate proxy for P. Practically applying this usually involves declaring a Q which is true in exactly those situations we know P to be. As a simple example, consider the formula

$$P(0) \wedge P(1) \wedge P(2)$$

Define a predicate for circumscription Q as

$$Q(x) \equiv (x = 0) \vee (x = 1) \vee (x = 2)$$

essentially explicitly enumerating the cases in which P must be true. Circumscribing P in the example formula using Q yields

$$P(x) \rightarrow (x = 0) \vee (x = 1) \vee (x = 2)$$

This transformation gives more information about P than the original formula. Specifically, we now know that $\neg P(4)$.

Relation to Logic Programming

Circumscription is directly connected to modern logic programming through the negation features present in some logic language dialects. We express the database and rules as a single logical formula by encoding as a conjunction of known facts as predicates applied to concrete values and rules encoded as implications. Circumscribing over this formula will yield results consistent with a given logic program. It will also allow us to potentially derive more by giving us an explicit definition of a predicate. Circumscription provides the theoretical underpinnings for stratified negation. For example, take the system

$$\mathbf{square}(w) \quad \mathbf{square}(x)$$

$$\mathbf{silver}(x) \quad \mathbf{silver}(z)$$

$$\mathbf{circle}(\mathbf{X}) \Leftarrow \neg\mathbf{square}(\mathbf{X})$$

$$\mathbf{copper}(\mathbf{X}) \Leftarrow \neg\mathbf{silver}(\mathbf{X})$$

$$\mathbf{penny}(\mathbf{X}) \Leftarrow \mathbf{copper}(\mathbf{X}), \mathbf{circle}(\mathbf{X})$$

If we ask “Is y a penny?”, i.e. ?$\mathbf{penny}(y)$, the system without circumscription cannot answer yes. The system cannot derive that y is circular because it neither knows that a priori, nor does it know that it is non-square. Circumscribing over both $\mathbf{square}(\cdot)$ and $\mathbf{silver}()$ in the system allows us to derive $\neg\mathbf{square}(y)$ and $\neg\mathbf{silver}(y)$. In this way, circumscription forms the basis for the form of negation present in logic languages.

Chapter 3

Binary Type Recovery (BiTR)

First, we examine the problem of binary type recovery without Holmes. We designed this system prior to Holmes, but gave the inspiration for most of its features, as we discuss along the way.

3.1 Introduction

Type information is not used by the CPU, and as such, the compiler throws it away during normal compilation. This information has a wide variety of uses, including software verification, reverse engineering, and binary similarity detection. Reverse engineering relies heavily upon the reconstruction of types to make the lifted code comprehensible to a human. Similar data structure use can indicate similar code [21]. Knowledge of types can also assist in fuzzing and automated decompilation. All in all, many methods of analyzing

	Locals	Structures	Polymorphism	Recursion	Variable Recovery Ind.	Kind
TIE	●	◐	○	○	○	Static
Hex Rays	●	○	○	○	○	Static
Rewards	○	○	○	○	●	Dynamic
Howard	●	●	○	○	●	Dynamic
BiTR	●	●	●	●	●	Static

Table 3.1: Feature Matrix

binaries post-compilation require some degree of type construction to work properly, and better automated type reconstruction can help with this.

The *binary type reconstruction* problem is: Given a binary program without types, reconstruct the type information that the compiler had at code generation time, but did not emit with the final binary. We specifically focus on C, but hope that extensions to our techniques can eventually allow them to work on larger classes of languages. Effectively, the goal is to recover an abstraction stripped away by the compiler as it was emitting code. This is distinct from traditional type inference in two important ways: First, even if the code does not have a valid type assignment, it is important to succeed in assigning meaningful types to most variables. Second, operations with the same concrete implementation are not differentiated, e.g. adding numbers vs indexing into an array, and this differentiation must occur during inference.

Type reconstruction has received increased attention in recent years because it is an important activity for reverse engineering COTS (Commercial Off The Shelf) software. Recent work includes TIE [49], the Hex Rays Decompiler [35], Rewards [50], and Howard [68]. At a high level, the goal of each tool was the same: recover high-level type information from low-level code. However, the scope and fidelity of each tool has varied considerably.

Table 3.1 shows an overview of current research and the scope of reconstruction performed. Rewards proposed type propagation from known type sources, such as system and library calls, to type sinks in a dynamic trace, and using this information to type the cells in a memory image. Howard, Hex Rays, and TIE all also do type propagation. Hex Rays and Howard add the ability to infer local variables, which they then label with propagated type information. TIE adds the ability to infer variables not used in a trace, along with a type range. TIE also infers general structural types. Hex Rays and Rewards do not infer any structures, they simply propagate information from known sources. Howard infers structure fields and arrays accessed during a dynamic trace, but cannot handle recursive definitions such as a linked list [68]. In practice, however, TIE outputs structure types rarely, and in our experiments, only inferred a single field. There are also systems for type forensics, where the analyst is generally given a set of types and the goal is to match them to compiled code and/or a memory image. We discuss this work in § 3.5. Static approaches have the benefit of typing all the visible code in the binary, while dynamic approaches have the benefit of knowing true address information as memory accesses occur. When reverse engineering, one often wants to look at an area of code first to decide whether it is worth exploring, rather than needing to find a way to reach that code before considering its nature. This makes the dynamic approach less well suited to the task at hand.

Unfortunately, current techniques left large amounts of code untyped. In particular, previous work did not infer polymorphic types. In C, these are often implemented by casting to/from `void*`. Nor did previous techniques infer recursive types, such as linked lists. Specifically, Hex Rays, Rewards, and Howard did not make type inference a central goal, but instead focused on type propagation. TIE *inferred* types in addition to doing propagation. It did this by first accumulating a set of constraints for typing based on how code used and defined variables. Next, the type constraint solver attempted to unify the constraints into a typing solution. The main limitation of TIE was the constraint solver was not powerful enough to include polymorphic or recursive types, and did a poor job coming up with precise answers for structures.

When propagation from sources or sinks cannot determine the type of a register or memory cell, reverse engineering the type comes down to analyzing how the program uses and defines the register, and then finding a typing that is consistent with all uses. The general approach to solving this mechanically is to first formulate a system of constraints from the code, then solve them by some mechanism. There are a large number of ways to approach this: Are variables constrained, or are registers and memory locations? Should we represent aliased operations like pointer arithmetic and addition by intersection-typed functions, or multiple applicable rules? How does the solver prioritize possible branches of a typing to explore?

In this chapter we present BiTR, a type recovery system for compiled code that handles the full range of C data types excluding unions. Like TIE, we take a type inference approach where analysis first generates constraints on the types of variables, which are then solved to an upper and lower bound on its type. The main difference is that our constraint solver uses a more powerful constraint and type framework, avoiding issues plaguing previous work. We observed that three main design decisions hampered TIE in particular. First, TIE's constraint solver attempted to brute force a solution to typing constraints. BiTR introduces the notion of consistency of constraints, which we use to intelligently prune inconsistent solutions during type resolution. Second, TIE's use of structure subtyping obscures the link between different pointers in a way that makes it harder for their constraint solving engine to recover structures. Third, TIE required variable recovery before type inference, thus tying overall performance to the variable recovery engine. Often a reverse engineer only wants to know the type of a memory cell, as in Rewards. In such cases we need not discern the type of every variable. BiTR decouples variable recovery from inferring types.

To test our approach, we implemented our techniques in BiTR and measured the accuracy of our results. We type approximately twice as many more locations (16.8% in TIE vs 36.78% in BiTR), and do so more accurately. Further, we generate types about five times faster than TIE.

In this chapter we show the following specific results:

- We provide a *descriptive* type system with *structure*, *recursive*, and *polymorphic* types (§ 3.2.2), designed for inferrability, expressiveness, and specificity.
- We develop an efficient type inference engine. Our main observation is that early branch pruning greatly speeds up the search of the typing space (§ 3.4.6).
- We develop techniques that show variable recovery is not required in order to do type reconstruction, assuming the inference system has sufficient capability to deal with structures (§ 3.3.2).
- We implement our technique and show that we can correctly reverse engineer twice as many types as TIE (§ 3.4.3).

§ 3.2 describes the type system used to describe the types of registers throughout the code. § 3.3.1 shows a worked example of our inference methodology. § 3.3 describes how the system works mechanically, building on the intuition from the worked example. § 3.4 presents our experiments and results, evaluating the effectiveness of BiTR and explaining in detail why we got the results we did, and subtleties in the experiments.

3.2 Type System

We approach the problem of recovering types from compiled code with the goal of recovering the type of every register at every program point. The main complication here is that there are operations with multiple possible meanings, and we must discover which one is actually in use. For example, in the case of addition, the operation may mean any of array indexing, struct indexing, or arithmetic addition. There may be multiple legal interpretations available in some cases. We design our type system to try to restrict the number of legal interpretations as much as possible, while still accepting real-world programs generated under a C-like paradigm.

To this end, we set out to form a descriptive type system. This means we primarily focus on describing what actions are actually taken in practice; we focus on the property that if what the code is doing is reasonable, we accept and characterize how that could be the case, rather than rejecting the code outright because it does not fit our type system. We must take an approach along these lines, because unlike in a compiler, we cannot simply reject the code if the binary fails to make clear its own safety. Before diving into the type system itself, we examine some of the decisions made when designing a type system for type reconstruction, and why we made the decisions we did. Then, we explain the type system used by BiTR internally to describe and infer types before BiTR projects the internal types out to C-like types upon completion.

3.2.1 Design Space

To describe the types of the registers in compiled programs, we have a wide variety of possibilities to consider. Along one axis, we can try to be strict or permissive in our typing. The benefit of greater strictness here is that we can claim more properties about the typed code. However, strict type systems have the downside that if we receive code that does something outside the small window of proof techniques we have envisioned, we will suddenly be getting little to no information about the binary. Adding the ability to perform more operations (for example, array indexing) to the type system is usually costly in terms of complexity of inference implementation and in terms of speed. Permissiveness has the upside of being willing to skip some of these complex operations (for example, trying to prove that an array index is in bounds and a multiple of the size) and of being more likely to give usable output even if a proof strategy fails. However, permissive type systems also have the downside that there may be more typings available for a given program. Additionally, a satisfying typing in such a system will be unlikely to provide useful properties about the program.

The specificity of our type system lies along another axis. The more features we add, the more complex the inference will become, but also the more useful the produced typings will be. If we are strict, the specificity governs what we can and cannot type. If we are permissive, this governs how well we will be able to use the information present in the code. An example of specificity would be whether we have an all encompassing *code pointer* type versus one that encodes some kinds of input and output types, or in the extreme case, preconditions and post-conditions on the values. Expressiveness instead deals with those things that we would simply be unable to even talk about otherwise. For example, without some kind of recursive type, all data structures would need to have a finite depth known ahead of time.

In our environment, permissiveness has an advantage over strictness. A strict typing system would provide much value as an analysis framework, but if we want to work generally on the bulk of programs, a strictly safe environment would exclude too many of programs from analysis. Along expressiveness and specificity, we make compromises in order to keep the system tractable. In the area of specificity, we look to differentiate regular pointers from arrays, and to allow for the description of polymorphic data structures. In expressiveness, we add support for recursive types. To the best of the authors' knowledge, we are the only reconstruction system to support polymorphic types, and one of two to support any form of recursive types [26]. Using our approach, we can infer recursive types for free. We also support polymorphic types through appropriate use of subtyping with a similar method. The differentiation of pointers from arrays is a minor point theoretically, but in practice it can be convenient to know whether a variable points to a single cell or multiple. In order to avoid complex dependent typing, we ignore array lengths and any form of proof that a computed value has some property.

3.2.2 BiTR Type System

The BiTR type system expresses the core reasoning concepts used by the tool to decide what type to assign to a register or expression, and what range of types it is considering. In Figure 3.1 we show the grammar for defining the types.

Types. The int_w type represents integers of width w. Our intermediate semantics language (BIL) represents memory writes as non-destructive updates to a global array. As such, we need a type for that array: mem. This type can be directly inferred from the source language, as a value of type mem is only created by performing a write to an existing mem, and that operation is unambiguous. It does not correspond to anything in C or other source language, it is just to type our memory variable. To represent flags or other single bit registers, we use the bool type. Technically, we could type these values with single-bit unsigned integers. However, we wish to distinguish between variables we perform arithmetic on versus those we apply boolean operations to, since we are trying to recover abstractions.

$\top$, $\bot$, $\top_w$, and $\bot_w$ are synthetic types for use in subtyping bounds and to complete our lattice. $\top$ and $\bot$ are universal top and bottom for all our types. $\top_w$ and $\bot_w$ are provide

$$
\begin{array}{rllll}
\tau ::= & \mathsf{int}_w & \mid \mathsf{mem} & \mid \mathsf{bool} & \\
 & \mid \top & \mid \bot & \mid \top_w & \mid \bot_w \\
 & \mid rv^*@o & \mid rv[\cdot]@o & & \\
 & \mid \mu A.\tau_A & \mid \forall \tau \trianglelefteq A \trianglelefteq \tau'.\tau_A & & \\
 & \mid \mathsf{code} & & &
\end{array}
$$

$$
\begin{aligned}
\rho &::= (o : A)^* \\
\tau_A &::= \tau \text{ which may use the type variable A} \\
w &::= \text{any positive natural number} \\
 &\quad \mid s \text{ size of pointer} \\
A &::= \text{type variables} \\
rv &::= \text{region variables} \\
o &::= \text{any integer}
\end{aligned}
$$

Figure 3.1: Type Grammar

-8	code
0	$x^*@n$
4	int_{32}

Figure 3.2: An Example Stack Region named x

a top and bottom bound for types of size w. This allows us to use subtyping bounds to indicate knowledge of the size of something, as given by the width of a write or similar low level clues.

Region variables (and the regions they represent) are specific to our particular system. The grammar represents regions as ρ. Regions are a collection of mappings from offsets to type variables. The regions form the basis for our inference of structs. Figure 3.2 shows the region corresponding to the stack for a function with one integer stack variable. Note that when combined with an offset, the same region can be represent the stored base pointer (if the function is a leaf function, otherwise the base pointer must be abstract), the initial stack pointer, and the stack pointer right before return. In another example, we can represent a structure representing a sized string

```
struct {
  int size;
  char* str;
}
```

in BiTR as $r_0^*@0$ where

$$r_0 = (0 : t_0)(4 : t_1) \qquad t_0 = \mathsf{int}_{32} \qquad t_1 = r_1[\cdot]@0$$

$$r_1 = (0 : t_2) \qquad t_2 = \mathsf{int}_8$$

At first, this might seem a clumsy way of dealing with structures, but it has a specific important benefit — it can express that two pointers point at the same kind of structure, or that two structure members point at the same type, even when that structure definition or type information is incomplete. This allows information from different interactions with the same structure to naturally propagate into all uses.

$rv^*@o$ and $rv[\cdot]@o$ represent pointer and array types respectively. The rv indicates which form of region the pointer points at. The o acts as an offset into that region, allowing two pointers at a constant offset from one another to refer to the same region. Adding a constant value to either a pointer or an array can perform an indexing into the region, accessing one of the struct fields. Arrays can have variable values or constant values added to them to index into the array, keeping the offset constant. Were we to attempt to detect array bounds, we could describe $rv^*@o$ as $rv[\cdot]@o$ with size 1. However, as we allow arbitrary indexing into arrays, but not into pointers, it is useful to distinguish between the two.

$\mu A.\tau_A$ introduces a recursive type. Within τ_A, A refers to τ_A. Since our type system does not have additive types as a primitive, the only useful place to put A is inside a region that a pointer uses. This provides an implicit option type (our pointers are nullable), allowing the recursive types to be finite size. This corresponds to the common C usage of using a struct pointer as a member of the struct itself.

$\forall \tau \trianglelefteq A \trianglelefteq \tau'.\tau_A$ constructs a polymorphic type. τ is the lower bound, and τ' the upper bound. Our inference system does not directly generate polymorphic types. It instead generates a network of bounding constraints between type variables, possibly including cycles or partially constrained types. When the value of a specific type variable is requested, partially constrained types are universally quantified, and cyclically constrained types are quantified via μ. This allows for the expression of polymorphic linked lists and other simple data structures. This allows us to deal with some cases where C programmers use `void*` and casts to deal with their lack of polymorphism. The upper and lower bounds are most commonly used in conjunction with $\top_w$ and $\bot_w$ to indicate a type variable for which only we only know its size, but in principle could express other restrictions.

`code` represents any pointer to code that is a valid jump target. In future work, we may attempt to type this more specifically, expressing type preconditions for the jump and treating it more like a continuation. Function pointers, and values created by `setjmp` use this type. However, its most common and practical use is to represent the return pointer the function jumps to in order to return control to its caller.

Subtyping An important feature of the system is subtyping — this allows us to constrain the bounds of a type even when we do not know everything about it yet. We define subtyping via meet: if $\tau_0 \wedge \tau_1 = \tau_0$, then $\tau_0 \sqsubseteq \tau_1$.

$$\frac{}{\tau \wedge \tau = \tau} \qquad \frac{}{\top \wedge \tau = \tau} \qquad \frac{\tau' \wedge \tau = \tau''}{\tau \wedge \tau' = \tau''} \qquad \frac{\mathrm{sizeof}(\tau) = s}{\top_s \wedge \tau = \tau}$$

$$\frac{\mathrm{sizeof}(\tau) \neq \mathrm{sizeof}(\tau')}{\tau \wedge \tau' = \bot} \qquad \frac{\mathrm{sizeof}(r^*@o), w}{r^*@o \wedge \mathsf{int}_w = r^*@o}$$

$$\frac{\mathrm{sizeof}(r^*@o), w}{r[\cdot]@o \wedge \mathsf{int}_w = r[\cdot]@o} \qquad \frac{r@o = r'@o'}{r[\cdot]@o \wedge r'^*@o' = r[\cdot]@o}$$

$$\frac{r@o \neq r'@o'}{r^*@o \wedge r'^*@o' = \bot s}$$

$$\frac{r@o \neq r'@o'}{r[\cdot]@o \wedge r'^*@o' = \bot s} \qquad \frac{r@o \neq r'@o}{r[\cdot]@o \wedge r'[\cdot]@o' = \bot s}$$

$$\frac{\text{No above rules apply} \quad \mathrm{sizeof}(\tau) = \mathrm{sizeof}(\tau') = w}{\tau \wedge \tau' = \bot_w}$$

$$\frac{\text{No above rules apply} \quad \mathrm{sizeof}(\tau) \neq \mathrm{sizeof}(\tau')}{\tau \wedge \tau' = \bot}$$

We omit structural subtyping here. This is an intentional omission, with the rationale that casting from one compatible struct to another is an uncommon operation in C, and removing the possibility of such a cast allows better propagation of information about the structures by demanding that the unification of regions to proceed.

Expressions. We also require rules explaining expression typing. We omit some of the more obscure rules dealing with different types of casting bitvector concatenation and slicing for brevity.

$$\frac{n\ \text{const} \quad e \sqsubseteq r^*@o}{r, o - n^*@ \sqsubseteq e + n} \qquad \frac{n\ \text{const} \quad e \sqsubseteq r[\cdot]@o}{r, o - n[\cdot]@ \sqsubseteq e + n}$$

$$\frac{e' \sqsubseteq \mathsf{int}_s \quad e \sqsubseteq r[\cdot]@o}{r, o[\cdot]@ \sqsubseteq e + e'} \qquad \frac{e \sqsubseteq r^*@o}{*e : r@o(0)} \qquad \frac{e \sqsubseteq r[\cdot]@o}{*e : r@o(0)}$$

$\oplus$ here is a substitute for most mathematical operations (+, -, etc), and ϕ is an SSA ϕ node.

$$\frac{e : \mathsf{int}_w \quad e' : \mathsf{int}_w}{e \oplus e'} \qquad \frac{e \sqsubseteq \tau \quad e' \sqsubseteq \tau'}{\phi(e, e') \sqsubseteq \tau \vee \tau'}$$

$$\frac{e \sqsubseteq r^*@o \quad e' : r@o(0)}{*e = e' : \mathsf{mem}} \qquad \frac{e \sqsubseteq r[\cdot]@o \quad e' : r@o(0)}{*e = e' : \mathsf{mem}}$$

Type and Region Variable Binding. In a traditional type system, the program binds variables before use. However, as this system was primarily designed for inference rather than direct use, it assumes initially that all region variables and type variables may be mutually recursive. A set of bindings for type variables and region variables which satisfies the constraints we will describe later forms the solution to our typing problem. However, this system of a mess of mutually recursive bindings is difficult for humans to read and understand, so when the user asks for the binding for a given type variable, we narrow the scopes of type variables as much as possible, introducing $\forall$ and μ where appropriate, while substituting in region variables for their bindings, making a self-contained type. The polymorphic types arise from type variables who are insufficiently restricted in the response type. Recursive types arise from type variables whose bounds refer to themselves.

In summary, during inference, all region variables and type variables are potentially mutually recursive and exist together. When the type is output, μ and $\forall$ bind new type variables, and region variables do not exist as we substitute them with the regions they represent.

3.2.3 Approach

In order to actually generate types according to this model, we first lift all the statements to BIL[16] (an IL for modeling CPUs used by BAP) to make them easier to analyze. Next, BiTR generates a set of subtyping constraints for each statement, restricting the types that each register could have. Finally, we search the constraint space for a maximally correct solution, generating a narrow range of types. Unfortunately, aspects of our typing system, namely ad-hoc polymorphism, subtyping, and equirecursion, do not coexist in any exiting unification system the authors could find, so the authors wrote a new one to solve the constraints.

3.3 Inference Method

There are two major components in the inference of types for a piece of code. First, we generate type constraints based on the action of the code itself, with every update to a register assigned its own type variable. BIL statements which have multiple possible meanings (an add that could be a numeric or a pointer operation for example) generate a disjunction constraint. Then, we solve these constraints, and use the now-known types for each type variable to form the solution. The constraint generation phase occurs on SSA-form BIL, as generated by BAP [16]. This allows the constraint generation to use the use-def information built into SSA, allowing separate constraint generation for each statement. The constraint solving is the more difficult part, and uses an extended form of unification in order to transform the constraints into a set of conservative conditions on the type of a given register definition. The choice of which constraint to satisfy in each disjunction effectively specifies what operation would a decompiler would select in the translation to a typed language

```c
struct list {
  struct list* next;
  int v;
};

int loop_sum(struct list* l) {
  int v = 0;
  while (l != NULL) {
    v += l->v;
    l = l->next;
  }
  return v;
}
```

Figure 3.3: Loop Sum C

without intersection types [40, 65][1].

3.3.1 List Summation Example

First, we demonstrate BITR in action. We use a `loop_sum` function, shown in Figure 3.3, which sums the elements in a list. Figure 3.4 shows the disassembly of the compiled code. We have replaced all the addresses with an incrementing label to enable us to reference instructions more easily in this example. First, we lift the compiled code to BIL, then lift that to SSA, and finally run simple optimizations on the result to make it more readable. This leaves the code as in Figure 3.7(Appendix), but without the annotations. This diagram also contains an embedded disassembly showing where the code came from.

The first thing a reverse engineer would do is to identify that the program uses `rsp` as the stack, see the standard function prologue in instructions 00 to 01, and skip past the prologue. At this point instruction 02 loads `rdi` into stack slot $-0x18$, and remember that this stack slot has a type corresponding to the first argument of the function, assuming that normal AMD64 calling conventions are in use. Instruction 03 then initializes stack slot $-0x4$ to 0. However, since the write is 32 bits wide and not a 64-bit wide write, we immediately

---

[1] Intersection types represent a more general type for values which can have multiple possible types which are not partially ordered on the subtyping lattice.

```
loop_sum:
00:        push   %rbp
01:        mov    %rsp,%rbp
02:        mov    %rdi,-0x18(%rbp)
03:        movl   $0x0,-0x4(%rbp)
04:        jmp    11
05:        mov    -0x18(%rbp),%rax
06:        mov    0x8(%rax),%eax
07:        add    %eax,-0x4(%rbp)
08:        mov    -0x18(%rbp),%rax
09:        mov    (%rax),%rax
10:        mov    %rax,-0x18(%rbp)
11:        cmpq   $0x0,-0x18(%rbp)
12:        jne    5
13:        mov    -0x4(%rbp),%eax
14:        pop    %rbp
15:        retq
```

Figure 3.4: Loop Sum Assembly (64-bit)

know the stack slot $-0x4$ is not a pointer. So, we assume stack slot $-0x4$ is an integer. Next, following the jump from 04 to 11, 11 compares 0 against stack slot $-0x18$, the one that contains the first function argument.

Along the inequality case in 12, the next instruction is 05. This branch dereferences slot $-0x18$, adds 8, and dereferences it again. Now we know that the input must be a pointer, with *something* at offset 8. Finally, we note that the width of the read is 32 bits, so we again have the situation from earlier — this is an integer. At 07, the code adds the value read out back into stack slot $-0x4$. 08-09 dereference slot $-0x18$, so we know that in addition to pointing at something at offset 8, stack slot $-0x18$ also points to something at offset 0. Instruction 10 moves that value into stack slot $-0x18$! We now know that in addition to having a 32-bit integer at offset 8, at offset 0 stack slot $-0x18$ will have a value the same type as itself. If we assume the branch where our new pointer is null as we traverse 11 and 12 this time, we see that the value at stack slot $-0x4$ is the one returned. As a result, we now know that this function takes a linked list of integers as an argument, and returns an integer.

Now, this is roughly how a human would solve the problem. There are heuristics (e.g. this branch is feasible because the value is nonzero, then the code dereferences this variable, so this variable is a nullable pointer), shortcuts (ignore the function prelude), and tricks (e.g. treating stack slots as variables) that the human used which are not ideal for automated analysis. Branch feasibility is potentially expensive, function preludes vary across compilers and options, and stack-slots-as-variables can break when pointer math uses stack addresses. However, there a number of tricks in that narration which are relevant to computers. For example, when we found something out about a value in a register, and the register came from stack slot $-0x18$, we would not just assign the property to that register, we would also assign the property to the stack slot, and the function argument. This process that enabled us to discern that the struct had at least two accessed fields. Additionally, we noticed that there was dataflow from a variable into itself, and used this to realize that the input variable must contain a recursive type.

We will now approach this in a more mechanized way, similar to how our reconstruction system would attack the problem. We omit some steps for brevity, but the reasoning will follow the same basic procedure. Assume for notation that $\tau(v)$ is a function that grabs the type variable corresponding to a variable. First, we go about generating constraints for each BIL statement. For example, when we subtract 8 from the initial `rsp`, we have two possibilities — either `rsp` was an integer, in which case we're doing arithmetic and would want to generate

$$(\mathsf{int}_{64} \sqsubseteq \tau(\mathrm{R\_RSP\_211})) \wedge (\tau(\mathrm{R\_RSP\_328}) \sqsubseteq \mathsf{int}_{64}))$$

However, if we have a pointer, we instead want something along the lines of

$$(\tau(\mathrm{R\_RSP\_328}) = 0^{*}@0) \wedge (\tau(\mathrm{R\_RSP\_211}) = 0^{*}@-8)$$

(0 is a fresh region variable in this example). Since we do not know which operation the instruction represents at constraint generation time (i.e. which arm of the intersection type for the addition operator is actually in use), we take the disjunction of these choices to get

the total constraint. The next statement is a store; the choice of operations is unambiguous:

$$\begin{array}{rr} (\mathsf{mem} \sqsubseteq \tau(\text{mem64\_212}))\wedge & (\tau(\text{mem64\_327}) \sqsubseteq \mathsf{mem}) \\ \wedge(\tau(\text{R\_RSP\_211}) = 1^{*}@0)\wedge & (1 : (0 : 2)) \wedge (\tau(\text{R\_RBP\_0}) \sqsubseteq 2) \end{array}$$

The region variable 1, and type variable 2, are both fresh here. We derive the rest of the constraints in a similar fashion.

Now that we have this list of constraints, how are we going to solve them? The difficulty lies in these optional clauses, so we handle those last. We can start maintaining sets of type variables which form equivalence classes to make sure the information gained constrains to all of them. We can bound these equivalence classes above and below. We can do the same for region variables, but keeping track of offsets.

Actually trudging through this would take a while, but we can focus on seeing that `rdi` in this example must contain a recursive reference. `rdi` corresponds to type variable 16 in the constraint list provided. Selecting non-optional constraints first, and resolving the type variable unification and region variable unification constraints arrives at this quickly. The computer does not follow a pattern with a goal in mind like this. The inference engine just goes through the constraints which it can consistently absorb one by one, generating a *context* which contains a reduced form of the constraints.

Though much less practical for the manual reverse engineer, this method is vastly more mechanizable and less brittle. Effectively, to finish the example, the reverser would keep adding to their understanding, consuming constraints, until a disjunction makes a choice necessary. At that point, the reverser would bookmark their current understanding, and try one of the choices. If a given path did not work out, the reverser would backtrack to another choice. The key here is that incorrect choices will fail quickly; if the reverser tries to pick the integer math option on an addition, and the operation was really pointer arithmetic, by absorbing the non-choice constraints first, it will instantly see a contradiction in the accumulated knowledge of the situation.

Figure 3.7 shows the complete result of running this algorithm in full on the `loop_sum` example. That representation, while more complete, is complex and verbose, so here we instead display the recovered types of selected registers at function entry. The stack (`esp`) at function entry:

$$*(-32 : \mu\mathrm{A}. * (0 : \mathrm{A})(8 : \mathsf{int}_{32}))$$
$$(-12 : \mathsf{int}_{32})(-8 : \forall B \trianglelefteq \bot_{64} \trianglelefteq \top_{64}.B)$$

The -8 position on the stack is to contain the old value of `rbp`. Its value is never used or defined, so the algorithm does not discover anything beyond its size. At position -12, we see our first local variable. We know from manual inspection that this contained the running sum. The algorithm inferred it to be a 32-bit integer, as we would hope. Lastly, at -32 we see the local pointer the function used to iterate through the linked list - it contains a next pointer at offset 0, and a 32-bit integer payload at offset 8.

Examining `rdi` at function start gives us our input type, since this function uses the normal C 64-bit calling convention:

$$\mu\mathrm{A}. * (0 : \mathrm{A})(8 : \mathsf{int}_{32})$$

This gives us what we expected - the function is taking in a linked list of 32-bit integers. Finally, we examine `rax` at function exit to see what it returns, and see that it contains a padded int_{32}

3.3.2 Sufficiency of Register Types

Previous work [26, 49] has used some form of variable recovery before attempting to regenerate types in order to avoid dealing with storage locations whose type will change as the program executes. TIE used methods from DIVINE [12] to find its list of variable locations, while SecondWrite depended on LLVM's `mem2reg` pass. As SecondWrite mentions, DIVINE is slow, and thus poorly suited to large scale analysis. SecondWrite's choice of `mem2reg` is much faster, but any nontrivial use of a stack address will prevent that slot from promotion to a variable, and therefore prevent its analysis. Instead we opt to avoid the notion of variable recovery during our type recovery. Any access to a variable must either be through one of the available registers, or via a prearranged, well-known address (i.e. for a global). As a result, if we track the types of registers, including the fields of their structures, we recover the types of the variables without even considering which areas were originally variables and which are not until evaluation. This also removes dependence on assumptions of variable access patterns, and provides a more direct view of the types at the assembly level. With minor tweaks at function call boundaries, even the stack bears representation as another struct pointer. This major insight allows us to avoid dependence on potentially expensive or fragile analyses as preconditions for our inference.

3.3.3 Constraint Generation

Using a SSA-based representation means we can examine statements separately, as the transformation encodes the dataflow problem in the naming. The constraint generation does not need to ask whether this `eax` and that `eax` are the same, as the variable names identify a unique definition site. As a result, we do not need to consider the context in which a statement occurs in order to generate the constraints for that statement. The constraints will interact with other constraints, but this will dispatch on type variable matching rather than control flow. This greatly simplifies this step.

Constraint Forms

We can constrain our unifier based on the statements defining and using variables. First, we can apply upper and lower bounds to a type variable. This expresses that in whatever solution we come up with, the type variable must fall in a given range. For example, if the program assigns a pointer to a variable, its type must be above the pointer type. Similarly, if the program jumps to a variable, that variable must fall below `code`. We represent these restrictions as $A \sqsubseteq \tau$, $\tau \sqsubseteq A$, or $A \sqsubseteq B$. Notably, a type variable must be alone on at

$$\begin{aligned}
\text{constr} ::= {} & A \sqsubseteq \tau \,|\, \tau \sqsubseteq A \,|\, A \sqsubseteq B \\
& |\, A \cong \tau \,|\, A \cong B \\
& |\, rv : \rho \\
& |\, \text{constr} \wedge \text{constr} \\
\text{stmt constr} ::= {} & \text{constr} \\
& |\, \text{stmt constr} \vee \text{stmt constr} \\
\text{program constr} ::= {} & \text{stmt constr} \\
& |\, \text{program constr} \wedge \text{program constr}
\end{aligned}$$

Figure 3.5: Constraint Grammar

least one side of the constraint. We were able to express all the expression and statement constraints in this form. Disallowing statements of the form $\tau \sqsubseteq \tau'$ made implementing the solver easier due to the ability to index any constrained entity by a type variable rather than needing the ability to break down both sides simultaneously to make a the original constraint hold true.

Additionally, we have constraints for explicit unification, of the form $A \cong \tau$ or $A \cong B$. Again, we intentionally did not allow $\tau \cong \tau'$ for simplicity. This constraint indicates that we somehow either know the exact content of a type variable, or that two type variables really refer to the same thing. This is primarily useful for dealing with assignments to structures, where we want to merge the constraints accumulated so far on fields of two regions discovered to be the same. For most purposes, unifying a type variable with a type is the same as applying an upper and lower bound of that type to the type variable. The one exception to this is for pointers, which due to their subtyping structure, will not unify their regions unless an exact match on all defined offsets is present.

The last kind of constraint is the unification of region variables with regions, written $rv : \rho$. This kind of constraint requires that those fields defined in ρ are in the solution for rv, and unify with the type variables in those fields in ρ. This allows conveniently constraining portions of a structure type at a time.

By taking the conjunction of constraints formed like this, we can express any particular interpretation of a statement. However, one of the unique parts of this particular typing problem is that some of our functions (especially $+$) have multiple interpretations which cannot be conveniently formed into a single type. For example, it is unclear whether x is a number or pointer in the expression $x + 2$, and as a result, the type of $x + 2$ is unclear. This forces us to either consider intersection types [40, 65] or use a disjunction of constraints per statement. In the intersection typing approach, the system generates as complete a type as possible for each variable. For example, if we were trying to type a function $f(x) = x + 2$, the type would be similar to $f : (\text{int} \rightarrow \text{int}) \wedge (\text{ptr}(r@n) \rightarrow \text{ptr}(r@(n + 2)))$. This approach initially seems more elegant because it allows more complete descriptions of a piece of code. This approach fares poorly in our application because even inference of well-behaved code can quickly become exponential both in the time taken and in the size of the result type. Instead, we opt to generate a disjunction of constraints for statements which include expressions which would require intersection typing for a most general type. As long as we are satisfying one constraint, the expression will be legal, and in practice, operations like $+$ are not used for different purposes in the same generated code. This approach leaves out some possible typings (e.g. if whether a variable is a pointer or an integer is unclear, the system will end up needing to select one) and makes a small number of programs no longer legal (for example, a non-builtin plus function used both for pointer arithmetic and for integer math). Additionally, the choice to use constraints will make our search problem (in terms of finding which interpretations work) more tractable than inference would be in the intersection typing case. As a result, a statement's constraint (stmt constr) is a disjunction of conjunctions of the core constraint type (constr). In order to describe the entire program, we take the conjunction of all the statement constraints and solve the result.

As each of these constraints are separately generated, in a Datalog-based system we could express their generation as an external predicate on a single rule. This rule could generate a separate fact for each possible disjunction for an IL statement, or a single no-op constraint for instructions which would normally not have yielded one.

Inter-procedural Constraints

The basics of inter-procedural analysis are straightforwards in this system. We use unique variables when lifting each function and store them in a table. On a function call, we look up the target function's input registers, and say that their types must be a supertype of the type of those same registers at the event of function call. Next, we process the output registers similarly, applying subtyping constraints here instead. There are two issues here, both deriving from the stack: pointer super/subtyping and stack slot re-use. The first issue occurs when applying a supertyping to the stack register upon a call. The stack register's struct will then include temporaries from the callee. This will work fine, until two functions calls occur in sequence with incompatible local stacks. This will cause an issue because the stack is now constrained to have incompatible uses of stack slots beneath the caller's stack.

The second issue is that compilers will commonly re-use stack slots for calling functions, so if the program calls functions with incompatible inputs in sequence, the stack will be ill-typed. The easiest approach is to simply assume that the stack register has lost all meaning post function call, and re-infer the relevant portions based on its use after that. However, this will be dropping potentially useful information. A slightly more sophisticated approach would be to try to identify function call prologues and pull the stack type from before them. Unfortunately, that approach would be compiler specific.

Instead, we add the ability to erase type variables from regions after a function call occurs. For example, on an i386 system, calling a two argument function will cause the bottom three values on the stack (e.g. including the return pointer) to no longer be present in the type of the stack post call. This is one of the few convention-specific adaptations of the system; the function call ABI is agnostic, as the convention definition is just a description of what registers each call uses/defines, and what registers the convention uses for input and output. However, in order to deal with stack-based calling convention, we have adapted to the notion that the stack pointer has a special set of invariants at calls.

Examples

One example of a simple constraint generation would be for a multiplication. If we have the a fragment of code in the IL $X_1 : 64 = X_0 : 64 * 3$, we are dealing with the simpler case of a non-intersecting expression. We know the exact bit-width of X_0 and X_1 from the lifting process. Assuming that X_0 and X_1 correspond to the type variables τ_0 and τ_1 respectively, we would generate the constraint $(\tau_0 \sqsubseteq \mathsf{int}_{64}) \wedge (\mathsf{int}_{64} \sqsubseteq \tau_1)$. Note that these constraints are only one-sided subtypings. X_0 is only constrained to be usable as an integer. For all we know X_0 could have been a pointer, and this would still be legal. X_1 on the other hand must be definable by an integer. As a result, if X_1 is later dereferenced, the system will find this to be inconsistent. In a slightly more complex example, we examine $Y_1 : 64 = Y_0 : 64 + 8$ in a system with 64-bit pointers. In this case, this statement needs to generate two possibilities — one assuming that addition is an operation over integers, and one assuming the addition describes pointer arithmetic. Assuming type variables similar to previous example, we end up with a constraint $((\tau_0 \sqsubseteq \mathsf{int}_{64}) \wedge (\mathsf{int}_{64} \sqsubseteq \tau_1)) \vee ((\tau_0 \sqsubseteq rv^{*}@0) \wedge (rv^{*}@8 \sqsubseteq \tau_1))$. This describes both the pointer structure indexing behavior and the integer behavior simultaneously. Later, when trying to solve the constraints, we will have to select one of these behaviors. In the actual system, we also must cope with the possibility that 8 is an array index, so we add yet another constraint to the disjunction.

3.3.4 Unification

Unification is the process of coming up with a valid substitution for a set of type variables such that the substituted system will satisfy some set of constraints. Normally, these are only equality constraints. However, in our system we are simultaneously solving regular unification constraints and subtyping constraints to come up with a substitution for each type variable that will satisfy not only the equality constraints, but also the subtype ranges. In order to do this, we maintain a context that tracks which constraints we have absorbed, maintains a simplified form of constraints, and allows for efficient checking of whether the context is still consistent.

The simplest kind of constraint is a lower or upper bound. If the constraint is entirely abstract (e.g. both are type variables, and neither type variable has a known substitution) then we just record the relation into our context for consistency checking as we load other constraints. If one bound is concrete, we load that bound into the type variable's constraint in the context, taking a meet or join as necessary. If both bounds are concrete, we check that the two types are subtypes, possibly propagating requirements to the context if both are pointers. In the special case where the constrained types are pointers, we want to delay processing of this constraint until the context has processed the rest of the constraints. The reason for this is that we need all of the offsets defined on the pointers that ever will be in order to propagate them across during this. In practice, these constraints tend to match function calls, and so running them last is usually a good decision; each function is usually understandable on its own.

When two type variables must be equal by a constraint, if both are abstract, the context merges their bounds, and one of their equivalence classes chosen as the representative for both. If only one is concrete, the system checks that the determined type matches concrete bounds (e.g. bounds which are types) and then takes all the bounds which are on type variables, and sends them to the corresponding type variable. For example, if we are unifying τ_0 and τ_1, and we know τ_0 is a int_{64}, and $\tau_1 \sqsubseteq \tau_2$ in the context, then we might end up picking τ_0 as the representative for τ_1, deleting τ_1's constraint entry, and adding $\tau_1 \sqsubseteq \tau_2$ to τ_2's constraints. If both type variables are concrete, the system verifies equality, unifying argument regions in the case of pointers.

During pointer unification, region variables will need unification. First, we need to make sure that for each element in the region, we unify those type variables. Then, we need to select one region variable to be the representative for all the equivalent region variables. Notably, since all pointers are modulo offsets, each region variable also needs to know its offset from the representative. Finally, we update that region with all the type variables that had definitions in one but not the other. If we want to unify a region variable with a sample region, the core operation is to provide a set of type variable unifications for some subset of offsets.

The primary design issues here are to avoid cycles in updates when there are cycles in the types, and to track only the relevant parts of the constraints (effectively reducing them). This allows us to efficiently check whether or not we have violated constraints in order to ensure that our search through the possible disjoint constraints can take place efficiently.

This operation, when implemented in Datalog, would need some care to avoid excessive redundant computation. When adding two constraints to the solution context, the order should not matter if both of them would succeed. Specifically, they should commute. Additionally, if we have solved some set of constraints together, we do not wish to attempt to solve any subset. These two properties together suggest a lattice-like aggregation structure.

3.3.5 Search

As we alluded to before, instead of dealing with exponentially sized types, we choose to use constraints which were potentially disjunctive. Unfortunately, having disjunctions in our constraints means we have to make choices when attempting to unify them. This forms a sort of search problem where for each statement, we want to select the statement that will lead us to a valid unifier, if possible. Initially, this seems worrisome, as there are a potentially exponential number of choices. Luckily, as alluded to in our discussion of how to absorb a constraint into the context, we can cheaply check for correctness in partially inferred contexts. As a result, we can make a choice, and then if the choice is wrong, stop before we have spent time dealing with the whole path. This, combined with the desire to only require a single path, not all paths, reduces what would naively be an exponential process to a tractable one.

Unfortunately, not all programs are typeable. This can occur for a number of reasons, including the program doing something that is actively unsafe, unmodeled operations, and lifting or control flow analysis errors. A good type recovery algorithm needs to be robust in the face of this, so we need some goal for what to do if the constraints are not simultaneously satisfiable. We choose to select an answer which satisfies the maximum number of constraints. In the degenerate case where all constraints are simultaneously satisfiable, the satisfying solution is still the best one. In situations where the constraints are not simultaneously satisfiable, this goal corresponds to assuming we misunderstood the meaning of a minimum number of statements.

First, we sort the constraints by the number of disjunctive clauses. This means that in the case where the constraints are satisfiable, the solver processes the non-branching constraints first. This is both positive from a search perspective (early decisions are less likely to be wrong), and from a domain specific perspective, as pointer reads and writes are of this form. Processing pointer reads and writes early means that incorrect choices for pointer arithmetic are likely to fail immediately. Then, for every constraint, we process each disjunction into a separate possible context. We then score each context with a triple of the number of constraints possibly satisfied, the number of constraints already satisfied, and a tie-breaker value. In each of the constraints, the disjunction ordering matches how probable the interpretation is. For example, doing array indexing by a constant is less likely than doing struct indexing by a constant. We base the tie breaker on the combined likelihood of each disjunction choice in a vacuum. At each step, we take the current context under examination, grab its next constraint, then for each choice, or the choice of dropping the constraint, generate the possible next steps, and place them into a heap. The sorting order for the heap is first by constraints already processed (to avoid backtracking when we do not need to), then by possible constraints to solve (to ensure we will search for the best solution first), and finally by the tie breaker, to prefer constraint choices that are more likely a priori.

3.3.6 Non-Monotonicity

Searching for a minimum number of dropped constraints again informs Holmes design. Enabling the arbitrary dropping of constraints in a Datalog representation would result in an intermediate state too large to deal with. Even if dropping only one constraint is sufficient, a program written in this way would compute the potential dropping of every possible constraint, resulting in an enormous set of facts. In order to make this more practical, we would either need to introduce control flow primitives (such as Prolog's cut) or some form of non-monotonic reasoning. Assuming no explicit control flow primitives, we require non-monotonic reasoning because adding a new arm to a disjunctive constraint would add a fact, but possibly *reduce* the number of facts which a correctly designed system would derive due to decreasing the number of dropped constraints.

This form of non-monotonicity matches circumscription in combination with the call/cc feature. We can structure a maximum number of dropped constraints as a closed world hypothesis (circumscription), making an assumption that the number of constraints to drop will not increase. Then, if for a given number of maximum dropped constraints, it can determine that no complete solution can exist, the maximum number of dropped constraints can increase (call/cc), retracting the insolubility assertion in the process.

3.3.7 Limitations

BiTR does not implement every possible type, or understand every form of invariant. For example, BiTR does not know how to deal with union types, even if a tag indicates which type the variable is. We could extend to deal with such types, but doing so would make the system a good deal more complicated, and move from being only dataflow dependent to being control flow dependent as well. Additionally, BiTR does not analyze value bounds on types, as one might expect from an enumeration type or an array index. Adding this form of analysis would require more detailed understanding of the numeric operation, and would require analysis resembling VSA[11].

Another issue is in the use of functions with variable arity. In order to do inter-procedural analysis, BiTR matches the input and output registers together for unification. However, without separate instantiation of the function at each call site, the varargs portions of the function's input stack will not match across usage of the functions. If functions were separately instantiated however, information from each call site would not propagate to another. It would be possible to write code that special cased the varargs on x86 calling convention, but this is specialization and future work.

Finally, the more inconsistent the program, the longer the system will take to recover the types. Especially nonsensical programs can take a long time as the system attempts to optimize for the fewest number of broken constraints.

3.4 Evaluation

Finally, we must judge our system's performance directly, to demonstrate the effectiveness of our ideas in practice.

3.4.1 Metrics

In order to judge the usefulness of our system, we need some way to compute how well BiTR reconstructed types on a known compilation. Previous work usually split accuracy and conservatism into separate measures [26, 49]. Conservatism here is usually done as a measurement of how often inference is wrong, whether due to a bug or an inherent problem with the system. There have been multiple different approaches to accuracy measurement. TIE [49] used a distance metric based on lattice distance, with inner portions of the type counting less for accuracy. SecondWrite [26] reforms this somewhat, defining their pointer recovery accuracy based on detecting the number of indirections. However, their modification still leaves the question of how to evaluate structure correctness unaddressed. REWARDS [50], as a dynamic system, did take into account the layers of indirection, as REWARDS categorized memory locations in the image. However, REWARDS used a simple notion of correctness, checking whether the types presented for a given piece of memory were equal to the debug type in its entirety.

In addition to the differing notions of accuracy and what correct means, there is an issue with the decoupling of the conservatism, accuracy, and precision metrics. When decoupled, non-conservatism resulting in greater apparent accuracy and precision is not necessarily punished in accordance with which type variables non-conservatism occurs on.

Additionally, lattice distance is not a great measurement of accuracy, as simply guessing that all variables are integers of size equal to their storage size is effective on the distance scale. A majority variables are integers, and integers are only a short distance away from the pointer type if the type turns out to be pointer-sized, and this is directly at the middle of the top to bottom range. The lattice model further falls apart when we consider the notion of alternative lattices. When comparing two systems, which lattice should measure the distance between types on? Are two systems even comparable anymore? As more work occurs in type reconstruction, having a metric which will allow for the comparison of systems with radically different views of typing will be important.

We mix the notions of accuracy, conservatism, and precision together into a single metric. We define as our quality metric over a binary:

$$Q = E[B(A) \in C(A)] = \frac{1}{n} \sum_{A} \frac{|C(A) \cap B(A)|}{|B(A)|}$$

where $B(A)$ is the set of types BiTR recommends for type variable A, and C the set of types (usually a singleton) given by debugging information or ground truth.

This has a slight abuse of notation. Within the expected value, $B(A)$ is a random variable that generates a uniformly random member of the set $B(A)$. Additionally, in the expected value, A is the random variable picking a uniformly random value from the domain of C

This metric represents the rate at which a user of the system would be correct if it selected a type uniformly from our recommendations. This kind of evaluation represents the expected performance in an environment similar to what a user might experience if reverse engineering code — each time they ask the system for assistance, what is the probability that the system will recover the relevant portion of the debugging information.

Unfortunately, we can only compare ourselves to TIE with this metric, as we do not have

Implementation	Unconstrained	Constrained	Conservativity
BiTR	21.02% ± 2.64	38.00% ± 5.81	91.72% ± 3.35
TIE	12.64% ± 7.06	16.80% ± 1.63	90.31% ± 2.66
Improvement	8.38%	21.2%	1.41%

Table 3.2: Result Summary (Probability Metric)

access to implementations of the other systems. Note that while TIE describes a methodology for comparing its results against structure types, it does not in practice actually produce structure types. We report conservativeness and average lattice distance as well, but we do not consider these our primary objectives.

3.4.2 Can BiTR recover types in sample cases?

In small example programs which sum lists, walk binary trees, and perform simple look-ups in arrays of structs, we achieved 100% success. They provide a stark comparison to TIE, which produced 11% on summation of lists (corresponding to all things of a given size being valid), correctly identified the temporary variables in the tree walking for 15%, and could not determine that the global array containing the structs in existed, getting another 12% on temporary variables. We mention these programs to highlight that BiTR handles behavior that was previously not even measured in the literature. Hex-Rays' [35] performance on the loop summation example, as seen in the introduction, would have scored a 14% with that output. Note that the comparison here is somewhat unfair, because IDA must express exactly one type to the user. As a result, Hex-Rays' other numbers are not reported as the comparison is not meaningful or fair.

3.4.3 Does BiTR's recovery ability exceed previous work?

For a more real-world evaluation, we ran our system over `coreutils`, a collection of common programs installed on nearly every UNIX system. The programs in `coreutils` represent a variety of different kinds of workloads, and are open source, so we were readily able to build

Implementation	Unconstrained Dist.	Constrained Dist.
BiTR	1.32 ± 0.06	0.65 ± 0.075
TIE	2.10 ± 0.044	1.58 ± 0.25
Improvement	0.78	0.93

Table 3.3: Result Summary (TIE Distance Metric)

and examine debug symbols for them. We use this data set to answer this and our remaining questions.

In an effort to demonstrate the validity of our new metric and compare ourselves to systems other than TIE (on which we were able to generate our new metric scoring due to TIE access), we also report statistics of conservatism and distance in the same way as previous work [26, 49]. Overall, on `coreutils` we achieve conservatism of 95.95% and an average distance of only 1.33. If we only examine type variables which had at least one constraint reference them, we see instead an average distance of 0.68, an improvement on previous work.

3.4.4 Is BiTR's recovery good on an absolute scale?

Unfortunately, we cannot run most of the other systems under this metric, as implementations are not readily available. However, we did get access to an implementation of TIE [49], so we ran TIE under this new metric in addition to our work. Over all the type information contained in the DWARF information for `coreutils`, BiTR achieved a 21.01% recovery rate. TIE, when scored on this metric, achieved a 12.65% rate.

As our new metric has a direct interpretation in terms of its usefulness to the reverse engineer, these values show that BiTR is useful for recovering types. Additionally, it highlights the progress made on an absolute scale. However, it also shows how much further the field has to go — a one in five chance of having the correct type is unlikely to be the best possible solution.

Under-constrained Types. One of the biggest sources of error in our analysis is under-constrained types. The code references and operates on these variables, but due to the low number of operations performed on them, there is no way to know definitively what the type of these variables are. For example, in the C snippet

```
int f(void* x) {
  ? v = x;
  return (v == 0);
}
```

we would get code which would be legal if our mystery type (represented by a ?) were either an `int` or a pointer of some variety. We can use heuristics when outputting a type for decompilation (for example, preferring the lower bound of a type in order to avoid casting), but since both are legal, and all constraints on `v` are visible in this snippet, we would never be able to get an exact type.

Dealing with these would be a matter of better heuristics, and would have left the realm of program analysis and gone into the realm of programmer analysis. The heuristic of selecting the lower bound is not a good one, as this will reduce conservativity, but sometimes this choice may make sense. Developing heuristics to deal with this issue is outside the scope of this work, but presents an interesting challenge discovered in our experiments.

3.4.5 How does BiTR fare at recovering constrained types?

In order to more closely examine the effect that unconstrained type variables have on our metrics, we generated a probability mass plot of response quality in both situations, as shown in Figure 3.6. This shows the difference between responses that BiTR has constraints on, and those it does not. When we restrict ourselves to type variables which the code accessed, our recovery rate spikes to 36.78%, showing great overall improvement.

This makes sense as a measurement because it demonstrates the ability of the tool to police the boundaries of its own knowledge. Specifically, it measures the performance of the tool when used as an advisor rather than as the sole source of types. When BiTR has seen

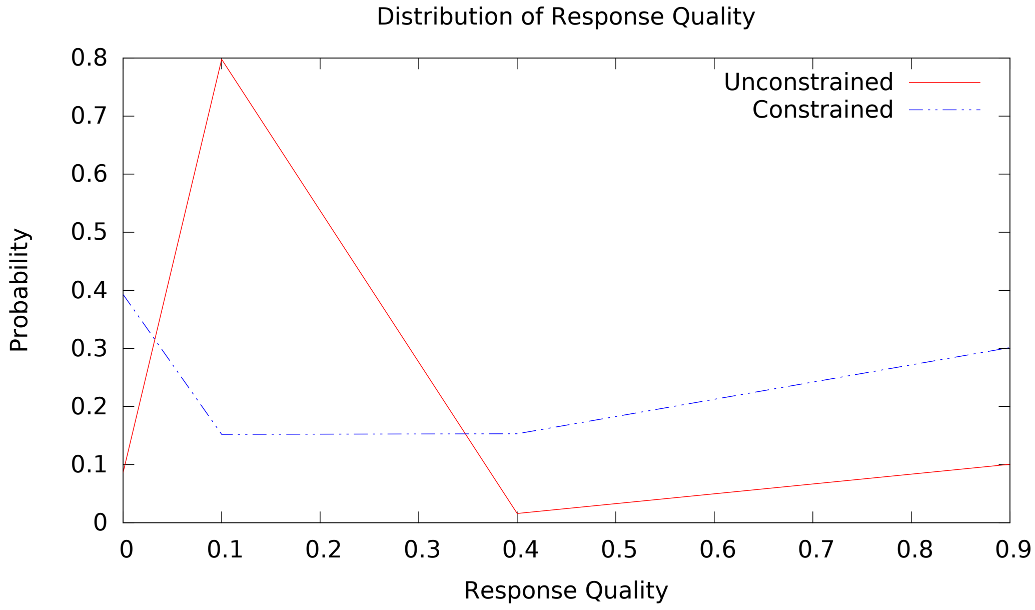


Figure 3.6: Effects of Unconstrained Type Variables

the type variable in question constrained, it has a much greater level of information available on what the type could be.

Unconstrained Type Variables. The primary source of error reflected in this difference is data that is never written to or read from. For example, while most `coreutils` binaries contain a `close_stream` function which operates on a stream structure, there is no indication of the inner properties of this structure in the code itself. However, the compiler constructs debugging symbols in the presence of header files that describe the fields of the stream structure. As a result, the score is strongly deflated. We can partially mitigate this by using type signatures for library functions, but we did not use signatures for all functions in our experiments, nor would this information always be available. BITR can recover some unconstrained variables through appropriate type signing (or analysis of the target shared library). Unfortunately, there are also cases where some fields of a structure are never referenced over its entire life cycle.

3.4.6 How does BiTR scale?

It is important to note its scaling characteristics relative to the size of the input problem. Two good proxies for problem size are number of type variables and number of constraints. In practice, in our conjunction of disjunctions, there are about approximately the same number of disjunctive clauses as there are type variables. BiTR experimentally uses an amount of memory linear in both of number of constraints and type variable count, and time proportional to the square. This happens because when checking consistency, the system checks each live type variable rather than only those whose constraints may have changed. Adding an optimization which only checked those type variables affected directly or indirectly by incorporating a new constraint would replace a factor of N with $K \log N$ by removing the requirement to check every live type variable.

3.5 Related Work

As this work stands at the confluence of compilation, instruction set architectures, static analysis, and type theory, there is a great deal of prior work that provided the foundation to create BiTR. There have been other attempts to perform binary type recovery. Type theorists have explored the relevant formal systems that enable us to appropriately describe the constraints imposed by the wide variety of instructions. Others have tried to build decompilers, each of which contains at least an attempt at type reconstruction.

3.5.1 Types in Compiled Code

In large part, previous work has considered dynamic approaches, which use execution traces to get information about concrete values. Another school of thought takes a more forensics-oriented approach, attacking the problem by looking for known data structures within a dump or trace. Finally, there is the school to which this work belongs, static type recovery, where the approach regenerates type information from a representation of the code, rather than from sample runs or matching known data structures.

Dynamic Type Description. Rewards [50] takes a dynamic, trace-oriented approach to the problem, taking execution traces and known system calls, and propagating types from system calls through the trace's reads and writes. The dynamic approach has the advantage that the analysis can know what values a memory location or register actually held at a given program point. Additionally, the dynamic approach does not have to solve the problem of indirect jumps, as when working with traces the next instruction is precisely computed. Finally, since Rewards had exact aliasing information via pointer values on each trace, flowing information from the system call barrier (their major outside source of types) is easier. The primary limitation of this approach is that it cannot assign appropriate types for structures which do not cross the system call barrier, such as container data structures. This approach of providing dynamic information from the crossing of the system call barrier could be supplied as additional constraints to a system like BiTR to further improve accuracy.

Howard [68] extends the work of Rewards by focusing on access patterns instead of simple propagation, and annotating variables from the original code, rather than locations on a dump.

Type Forensics. Another approach known as shape analysis [21, 37, 41, 61, 79] uses dynamic traces to generate shape graphs, which they then analyze to make guesses at the types of memory locations. The systems generate the shape graph by first generating a trace, then matching the access pattern to the simplest possible graph of type structures. Some generate this trace from the compiled program, and some must annotate the program prior to compilation to achieve this trace. Once the system generates the shape graph, it compares the shape graph to multiple possibilities of what the data structure might be in attempt to classify it e.g. as a binary tree or linked list. One benefit of this technique is that when the system finds a match, more information on a name of the data structure may be available. However, if the program uses a data structure not expected by the system, some of these methods will fall short. For example, MemPick will report it to be a generic graph. It also suffers from the standard dynamic analysis issue of being unable to generate types for paths the test cases did not drive it down.

Static Type Recovery. Like TIE [49], we built BITR on BAP [16], and also took the approach of trying to generate ranges of constraints. However, TIE performs much worse under our metric, which we feel more fully represents accuracy of more complicated types. TIE's metric is problematic for the reasons described in [26], but the proposed replacement metric is still dependent on a notion of distance. TIE is also slow, which hobbles its use as a large scale analysis tool. The use of DIVINE's methods was one of the bottlenecks, which we avoided in BITR by recovering the type of everything that is addressable through the registers or a constant integer used along a dataflow that ended in a read or write. TIE rarely inferred structure types, though its type system contained them. This was in part encouraged by a metric which put less weight on the types of struct members. Finally, if run on a static binary (e.g. without dynamic library hints), the amount TIE could infer itself was minimal.

SecondWrite [26] instead takes the approach of lifting to a LLVM-based IR [47], then using `mem2reg` to detect variables and LLVM pointer analysis to compute the types. Their reconstruction is simpler and faster than TIE's, but the approach has issues: `mem2reg` is a nice shortcut, but has the problem that `mem2reg` will not promote anything which has a use other than a load or store [47]. As a result, if on-stack references are in use, those stack slots will not be properly promoted to variables. Additionally, dependence on pointer analysis leaves them without a way to detect recursive types within their framework, and makes nested structures unlikely to work.

Another work focusing primarily on structure recovery [73] approached the problem from the angle of figuring out what idioms compilers used to address arrays and structs, and then tried to reconstruct structs and arrays. However, by the authors' own admission, this approach cannot handle nested structs. Additionally, their dependence on assumptions about how the compiler will act and how the source language must work cause the output to be of limited use for understanding properties of code which was not necessarily built by the compilers or language expected.

3.5.2 Type Theory

Some of the inspiration for this form of type characterization §3.2 came from intersection typing [40, 65]. Though we did not end up using intersection types for inferrability purposes (even the decidability of the inference turns out to be difficult and limited [43]), this work informed our choice of a constraint-intersection §3.3 approach instead of type-intersection approach.

Earlier efforts to generate typed assembly [22, 55] also bear similarity to our work. Typed assembly language methodologies are attempting to assign types to the registers in compiled code during compilation. Some of the TAL ideas are applicable, and still others could potentially help in future reconstruction work as safer types. However, the majority are inappropriate for the work because the compiler or author must make the code conform to the system, rather than the system describing the code.

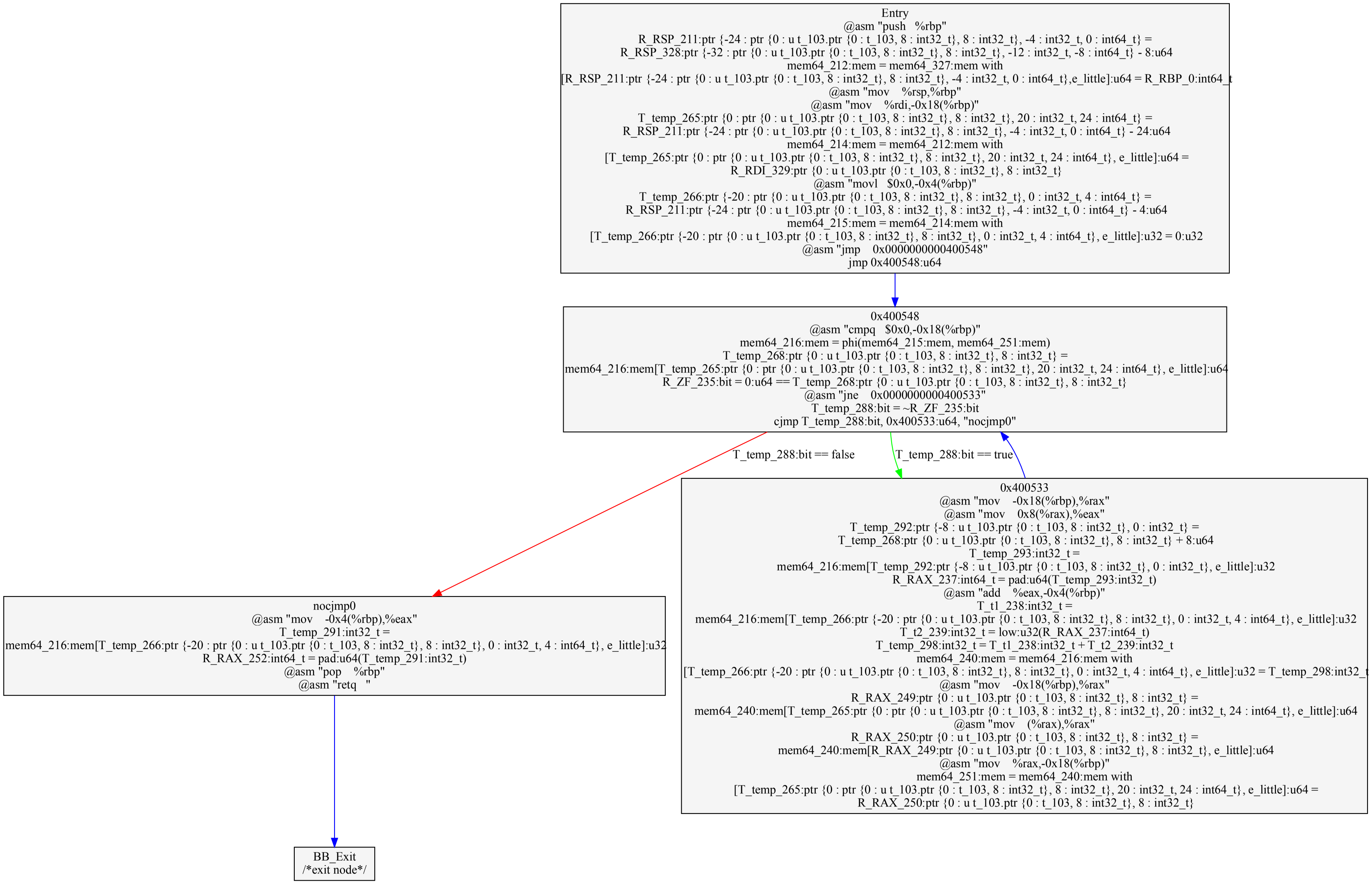
3.5.3 Decompilation

One of the main applications for type reconstruction is decompilation. Some approaches [56] even suggest that the type reconstruction can help guide the decompilation itself rather than simply being a set of annotations applied at the end. This idea has existed [24] in decompilation for a while, but progress has been slow. More recent decompilers [64] have used some of the other research [49] in the area to improve their results as well. Given the poor state of affairs in Hex-Rays [35], more work in this field could improving the usability of much of the decompiler work would not be surprising.

3.6 Motivating Holmes

BiTRwas designed before the inception of Holmes. Despite this, its operation matches the computational patterns we will introduce when we describe the language (§ 4). The constraint generation step matches simple Datalog rules, with external predicates or an equivalent feature (§ 4.2.2). We process SSA instructions one at a time and transform them into constraints to try to simultaneously satisfy. The solving phase matches up with the aggregation feature (§ 4.2.3). We can promote a single constraint into a partial solution context, and the operation to add a new constraint to the context transformed into a join operation between two partial contexts. Finally, attempting to leave out a minimum of constraints corresponds to circumscription(§ 4.2.4) and call/cc (§ 4.2.5). Specifically, we circumscribe over the quantity of predicates to ignore in a final answer. If no final answer can is possible, we increase the number by using a `max` join operation on the maximum number of dropped clauses. If we circumscribe over the list of available merged contexts at well, we can detect the case of an inability to produce an answer, and increase the number of potential dropped clauses. This step requires call/cc, as we must circumscribe over the maximum number of predicates to drop to know there is no solution, but our response to knowing there is no solution is to extend that predicate.

A BiTR system re-implemented in this way would still calculate more than the system described here. This is because it would examine *all* minimal constraint dropping solutions rather than stopping after finding one. It would also gain in incrementality, as adding or removing a constraint (such as due to user annotation, or a small patch) would be able to operate on the existing state as much as possible. The non-Datalog implementation would need additional work to become incremental in this way.



```
Entry
@asm "push   %rbp"
R_RSP_211:ptr {-24 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, -4 : int32_t, 0 : int64_t} =
R_RSP_328:ptr {-32 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, -12 : int32_t, -8 : int64_t} - 8:u64
mem64_212:mem = mem64_327:mem with
[R_RSP_211:ptr {-24 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, -4 : int32_t, 0 : int64_t},e_little]:u64 = R_RBP_0:int64_t
@asm "mov    %rsp,%rbp"
@asm "mov    %rdi,-0x18(%rbp)"
T_temp_265:ptr {0 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, 20 : int32_t, 24 : int64_t} =
R_RSP_211:ptr {-24 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, -4 : int32_t, 0 : int64_t} - 24:u64
mem64_214:mem = mem64_212:mem with
[T_temp_265:ptr {0 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, 20 : int32_t, 24 : int64_t}, e_little]:u64 =
R_RDI_329:ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}
@asm "movl   $0x0,-0x4(%rbp)"
T_temp_266:ptr {-20 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, 0 : int32_t, 4 : int64_t} =
R_RSP_211:ptr {-24 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, -4 : int32_t, 0 : int64_t} - 4:u64
mem64_215:mem = mem64_214:mem with
[T_temp_266:ptr {-20 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, 0 : int32_t, 4 : int64_t}, e_little]:u32 = 0:u32
@asm "jmp    0x0000000000400548"
jmp 0x400548:u64
```

```
0x400548
@asm "cmpq   $0x0,-0x18(%rbp)"
mem64_216:mem = phi(mem64_215:mem, mem64_251:mem)
T_temp_268:ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t} =
mem64_216:mem[T_temp_265:ptr {0 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, 20 : int32_t, 24 : int64_t}, e_little]:u64
R_ZF_235:bit = 0:u64 == T_temp_268:ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}
@asm "jne    0x0000000000400533"
T_temp_288:bit = ~R_ZF_235:bit
cjmp T_temp_288:bit, 0x400533:u64, "nocjmp0"
```

```
0x400533
@asm "mov    -0x18(%rbp),%rax"
@asm "mov    0x8(%rax),%eax"
T_temp_292:ptr {-8 : u t_103.ptr {0 : t_103, 8 : int32_t}, 0 : int32_t} =
T_temp_268:ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t} + 8:u64
T_temp_293:int32_t =
mem64_216:mem[T_temp_292:ptr {-8 : u t_103.ptr {0 : t_103, 8 : int32_t}, 0 : int32_t}, e_little]:u32
R_RAX_237:int64_t = pad:u64(T_temp_293:int32_t)
@asm "add    %eax,-0x4(%rbp)"
T_t1_238:int32_t =
mem64_216:mem[T_temp_266:ptr {-20 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, 0 : int32_t, 4 : int64_t}, e_little]:u32
T_t2_239:int32_t = low:u32(R_RAX_237:int64_t)
T_temp_298:int32_t = T_t1_238:int32_t + T_t2_239:int32_t
mem64_240:mem = mem64_216:mem with
[T_temp_266:ptr {-20 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, 0 : int32_t, 4 : int64_t}, e_little]:u32 = T_temp_298:int32_t
@asm "mov    -0x18(%rbp),%rax"
R_RAX_249:ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t} =
mem64_240:mem[T_temp_265:ptr {0 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, 20 : int32_t, 24 : int64_t}, e_little]:u64
@asm "mov    (%rax),%rax"
R_RAX_250:ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t} =
mem64_240:mem[R_RAX_249:ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, e_little]:u64
@asm "mov    %rax,-0x18(%rbp)"
mem64_251:mem = mem64_240:mem with
[T_temp_265:ptr {0 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, 20 : int32_t, 24 : int64_t}, e_little]:u64 =
R_RAX_250:ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}
```

```
nocjmp0
@asm "mov    -0x4(%rbp),%eax"
T_temp_291:int32_t =
mem64_216:mem[T_temp_266:ptr {-20 : ptr {0 : u t_103.ptr {0 : t_103, 8 : int32_t}, 8 : int32_t}, 0 : int32_t, 4 : int64_t}, e_little]:u32
R_RAX_252:int64_t = pad:u64(T_temp_291:int32_t)
@asm "pop    %rbp"
@asm "retq   "
```

```
BB_Exit
/*exit node*/
```

Figure 3.7: Solved List Summation

Chapter 4

Holmes

Holmes is a dialect of Datalog, tailored with extensions for the specific use case of analyzing compiled code. Specifically, a normal dialect of Datalog will fall short on several tools desired by the analysis author:

- Data structures
- Aggregation
- Negation

4.1 Feature Selection

To address these lacks, we add a few features to base Datalog.

4.1.1 Callbacks

Tasks which do not involve a fixpoint, but do involve computation, can frequently be both more difficult and more expensive to write in pure Datalog. For example, parsing an ELF and splitting it up into segments, sections, generating records for its symbols, etc. could in principle be written in Datalog. However, this would be difficult to write (operating on a string as a linked list, or similar structure), slow to compute due to many-way joins, and would require that the input first be transformed before even entering the program. Other similar examples include lifting (translating a sequence of bytes into a semantics language), concrete execution, and arithmetic.

Previous approaches have noted that many of these steps come towards the beginning of analysis, and perform these tasks as a precompute phase before handing the results to Datalog to process. In our case, we are trying specifically to avoid such phasing. The lifter might be needed again for previously undiscovered code. The loader might be needed again if we discover a call to `execve` and wish to follow it. Doing a phased Datalog prevents the easy interleaving of these functionalities into the global fixpoint.

Datalog predicates are not always the best data structure for all tasks. Datalog predicates can effectively be viewed as an append-only, deduplicated, index-by-anything table for each predicate with rows corresponding to true instances of the predicate. This structure is very versatile, and can represent a wide variety of concepts. However, some concepts are better represented in other ways. One example is ILs and ASTs. As frequently nested, branched structures, they *can* be represented in Datalog, but walking one would take a very large number of lookups compared to using a traditional algebraic data type approach, not to mention the clumsiness. Other similar concepts include formulae (as in SMT) and any kind of variable-length buffer representation. All of these can be done in pure Datalog, assuming appropriate preprocessing has been done. However, the resulting time and space costs make this something to be avoided.

To address the above two, we add the ability to register a callback to a Datalog rule. If specified, whenever a that rule fires, the corresponding callback will be used supplementarily to determine the values to substitute into the head. This allows use of traditional functional or imperative style code to implement data structure transformations or perform operations which would be slow to do in the base Datalog. Additionally, it allows us to more readily incorporate existing code (such as the BAP [16] lifter) rather than rewriting it from scratch.

This is equivalently powerful to external predicates in other languages in terms of expressivity. Any callback specified could instead be turned into an external predicate and simply appended to the query. A query involving external predicates might need to be split up into phases to be expressed in callbacks. If an output variable of an external predicate is present in another term in the query, one would need to do a secondary join after evaluating the external predicate. As the callback only occurs at the *end* of a query, there will only be one join per query. The callback restriction simplifies the design of the Datalog engine (the join engine is entirely separate from the callbacks), at the cost of the ability for a sufficiently advanced engine to better optimize such queries.

4.1.2 Monotonic Aggregation

Traditionally in Datalog, the body of a rule may only access a fixed (though arbitrary) number of facts at the same time. Even counting can be difficult. To verify that there are at least three instances of some predicate p, one would normally write:

```
p(x) & p(y) & p(z) & neq(x, y) & neq(y, z) & neq(x, y)
```

The size of this query grows as n^2 in the number of elements to be counted.

This same difficulty occurs when encoding a dataflow or abstract interpretation algorithm into Datalog. When two branches come together, a new fact representing the state with the meet lattice operation for the chosen domain applied needs to be generated. If we do this naively, simply matching on the existence of two states at that program point and generating a new one by merging, the resulting runtime will be super-linear in the number of performed meets.

In existing systems [14] this is dealt with by ensuring the state in question can be extended simply by adding more facts. This solution works in some cases, but it prevents the use of data structures like strided intervals [11] or widening operators in dataflow algorithms which lack finite descending chains.[1] This is because all of these situations require reasoning about

---

[1] A lattice is said to have finite descending chains if at any point at the lattice, there are a finite number of values less than it on the lattice. This property is desirable for a dataflow analysis because it guarantees termination. When finite descending chains are not present, a dataflow analysis can provide a "widening" operator, which can force the termination of the system by moving farther in the lattice under heuristic conditions.

a variable sized subset of the data to make their conclusion, not just a fixed window.

Finally, external solvers often need to receive all the inputs up front, rather than incrementally. Calling out to an SMT solver will not work if the formula from symbolic execution is stored as facts in a Datalog representation; the program would first have to walk them with a rule and a callback (or a rule and an external predicate in another system) to build up a viable representation and hand it off. The same is true even of simpler concepts, like applying Steensgaard's algorithm [71] to a set of constraints - the algorithm will either need to process all constraints at once, or it will end up store incremental program states in the database as well, ending up back at the n^2 problem.

Traditionally, this is dealt with by applying a post-processing step to the Datalog computation. After rules have been executed to saturation, a query is run, and the aggregation is performed by an outside-of-Datalog program. As stated earlier though, we want all portions of the analysis to be able to trigger all others to avoid explicit phasing.

Some of these specific scenarios can be worked around with via clever rules, but they do not apply universally. For example, the counting check might instead use a greater than operator instead of not-equals, assuming that field is ordered. The resulting query would then only have linear size in the count to check against. However, this construction still only handles counting a fixed number of unique results.

To address the issue of combining information from multiple facts efficiently, we allow for predicates with aggregation. If a predicate is declared with aggregation, a provided meet operator will be used to aggregate submissions to each aggregate field for which all non-aggregate fields match. In the case of counting, we simply use set-union as our meet operator. For dataflow or abstract interpretations, we can have parameters like program location be non-aggregate fields, while the state is an aggregated one. Programs using this feature need to be aware that they may receive the aggregation of any subset of derived facts, and are only guaranteed to ever receive the aggregation at the fixpoint.

4.1.3 Hypothetical Circumscription

Some questions revolve more around what isn't there than what is. For instance, if `ud2`[2] is found in the binary, we might wish to determine if it is in fact statically unreachable. This requires us to be able to state that we know *all* of the edges entering that basic block[3], not some subset.

As a more concrete application, if we have an algorithm which works on the SSA[4] representation of a function, creating an SSA representation of that function requires the entire control flow graph. If we add edges later, conclusions derived from the incomplete SSA form might become incorrect.

Traditional Datalog either disallows negation, or allows it through explicit stratification. In the context of doing program analysis on binaries, we might wish to avoid this even when reasoning purely monotonically. Consider an analysis which determines whether a function will never return. This information is important in analysis of a calling function because it should not expect control to proceed past the called function. To declare that a function will never return when called, we must know *all* the paths within it, not just some of them. As a result, we are implicitly talking about knowing the negation of additional edges in the control flow graph.

If we employed stratified negation, the system needs to declare an entire predicate saturated, not part of one. As a result, to reason based on the absence of any control flow edge, the system would need to assume saturation of the entire control flow graph. This leaves us unable to employ information that a called function will never return in the CFG definition for a calling function.

---

[2] LLVM inserts this instruction to denote unreachable code, and is intended to cause a trap if hit

[3] A basic block is a sequence of instructions with exactly one entry point and one exit point. Other code in the program does not jump to the middle of the block, and execution does not leave the block until the end.

[4] SSA, or single static assignment, is an intermediate representation frequently used by compilers to make explicit the dependency of variables on previously computed values. Each variable in SSA form is defined precisely once in the program, and additional variables are created for cases involving mutation. When multiple control flow paths to define a variable exist, a ϕ expression is used, selecting a previous definition indexed by the path taken to arrive at the current block.

To address this need, we add hypothetical circumscription. The core concept is that we can at need assume that a particular chunk of information is expanded, and reason forwards. In the event that this turns out to be false, we can retract that assumption, and reason forwards again. This allows us to deal with cases of negation which are not trivially stratified. In the language, this feature is implemented as the an additional kind of body atom. If a rule circumscriptively matches on an aggregated predicate, the resulting computation will be as if it matched only on the aggregation which is present in the final fixpoint database. Circumscription is in contrast with monotonic aggregation, where rules must have correct operation for any subset of the possible aggregations along the way. This can be used to implement the stratified case in a straightforward manner, and also to support dynamic negation as describe in the never-returning example above.

4.1.4 Call/CC

The astute reader may have become worried at the end of that last subsection, as it is not without reason that the standard approach to negation involves stratification. In the case that we have a negated inference cycle not just on predicates, as stratification prohibits, but on the actual facts, the approach as described so far would lead to alternation and indeterminacy.

Interestingly, there is actually a use case for allowing negated cycles in program analysis. In the case of the outputs of an dataflow analysis or a control flow recovery, we will need to circumscribe over their results in order to know we have received the actual output. Using an incomplete run of an alias analysis, for example, would result in too-small upper bounds being used. However, due to the interconnectedness of these examples, the complete alias analysis information might alter the control flow graph by refining the information used to determine the target of indirect jumps. Changing the control flow graph would in turn invalidate the circumscribed alias analysis.

Strictly looking at the system thus far, this would loop. Altering the control flow graph would retract the alias analysis, which would retract the alteration to the control flow graph. Thinking about what a human would do if they had gone down the same reasoning path points to a potential solution. If the analyst had assumed they had been shown the totality of control flow, and from that, came to find a new control flow edge, the analyst would decide that their initial assumption must have been wrong, and that edge really is there. Essentially, this step is $(\neg P \to P) \to P$.

This matches the type signature of `call/cc`[5], and not without reason. In this case, the continuation is the reasoning strategy forwards, assuming P can be determined to be true. If P is not determinable to be true, this continuation cannot actually be invoked, and we never go down that path. If P can be determined to be true, in a traditional programming language we might go down that path. In our case, we are constantly watching for P to be determined to be true, and if it is, we immediately take the continuation.

This feature is invisible to the user other than for performance characteristics. The user need only specify their rules as usual, circumscribing over things which need to be complete, and if this `call/cc` condition arises, it will be automatically dealt with by reasoning forwards from the new (expanded) circumscription, after retracting relevant other derivations.

4.2 Informal Semantics

Before giving a more formal treatment of what outputs are correct for a given Holmes program, we describe how each feature functions informally, as in a programmer's guide. Afterwards, there is a simple traced execution of a toy program which employs all of these features.

---

[5] `call/cc` stands for "call with current continuation" and is a programming technique originating in Lisp. It allows a function passed to it to receive as an argument a function corresponding to the rest of the program, known as the continuation. It is commonly used to represent failure and backtracking, but is not limited to those uses.

4.2.1 Initial Syntax

In Holmes, there are three kinds of possible statements, predicate declarations, rule declarations, and queries. In a predicate declaration, a predicate is given a name, and a series of typed fields. When done in a tuple-predicate style, this looks like:

```
derefs_var (Var, Location)
```

where `Var` and `Location` are types in the embedding language. This style can be used in cases where the meaning of each field is obvious. In larger projects or more complicated predicates, it is preferable to name fields. A predicate with named fields is written as

```
derefs_var {var: Var, loc: Location}
```

To avoid confusion, a predicate may only be defined in one of these two styles.

Rule declarations contain a rule name, a head, and a body. The rule name is used purely for error reporting, debugging, and provenance reporting. It does not have any effect on the operation of the program. A body consists of a series of atoms, joined by `&` symbols. The order of atoms in the body has no effect on the program. Matches are written in one of two styles depending on the style of the predicate.

If the predicate is a tuple style predicate, we can write

```
tuple_predicate (w, ~1, _, q)
```

This unifies the first field with the variable w, the second field with the constant 1, leaves the third field unconstrained, and binds the last field to the variable q.

In the field style, we can instead write

```
field_predicate {w, x: ~1, z: q}
```

assuming the predicate has fields named w, x, y, and z. This implements the same match as above. Matching on field predicates introduces two shorthands. First, unused fields need not be named (y in this case). We could write `y: _`, but it is unnecessary, making it easier to work with higher arity predicates. Secondly, referencing a field with no binding implicitly binds it to a variable of the same name (`w` expands to `w:  w` in the example).

Heads are written exactly as a body atom, except that unbound fields and fields bound to variables not defined by the right side of the rule are disallowed. Putting it all together, a rule looks like

```
reaching_propagate: reaching {loc, var} <-
    reaching {loc: prev, var}
  & succ (prev, loc)
  & unchanged {loc: prev, var}
```

Rules written like this act as normal Datalog rules, interpreted with a fixpoint semantics. If facts are present in the database so that an assignment to variables exists for which the right side is all present, then the left hand side will be added to the database.

Queries are essentially named body, which may be evaluated as a way of querying the program state by the embedding language. The name of a query determines the name of the function which will appear in the interface of the resulting database object. Queries are written

```
?all_reaching: reaching {loc, var}
```

When the database's `query_all_reaching` function is called, it will respond with a set of possible assignments to `loc` and `var`, essentially returning the current state of that predicate to the user.

Facts are inserted at the level of the embedding language, the database can be run for an arbitrary number of steps (or to fixpoint), and queries can be performed whenever the database is not stepping.

4.2.2 Callbacks

Callbacks are used in Holmes to allow the use of more traditional procedural or functional style code for some rules. After writing a rule, add `+ f`, where `f` is the name of the callback to be invoked. When the rule is considered, it will first try to match its body. If this succeeds, the variable assignments which match will be passed, one at a time, to `f`. `f` will return for each input assignments a list of assignments to those variables present in the head, but unbound by the body. If there are no variables left undefined in the head, whether `f` returns a least an empty assignment structure (e.g. a list of one element, with that element being the null assignment) determines whether the rule will actually produce its head. For example, in the rule

```
simple_func: p(y) <- q(x) & r(x) + f
```

`f` would be expected to take in the value of `x`, and return a list of values for `y`. If we wrote

```
check_even: special_even(x) <- special(x) + is_even
```

A `is_even` function which checked the last bit of x, then either returned `[{}]` (a list containing the empty binding) for true or `[]` (an empty list) for false could implement this rule.

4.2.3 Monotonic Aggregation

Monotonic aggregation defines a new way to declare predicate fields. When declaring a field, a caret (`^`) followed by a function name in the embedding language with signature $(T, T) \to T$ where T is the type of the field may be provided. Such a function should be a lattice meet operator - it should be associative, commutative, and idempotent. This function acts as an “aggregator” for that field. Whenever a new fact is added which matches the non-aggregated fields of a another fact, they will be combined according to this aggregator. Because the order of rule execution is up to the engine, this means the rule may receive multiple, incomplete aggregations along the way, or it may only receive the final aggregation

- the only guarantee is that if run to a fixpoint, the rule will have received the final (largest) aggregation.

For example, say that we wrote the program:

```
p(i32, IntSet^union)

q(i32, IntSet)

promote: q(x, ints) <- p(x, ints)

?result: q(x, ints)
```

and then inserted facts

```
p(0, {1, 2})

p(1, {2, 5})

p(0, {1, 3})

p(0, {2, 4})
```

When calling `query_result` after the program completed, we would be guaranteed to always see the assignments `(0, {1, 2, 3, 4})`, `(1, {2, 5})`. However, we might also see present `(0, {1, 2})`, `(0, {1, 3})`, `(0, {2, 4})`, `(0, {1, 2, 3})`, `(0, {1, 2, 4})`. Their presence in the output is up to the discretion of the evaluation engine.

Generally, this kind of aggregation is useful in cases where a rule wants to operate on all the information that is available, but future information will not make any of its actions incorrect. Imagine `p` in our example as having a first field representing a variable, and the second representing values it held in a specific evaluation of a program. In this case, the aggregation allows us to query for the set of values we know are possible to find in that variable. Our inferences will not become wrong in the future, because the predicate describes a lower bound, which is allowed to move up (via union).

4.2.4 Hypothetical Circumscription

Hypothetical circumscription extends monotonic aggregation by allowing us to only examine the largest aggregation we find. Circumscription is written on an atom by prepending a tilde to the predicate name. If we extend the example from monotonic aggregation with a circumscriptive match on p, e.g.

```
promote_complete: r(x, ints) <- ~p(x, ints)
```

`r` will contain only the pairs $(0, \{1, 2, 3, 4\})$ and $(1, \{2, 5\})$ once the program is done executing. Examining the database in the middle of execution may yield different results, but the engine will correct these before a fixpoint is reached.

This tool is intended to be used to query an aggregation which aggregates in a different lattice direction than the one it is queried. As a concrete example, consider abstract interpretation. Abstract interpretation assigns bounds to a variable which get larger and larger as the computation proceeds. This can be monotonically queried to see if something is going to be in bounds - this will only ever go from false to true as the bounds expand, never the other direction. However, the most useful part of an abstract interpretation bound are the values it rules out - those which are outside the bounds. In order to know that a value is outside the bounds, we need to know that we are looking at the final aggregate for that particular domain, and there circumscription becomes important.

Well Behaved Circumscription

In the vast majority of programs, circumscription will not bring any surprises to the table. However, it is possible to encode a notion of choice using circumscription, causing the fixpoint evaluator to choose between one of two possible worlds. For example, consider the program

```
p(IntSet^union)

q(IntSet^union)

big_p_world: p(~{2}) <- ~q(~{1})
```

```
big_q_world: q(~{2}) <- ~p(~{1})
```

and insert both `p({1})` and `q({1})`. The engine will now either output `p({1, 2}), q({1})` or `p({1}), q({1, 2})`, and what it does is implementation defined, and may even differ from run to run.

This nondeterminism is usually not desired, but there are currently no checks to detect it, either at compilation time or at run time. Stratifying [72] circumscription will avoid this, but part of the strength of circumscription is to allow exactly unstratified negation. This can occur when there are two facts (not predicates) which have in their derivation a dependency cycle containing two circumscriptions. Even if this is the case, it may not force this nondeterminism, depending on the action of the rules. The condition where one circumscription is present is dealt with via `call/cc`, presented next.

Even if a program has has nondeterministic circumscription, all is not lost. The engine will still emit a single, consistent world in which all constraints described in the program hold. In many cases, this may be sufficient, and the nondeterminism embedded in the internals of the computation, rather than in the output.

4.2.5 call/cc

The call/cc feature does not add any new syntax; it extends the interpretation of circumscription to deal with the case where simple fixpoint evaluation will fail to find an output set that complies with all provided rules. Consider the program

```
p(IntSet^union)
inconsistent: p(~{2}) <- ~p(~{1})
```

where we add `p({1})` to the database initially. Naively, we would oscillate between the two states `p({1, 2})` and `p({1})`, never finding an answer. With call/cc, we instead interpret `p({1, 2})` only to be the correct result moving forwards. Essentially, the engine will assert `p({2})` on the basis that we cannot proceed without `p({2})`. More specifically, matching `p( {1})` is matching on, among other things, $\neg p(\{2\})$. From this, we derive $p(\{2\})$. This fits the form of call/cc, so we add $p(\{2\})$ to the database, with call/cc as the provenance rather than a rule.

In general, this feature is intended so that a circumscription is allowed to extend itself. For example, if an analysis assumes it will be provided the complete control flow graph (circumscription) in order to perform SSA, then determine that a new value is possible for an indirect call (extending the control flow graph), call/cc is the component that allows the engine to retain the new control flow edge despite the fact that the old SSA form needs to be retracted.

4.3 Implementation

In order to evaluate the language as a means towards program analysis, we need a running implementation.

4.3.1 Holmes (Old Implementation)

Initially, I produced a database backed implementation which compiled down to a combination of Rust and SQL (initially C++ and SQL) and had Postgres handle joins, deduplication, and data storage. This had the advantage of being able to handle significantly larger working sets in theory, but in practice had significant performance issues which lead me to change approaches. Despite this, I feel it is worth discussing here both because the failures of the implementation point out some of the unique challenges and simplifications that can be made in evaluating Datalog, but also because it seems inevitable that to analyze programs substantially larger than those examined in this thesis, either a distributed platform or a disk-backed system will need to be used. It is my hope that these lessons learned will help a future external-database based implementor avoid the same pitfalls. Most of the details here are focused on Postgres, but other systems take a generally similar approach so similar problems are likely to occur.

As a result, this section is mostly focused on what went wrong, rather than on how the system was constructed. The source is available at `https://github.com/maurer/holmes`, but be aware that it does not represent a complete implementation of the language. In particular, it only has partial support for aggregation, and no support for circumscription.

Indices

Database software usually does not know which indices would be ideal to keep, and since keeping extra indices is is expensive in both time and disk, most SQL systems require the user to specify the indices to keep manually. Work is ongoing [59] to remedy this problem, but is not yet a production tool. In the meantime, if we wish our translated Datalog queries to run efficiently, the database must be provided with a list of indices to keep.

We tried a number of heuristics, including indexing in a global attribute ordering, indexing per query based on left-to-right joins, and just indexing all fields in order, and having the programmer reorder fields to boost performance. None of these approaches worked in practice. Both the global ordering and the left-to-right joins failed in large part because the query planner would choose to reorder the joins at runtime in multiple different ways. The programmer manually ordering fields could find local optima, but because predicates are used in multiple ways, it too falls short.

The solution in use at the time this approach was switched away from was to annotate the program with an explicit set of indexes to keep. We generated these indices by profiling the running program, and adding indices which would allow the query planner to avoid nested loops or full table scans where possible.

Append-Only, High Write

One interesting aspect of a Datalog system that the workload is entirely append-only other than retraction events, which are intended to be rare. This knowledge is unused by the database in executing queries. If it materializes a view to execute a query, and an underlying table is updated by an append, it will re-materialize the whole view, not perform any kind of incremental maintenance.

One of the expensive parts of many queries was insisting that it only return results which contained at least one *new* fact - one which hadn't been returned in this query before. That tables can only be appended to could enable the incremental maintenance of the join, allowing more efficient computation of the join, and retrieval of only the new data.

There are also some database schemas (such as the star schema) which become more possible in the absence of mutation or deletion.

Query Planning

Query planning, while of benefit to users who do not know all their SQL ahead of time, or whose tables remain in steady states, was the biggest issue with this approach. Databases commonly use a component called a query planner to translate SQL statements into an internal representation (loops, merge joins, hash joins, index walks, etc) that they can concretely execute. This component depends on a variety of information, including but not limited to:

- Whether the statement was prepared
- If prepared, how many times it has been executed
- What indices are available
- Information from the statistics daemon

Other than examining what indices are available, these conditions turn out to be highly anti-productive for a Datalog workload.

The statistics daemon is designed with the assumption that there is a sort of "steady state" for a database, in which the relative sizes of the tables will remain similar. This makes sense for usual customers of databases, but in our case, a large part of operation looks like heavy insert activity on a specific table. As a result, the statistics daemon's information is generally woefully out of date.

We prepare virtually all statements, since we intend to execute them repeatedly and want to avoid time in the parser. However, as of the time this system was developed, Postgres would ossify the query plan as of the 5th time a prepared statement was executed. This was done based on the assumptions that SQL connections do not live so long that the database changes a lot, so by the fifth time the query is run, the plan is unlikely to be improved, and performance will be increased by avoiding the planner entirely. In practice, this means that any recursive rule (like one marking nodes as reachable, or performing a dataflow) will have suboptimal performance. The rule executes five times, and during that time, the statistics daemon either has old out of date information, or even if it updates, information that the table it's reading out of is terribly small. The query planner then makes bad decisions based on this, then sets them in stone. As a result, indices sit there unused, and logarithmic operations are done linearly.

If the statements are not prepared, we incur parsing and planning overhead on every query. In practice, those costs were low in comparison to the troublesome queries. The true problem with completely non-prepared statements is that the query planner would rapidly change strategies, meaning that which indices are needed would change at different points in execution.

Since in our case we have a fixed query set and a rapidly changing database, it would most likely make more sense to absorb the query planner into the compilation process somehow. Postgres did not at the time of implementation have a way for a client to provide it with an explicit query plan short of building and providing a plugin which ran said plan as a function.

Star Schema

As alluded to earlier, one benefit of an append-only workload is that star schemas have lower overhead, as garbage collecting the child tables is not necessary. Star schemas are normally used for “data warehousing”, a sort of large scale database where an organization's data is all loaded into a single schema before being pulled out again into smaller databases for actual processing. The idea is that most values are referenced rather than included directly in tables. Warehousing engineers are largely interested in the standardization of these values and the resulting compression.

In our case, a star schema is interesting both for reasons of compression, and for ease of indexing. Indexing an IL instruction sequence[6], whether by hash or by ordering, is much slower than sorting by a tuple of integers. We discovered this technique after the pivot to an in-memory database, so I have no observations of its performance, but I expect it would help.

Large Objects

With an external database, the use of large objects becomes non-trivially expensive. If the database is local, and the bus between the program and the database is shared memory, this is not a major issue. However, even over a local unix socket, repeated accesses to large objects can inhibit performance.

This shows up in practice when dealing with binary sections and segments during lifting. If the lifting rule needs the segment, the architecture, and an offset into that segment to perform the lifting, this can incur several copies of the segment per instruction. In my sample programs, most segments were between 300k and 600k bytes, causing this to incur a nontrivial cost.

The first solution, specific to this problem, was an all-at-once chunking of the segment. We requested the segment from the database, then produced a 16-byte chunk (maximum length of an x86 instruction is 15 bytes) at every offset, and sent it back. In the future, requests would access this chunked data rather than the original. This resulted in a 16-17x blowup of the space to store the base binary, but as that paled in comparison to everything else it was not especially significant.

The second solution was to add another extension to Datalog allowing some functions to

---

[6] IL instruction sequence refers to a lifted representation of an assembly instruction or sequence of assembly instructions into an "intermediate language" which as fewer, more orthogonal operations.

exist as special external predicates to be run database side. These all needed to be builtins, and while the approach was slightly more efficient, overall I no longer think the improvement warranted the complexity.

If I were to address this again today, I would use a star schema database side, and implement a cache client side for fetched star objects.

4.3.2 Mycroft

Mycroft is a row-oriented, single-threaded, in-memory Datalog engine, taking into account the experiences of the initial implementation. It operates as a macro which transforms Datalog into Rust code, which can then be compiled into a running program. In its current form, it addresses most, though not all, of the pain points encountered with Postgres. The query planner is replaced by a single plan, generated at compile time, which parameterizes itself only on the size of the relevant tables at that moment. This replacement also means that we know precisely what indices will be useful, and can generate them. The join algorithm is aware of the incrementality of append only joins, and uses this to speed up requests for new results. As Mycroft is in process and in memory, large objects are not a problem. They are returned as read-only references to the existing structure, and can be operated on that way. The implementation is available at `https://github.com/maurer/mycroft`, and as a crate on `crates.io` for direct inclusion in rust projects.

Callbacks vs. External Predicates

By selecting callbacks instead of external predicates, we pushed some of the execution strategy back onto the programmer, rather than needing the engine to determine it. With even one external predicate, it will not be clear to the engine how to structure the search for solutions to the body most efficiently without hints. For example, if we have $p(x, y) \& q(x, z) \& r(x, z)$, where p and q are traditional predicates, and $r(-, +)$ is external, there are three possible match orders. We could match on p, then invoke r, then match on q, the reverse, or match on p and q, then invoke r. All have merits depending on the extents of p and q and the runtime characteristics of r. When a query is purely data, this simplifies into join processing (§ 4.3.2). Efficient join computation is still complex, but is a thoroughly studied area. With r as a black box however, any such processing order will be fundamentally heuristic. With multiple external predicates in the same rule, the problem is exacerbated, with more orders becoming possible, and another uncharacterized evaluation cost.

Using the callback method, the programmer knows the callback will only be invoked on a complete match, and may use an intermediate predicate to structure the matching strategy how they choose. Additionally, since the functions are mono-directional and occupy a special place in the language, this removes the complexity of mode checking from both the compiler and the author/reader of code. In our implementation, the staging is explicit: The body is matched in its entirety, any callback is run, and the head is instantiated.

This method does have one performance downside. As Holmes does not have any notion of a temporary predicate, any rules which would use two external predicates normally will potentially eat up additional space. Forcing tabling when two external predicates would be used may not be desirable in all cases. This may *improve* performance at times due to effectively tabling a subquery of the intended query, but it may also waste space and time in others.

Typed Storage

Rather than store data values directly in rows as was done in the PostgreSQL-based implementation, here we keep a separate deduplication table for each type of data on which we operate. This allows us to efficiently map back and forth between values, which the callbacks need to consume, and integer keys, which are convenient for indexing and join algorithms. This is reminiscent of the star schemas discussed before. As our system is mostly-append (other than retractions due to circumscription) we design this as an insert-only structure. An additional benefit, more relevant here than with Postgres, is that this greatly reduces our memory footprint.

At compile time, each type present in one or more predicates has a modified robin-hood hash table declared for it. This table has two pieces: a vector backing which stores the actual data, and a vector of hash/key pairs. There are two operations this table needs to support: acquiring the key for an object, whether or not it's present already, and acquiring an object from its key. Finding the key for an object is accomplished by using a lookup on the hash table portion of the structure, inserting into both the table and the vector of data in the event of a lookup failure. Finding the object for a key (the more common case) works by indexing into the vector.

The only principal difference between this and a simpler design (a standard hash table mapping from the value to the key, and a vector mapping from a key to a value) is that it stores the data only once, and without any indirection. This gave a modest 23% time performance boost over the standard library implementation in time, and approximately halved space on an earlier version of the use-after-free detector. The closest approach still using the standard data structure would have been to use a smart pointer to share data between the data structures, or a hash table of hashes. The smart pointer caused trouble with the interfaces, and hashing twice incurred a performance penalty, so we used this custom hash table design for deduplication and unique key assignment.

Aggregation

As aggregation is described at the predicate level, we can implement it directly on the tuple storage. Tuple storage is structured as a map from the tuple of non-aggregate fields (reordered to the front) to a tuple of aggregate fields. These aggregate fields are represented by a triple of the value-keys to be aggregated, a current aggregate value, and an index indicating how many of the value-keys are aggregated in the cached value. This allows for a lazily updated computation of the meet.

When a tuple is inserted into the store, if a value with the same non-aggregate fields is present, the value-key list is extended, but the aggregate is left alone. If it is not present, we initialize the aggregate value with the value of the key in that slot, fill in the key in the keys-to-be-aggregated, and set the index to 1. When retrieving a tuple, we check whether the index is equal to the length of the comprising keys. If it is not, we start the iteration at the index, and perform iterative meets until the aggregate is up to date. We then return the tuple, extended by the aggregate fields and reordered.

Join Computation

Datalog computation is join-heavy, and as a result attempting to compute the join naively can lead to disastrous execution times. There are a variety of existing join approaches. RDBMSes tend to favor straightforward strategies, such as nested looping, hash join, and merge join. Merge joins require a relevant index, but generally perform substantially better unless tables are extremely small. Hash joins operate by creating an intermediate data structure of one of the tables which is indexed by the hash of the joined values.

However, for high-arity join patterns, better algorithms exist, usually formulated as "worst-case join" algorithms. Ngo showed [58] that it is possible to develop join algorithms which are optimal even under these conditions. This algorithm is rather complex, and is intended for theoretical results rather than actual implementation. However, LogicBlox [7] developed an algorithm known as Leapfrog Trie-join [77] which achieves these same bounds while remaining practically implementable over traditional indices. Unfortunately, this algorithm is patented, and so could not be used. This indicates a potential for future implementations to derive a novel approach from the AGM [8] bound or Ngo's [58] approach, but developing such an algorithm is beyond the scope of this thesis.

In Mycroft, we used a simultaneous merge join ordered from smallest table to largest table. An index is selected for each table which walks it in unification argument order, with constant arguments being sorted to the front. The first index is advanced to the first tuple where all the constant arguments match. This is made easier by the use of integer-only tuples, as the non-constant arguments can be represented as 0 in a query to the index. Then, candidate variable bindings are made to the unification terms (if possible) and the next table is considered. When on the last table, if a candidate set of assignments to the unification terms can be completed, it is emitted and the index advanced by one step. If the index cannot advance or the index fails to unify with earlier tables, we know that no further result is possible, and go back one table, and continue. This approach keeps around only a small amount of additional state, linear in the number of atoms in the query, as it is returning results.

However, due to our need for *incremental* results, we can improve this mechanism substantially. Rather than computing the entire join at once, we split it into sub-joins, one for each atom in the query. We have a separate, much smaller index for "new" facts in referenced predicates, requested by the query at database initialization time. We perform a sub-join with each predicate's large index swapped for this small index to get exactly those results which we would receive that we did not before, then chain them for a result. The small indexes are emptied during this operation, so they will not yield the same results again.

As an example, consider evaluating the query $A(x, y)\&B(y, z)\&C(z, x)$ for incremental results. The first time it is evaluated, we perform a full join, ignoring the sub-join strategy - it would be equivalent to performing the full join 3 times. Then, we insert two facts into A, and one into C. Running the query again, we perform three sub-joins, one on A', B, C, one on A, B', C, and one on A, B, C'. In our join algorithm above, remember that we sorted the smallest table to the left. As a result, the join with B' immediately terminates, yielding no results. For the join with C', it essentially acts as a join on A and B only, with a constant restriction. The join with A' is similar, but the A' portion of the join yields two facts, so it essentially runs two constant constrained joins of B and C.

Provenance Tracking

In order to later manage circumscription, or to allow a human to trace the reasoning of a program, we need to keep track of where facts come from. To do this, in conjunction with each tuple we store a list of possible justifications. A justification is composed of the ID of a rule, and the IDs of the facts used to match the body of that rule. An aggregation is represented simply as the list of fact IDs aggregated for the match. A map is additionally maintained from fact IDs to justifications which contain them.

Circumscription, Call/CC, and Retraction

Implementing circumscription essentially involves monitoring accessed aggregations to see if they would change, and responding with a retraction. The previous description of aggregates does not easily allow for this. A tuple insertion does not know if something has depended on this aggregation's completeness, and if so what. To deal with this, if a tuple is circumscriptively fetched, we replace the list of merged keys in the aggregate field with a newly minted aggregate ID. Three maps are maintained for aggregate IDs:

- Aggregate ID to comprising Fact IDs
- Aggregate ID to dependent justifications
- Fact IDs to Aggregate IDs they comprise

If a tuple insertion occurs and would need to update an aggregate represented by an aggregate ID, that aggregate ID is retracted. The retraction code acts as a work list, initially populated by the broken aggregation. First, it removes any justifications broken by the current retracted item. Then, it retracts (by adding to the work list) any facts which now lack justification. If the current retracted item is a fact, it also retracts any aggregate IDs which now have one fewer fact.

In the special case where the tuple just inserted was also retracted, we replace its justification with one referencing the members of its now broken aggregate ID. This ensures that while this justification no longer cares about the expansion of the circumscription, it will still be properly retracted if one of the facts in the original aggregate becomes invalid.

Future Work

There is plenty of room for improvement in the concrete implementation of the language engine.

Currently, we keep more indices than are strictly necessary. Even with our current join strategy, the count of indices kept could be reduced through a mechanism to match attribute ordering between queries more frequently. With a more modern join like tetris join [44] it could even be possible to keep a single index per predicate.

Results of some sub-joins get used repeatedly, and can be known not to change through topological sorting. Currently, this is exploited through manual tabling - the creation of dummy predicates to keep the completed join as a realized structure. However, it should be possible to generate these temporary structures automatically in some cases.

Pivoting indices from a simple in-memory btree to a MVCC[7]-style structure would allow multiple worker threads to be evaluating rules at the same time. As modern systems generally have additional cores, this should lead to performance improvements overall (though degradation in bottleneck phases). This approach also meshes well with optionally backing some data structures with disk due to either large size or low traffic. Many MVCC trees are already designed as on-disk data structures due to their use in traditional RDBMS systems. Allowing some data to reside on disk would increase the maximum size of analysis the system could perform on a single binary, or allow for easier multi-binary analysis.

In an ideal world, this system could even be distributed. Other than circumscription and the decision to terminate, every component of this system can operate safely with a partial knowledge of the database. As a result, it seems plausible that with appropriate heuristics for shuffling and synchronization around circumscription, this language could be well suited for distributed execution.

---

[7] MVCC stands for Multiple Version Concurrency Control. MVCC trees are common in database design because they map well to the block-at-a-time disk update structure and because they allow for multiple transactions to act on the same index in a way that makes it clear if the index was invalidated while using it. They accomplish this by retaining any portion of the tree which is being accessed by some transaction, and garbage collecting as threads leave. This results in "multiple versions" of the tree being accessible simultaneously in order to deal with concurrency contention, thus the name.

Chapter 5

Holmes Specification

Holmes differs from base Datalog in callbacks (analogous to external predicates), aggregates, circumscription, and call/cc. External predicates already have a well understood semantics, and we will model callbacks as interpreted functions by extension of our Herbrand universe during instantiation (§5.3.1). We can represent aggregates in the presence of interpreted functions by a syntactic transformation, other than for purposes of circumscription. Adding call/cc forces a solution to exist when it otherwise would not, adding a restricted set of solutions which violate the minimality constraint of stable set semantics when no stable set exists.

We will first go over the negation model embedded at base Datalog to demonstrate how it works (§ 5.1). Then, we will show how to construct the Herbrand universe for a program with callbacks, aggregates, and circumscription (§ 5.3). Finally, we will give semantics for the Herbrand instantiation of a Holmes program which interprets circumscription as infinite negation, and call/cc as the failure of one of those negated terms (§ 5.4).

5.1 Negation

5.1.1 Other forms of Negation

The first primary distinction to draw in negation is *strong negation* vs *negation-as-failure*, or NaF. Strong negation describes having knowledge that something is not the case, whereas NaF describes knowing that we will never have proof that something is the case. This difference is more formally defined by Gelfond and Lifshitz [31].[1] In this work we will focus on negation-as-failure.

One way of interpreting negation is *stable-set* semantics [30]. In stable-set semantics, to check whether a proposed model is a stable set, we first take the reduct of the logic program under the proposed model. To do this, we remove all negated atoms in the body which are not present in the proposed model (and so are satisfied), and then remove all rules which still have negated atoms in them (which were not satisfied). If the model is the minimal model of this reduct (this new reduced program), then it is a stable set, or stable model of the program. This interpretation admits an arbitrary number of potential stable models, including none.

Another approach is *well-founded negation* [75]. This approach constructs what it calls *unfounded sets*. These are sets of values which, relative to a given partial interpretation, are unprovable. It then augments the partial interpretation with the negation of the unfounded set, and constructs a new minimal model, treating negated atoms as only true if explicitly present in the new partial interpretation. This procedure runs until it reaches a fixed point, and produces exactly one well-founded partial model, which will be a subset of all stable models [75].

In our case, stable-set semantics falls short due to the potential for no model, and the well-founded approach may return partial solutions if there is no stable set or multiple stable sets.

---

[1] This article refers to strong negation as classical negation.

5.1.2 Negation in Holmes

Traditional Datalog bodies consist only of a set of atoms to be simultaneously satisfied. In addition to this, we allow the atoms of the body to be optionally negated. In the original input program, the head may not be negated.

As we have no function symbols or interpreted functions, we define a Herbrand universe $\mathbb{U}$ which is simply the set of constants in the program. $\mathbb{B}$ is the set of every predicate instantiated at every combination of values in $\mathbb{U}$.

We instantiate the program into the Herbrand universe, making a version of each rule with every variable instantiated at every member of $\mathbb{U}$. This eliminates pattern matching and reduces all rules to a set of rules of the form

$$p \leftarrow \bigwedge q_i \wedge \bigwedge \neg r_j$$

where $p, q_i, r_i \in \mathbb{B}$.

For example, if we begin with the program

$$A(x) \leftarrow B(x) \wedge C(0)$$

$$B(1) \leftarrow \neg C(1)$$

$$B(1) \leftarrow \neg A(0) \wedge \neg A(1)$$

We only see the symbols 0 and 1 present. This means $\mathbb{U} = \{0, 1\}$ for this program, and $\mathbb{B} = \{A(0), A(1), B(0), B(1), C(0), C(1)\}$. Instantiating all variables in the program at all possible values from $\mathbb{U}$, we get a variable-free version:

$$A(0) \leftarrow B(0) \wedge C(0)$$

$$A(1) \leftarrow B(1) \wedge C(0)$$

$$B(1) \leftarrow \neg C(1)$$

$$B(1) \leftarrow \neg A(0) \wedge \neg A(1)$$

We can now remove parameters from predicates by assigning a name to each member of $\mathbb{B}$.

$$p \leftarrow q \wedge r$$

$$s \leftarrow t \wedge r$$

$$t \leftarrow \neg u$$

$$t \leftarrow \neg p \wedge \neg s$$

Now the program is in the general form described above.

In derived forms, we also allow rules of the form

$$\neg p \leftarrow$$

which simply asserts $\neg p$. We do not allow rules of this form in the initial program to ensure that the initial program is not directly contradictory. We will use N to describe a possibly negated term, e.g. it has form p or $\neg p$ where $p \in \mathbb{B}$.

For a program Π which consists of a set of such rules,

$$\frac{N_0 \leftarrow \bigwedge_{i \in I} N_i \in \Pi \quad \forall i \in I.\Pi \models N_i}{\Pi \models N_0}\text{ m.p.}$$

We will define an interpretation of negation in such programs via a Kripke frame. We say a program Π is consistent if it does not model both truth and falsehood for a single fact.

$$\frac{\forall p \in \mathbb{B}(\Pi).\Pi \models p \rightarrow \Pi \not\models \neg p}{C(\Pi)}\text{ consistent}$$

We say a program Π decides a fact if it models it as either true or false.

$$\frac{\Pi \models p}{\Pi \lhd p}\text{ decide-true} \qquad \frac{\Pi \models \neg p}{\Pi \lhd p}\text{ decide-false}$$

These rules equip us to describe our Kripke frame. A program which decides all predicates and is not inconsistent is a world.

$$\frac{C(\Pi) \quad \forall p \in \mathbb{B}(\Pi).\Pi \lhd p}{W(\Pi)}\text{ world-base}$$

Any program which is a subset of a world is a world.

$$\frac{W(\Pi') \quad \Pi \subseteq \Pi'}{W(\Pi)} \text{ world-subs}$$

We define a relation between worlds $\rightsquigarrow$ describing adding an assumption to the program on the left to arrive at the program on the right.

$$\frac{p \in \mathbb{B}(\Pi) \quad \Pi \not\models p \quad W(\Pi \cup (\neg p \leftarrow))}{\Pi \rightsquigarrow \Pi \cup (\neg p \leftarrow)} \text{ assume-false}$$

$$\frac{p \in \mathbb{B}(\Pi) \quad \Pi \not\models p \quad W(\Pi \cup (p \leftarrow)) \quad \neg W(\Pi \cup (\neg p \leftarrow))}{\Pi \rightsquigarrow \Pi \cup (p \leftarrow)} \text{ assume-true}$$

We complete the Kripke frame by defining the accessibility as transitive closure over $\rightsquigarrow$.

$$\frac{}{\Pi \leq \Pi} \text{ refl} \qquad \frac{\Pi_0 \leq \Pi_1 \quad \Pi_1 \rightsquigarrow \Pi_2}{\Pi_0 \leq \Pi_2} \text{ assume}$$

We consider a formula F true for an input program Π if $\Pi \models \diamond F$ under this frame. Here, $\diamond$ means possibly, so this condition says that it is possibly the case that this formula holds. We can visualize this on a directed graph with worlds as nodes and the single-step form of our accessibility relation ($\rightsquigarrow$) as an edge. This condition means that if we start from Π, our input program, we can find a connected program Π' so that $\Pi' \models F$. Conceptually, this means that there exists an allowed choice of assumptions so that F holds.

Example

Consider the program we Herbrandized previously, extended with the initial fact $\{r\}$, calling the initial program Π_0. Initially, we have that $\Pi_0 \models r$. We cannot derive any more from Π_0 because r is not sufficient to invoke any rule through m.p. We proceed by considering a candidate world $\Pi_1 = \Pi_0 \cup \{\neg u \leftarrow\}$. Through iterative application of m.p. we get

$$\Pi_1 \models r \wedge \neg u \wedge t \wedge s$$

p and q are still undecided, so we create candidate worlds

$$\Pi_2 = \Pi_1 \cup \{\neg p \leftarrow\}$$

$$\Pi_3 = \Pi_2 \cup \{\neg q \leftarrow\}$$

At Π_3, we have through iterative application of m.p.

$$\Pi_3 \models \neg p \land \neg q \land r \land s \land t \land \neg u$$

As a result, for all each predicate, Π_3 decides it, e.g. $\Pi_3 \lhd p$, $\Pi_3 \lhd q$, etc. either through decides-true or decides-false depending on its negation in the earlier formula. Specifically,

$$\forall p \in \mathbb{B}(\Pi_3).\Pi_3 \lhd p$$

Since Π_3 only derives the truth of r, s, t, and does not derive any of their negation, by the *consistent* rule we have $C(\Pi_3)$. With these combined, we can apply the *world-base* rule and get $W(\Pi_3)$. By *world-subset*, Π_0, Π_1, Π_2 are all worlds as they are subsets of Π_3.

We focus on the accessibility relation next. If we apply *assume-false* to Π_3 and the predicate q, we get $\Pi_2 \rightsquigarrow \Pi_3$. Repeatedly applying it yields

$$\Pi_0 \rightsquigarrow \Pi_1 \rightsquigarrow \Pi_2 \rightsquigarrow \Pi_3$$

Starting with *refl* on Π_0, we get $\Pi_0 \leq \Pi_0$. Iteratively applying *assume*, we get that $\Pi_0 \leq \Pi_3$.

Now we combine this information to show that what Π_3 models is a legal output for the input program. Recall that

$$\Pi_3 \models \neg p \land \neg q \land r \land s \land t \land \neg u$$

Since $\Pi_0 \leq \Pi_3$, we can additionally say that the above formula is possible at Π_0.

$$\Pi_0 \models \diamond(\neg p \land \neg q \land r \land s \land t \land \neg u)$$

call/cc

Adding the *assume-true* rule is what differentiates our negation model from stable-set semantics. This rule allows the addition of positive assumptions (i.e. assuming p as opposed to only being able to assume $\neg p$). We specifically restrict this rule so that it requires that p

is not decided, and $\neg p$ would prevent a complete, consistent truth assignment to predicates. By restricting *assume-true* we still rule out trivial solutions like setting everything true, but in a looser way than the minimal model over the reduct (§ 5.1.1) It is looser because it allows for models which are not minimal over the reduct. When applying *assume-true*, it adds a true proposition which has no derivation from the rules. This proposition would not be present in any minimal model. As *assume-true* is only allowed in cases where *assume-false* would lead to no solution, it still prohibits the trivial solution (all predicates are in the model).

How Negations Differ

We consider the program consisting only of the rule $P \leftarrow \neg P$ to concretely examine the difference between these negation strategies.

In the case of a stable-set semantics, this can have no model. The only two candidate models are $\{P\}$ or $\emptyset$. Under the first model, the rule is not satisfied, so the reduct is the empty program. This reduct does not support P, and so it is not a minimal model, and not a stable set. Under the second model, the reduct is $P \leftarrow$, and so it is not a model of the reduct because P must hold.

Under well-founded negation, the unfounded set is empty, because P still appears on the left hand side of the rule. As a result, evaluation terminates immediately. Well-founded negation returns the *partial model* $\emptyset$.

Following our negation system, we begin with

$$\Pi = \{P \leftarrow \neg P\}$$

Now, we define two candidate worlds,

$$\Pi^- = \Pi \cup \{\neg P \leftarrow\}$$

$$\Pi^+ = \Pi \cup \{P \leftarrow\}$$

In the case of Π^-, we have both $\Pi^- \models P$ and $\Pi^- \models \neg P$, so $\neg C(\Pi^-)$. As a result, $\neg W(\Pi^-)$. For Π^+, we have only that $\Pi^+ \models P$, so $C(\Pi)$. Since P is the totality of $\mathbb{B}\Pi^+$, $W(\Pi^+)$ by *world-base*. By *world-subs*, $W(\Pi)$. We can apply *assume-true*, using $W(\Pi^+)$, $\neg W(\Pi^-)$, and that Π does not decide P. This gives $\Pi \rightsquigarrow \Pi^+$. Finally

$$\Pi^+ \models P$$

Since $\Pi \rightsquigarrow \Pi^+$, by *refl* and *assume* $\Pi \leq \Pi^+$, so

$$\Pi \models \diamond P$$

This simple example shows the differences between the approaches. In the face of this kind of uncertainty, stable-set semantics will choose not to give an answer, well-founded semantics will not decide either way, and Holmes-style semantics will allow call/cc to justify a fact in order to return a consistent complete answer.

In Holmes, circumscription corresponds to a potentially-infinite variety of *assume-false* in which the program assumes false every version of a specific aggregate greater than the proposed circumscription. call/cc corresponds to *assume-true*, but is a bit more complex. With finite *assume-false*, the *assume-true* rule can assert as true the undecided predicate for which *assume-false* failed. Since circumscription assumes false for multiple predicates, call/cc must select a specific predicate amongst those which has itself given rise to a contradiction to assert as true.

5.2 Example Input Program

The subsequent sections will describe how to support a full Holmes program rather than the toy subset above. We will use the following Holmes program as an input example to show how to perform each translation step more concretely.

Listing 5.1: Holmes Code

```
P(Int)
```

```
Q(Int)
R(Set(Int)^union)

inc: Q(y) <- P(x) + (y = inc(x))
promote_R: R(s) <- Q(x) + (s = promote(x))
check_threshold: P(x) <- R(s) + (x = threshold(s))
force_odd: Q(x) <- ~R(s) + (x = all_even(s))
```

where in pseudocode,

Listing 5.2: Procedural Pseudocode

```
fn inc(x: Int) -> [Int] {
  return [x + 1]
}

fn promote(x: Int) -> [Set(Int)] {
  return [Set.singleton(x)]
}

fn threshold(s: Set(Int)) -> [Int] {
  if s.sum() > 7 {
    return [101]
  } else {
    return []
  }
}

fn force_odd(s: Set(Int)) -> [Int] {
  if s.all(is_even) {
    return [s.max() + 1]
  } else {
    return []
  }
}
```

In Listing 5.1, we have explicitly written the input and output variables for clarity. In the actual existing Holmes implementation, these rules would lack those bindings. The variables bound in the body and free in the head would imply the inputs and outputs:

```
force_odd: Q(x) <- ~R(s) + all_even
```

This program has three predicates - P, Q, and R. P and Q range over integers. R ranges over sets of integers, and aggregates those sets via union. Since R has no non-aggregate parameters, every fact added to R will aggregate. The `inc` rule causes Q to contain the next integer for every member of P. The `promote_R` rule creates singleton sets from every member of Q and puts them into R. `check_threshold` adds $P(101)$ if R has a set which sums to more than 7. Finally, `force_odd` will add an odd number to R's set if it does not contain one by adding one more than the maximum of the set. Since `force_odd` circumscribes on R, it should do this only if the final result would otherwise contain only even numbers. The rule would do nothing on an intermediate state, unlike `check_threshold`.

5.3 Herbrandization (Revisited)

In the previous setting, we did not have any form of external code, and so could not produce new symbols. Now, we have both interpreted functions and lattice joins. We first show how to define the Herbrand universe and base in the presence of these complications. Then, we show how to instantiate a Holmes program at a Herbrand base constructed this way. In the previous setting, this just eliminated variables in the rules. Now, we also remove the callbacks and aggregates.

5.3.1 Herbrand universe

Instead of function symbols which produce new values, as in a traditional construction, we have interpreted functions and lattice joins. This is different from the normal construction because both interpreted functions and lattice joins may produce values which are equal to existing values. To address the equality issue, we assume that our construction receives the implementations of the callbacks and join operations as the real program would, and actually execute them on input values rather than creating symbolic expansions.

Define U_0 to be those constants present in the program, combined with varieties tupled up to the maximum arity of the provided functions. Consider joins as equivalent to callbacks which just happen to return only one argument. Let F be the set of functions, modified to take tuples for multiple arguments, and to return their results "flattened", e.g. if a callback would return $a = 1, b = 2$ and $a = 3, b = 4$, its representative in F returns $\{1, 2, 3, 4\}$. Since both joins and callbacks are typed, if an input would be outside their domain, they return $\emptyset$.

Given U_i, construct U_{i+1} by

$$U_{i+1} = \bigcup_{x \in U_i, f \in F} f(x)$$

We then combine these stages:

$$\mathbb{U} = \lim_{j \to \infty} \bigcup_{0 \leq i < j} U_i$$

The resulting universe will only be finite if the closure of all the functions and lattice joins over the original symbols is finite. This is unlikely to be the case in practical programs: even adding an "increment" function is sufficient to cause an infinite universe here.

We construct the Herbrand base $\mathbb{B}$ by instantiating each predicate at every value of $\mathbb{U}$ and adding a $\perp$ value. When determining what facts are true, this $\perp$ acts just like any other member of the base. We will later make use of $\perp$ in order to indicate that the program contradicts a circumscription and is inconsistent.

Example We start with the extensional database $P(1), P(3), P(5)$ and the program from Listing 5.1. Beginning with Herbrandization, we have

$$U_0 = \{1, 3, 5\}$$

since no other constants are present in the rules. Applying all function symbols possible to U_0 gives us

$$U_1 = \{2, 4, 6, \{1\}, \{3\}, \{5\}\}$$

Repeating the process gives

$$U_2 = \{3, 5, 7, \{2\}, \{4\}, \{6\}, \{1, 3\}, \{1, 5\}, \{3, 5\}\}$$

and so on, so that $\mathbb{U}$ is the set of all integers ≥ 1 and all sets of such integers. To create $\mathbb{B}$, we instantiate all predicates in against compatible values in $\mathbb{U}$, giving

$$\mathbb{B} = \{P(1), P(2), \cdots, Q(1), Q(2), \cdots, R(\{1\}), R(\{2\}), \cdots, R(\{1, 2\}) \cdots\}$$

5.3.2 Program Instantiation

For predicates which have aggregation (§ 4.2.3), rewrite them as rules with interpreted functions. Recall that

$$P(\tau_0 \cdots \tau_m, \sigma_0 \wedge j_0 \cdots \sigma_n \wedge j_n)$$

means we have a predicate P, which when there are two instances of P for which the first $m + 1$ arguments are equal, the latter $n + 1$ arguments will have their information merged by a lattice join operation indicated by j_i.

To translate the above form, we create a new function

$$j(a_0 \cdots a_n, b_0 \cdots b_n) = \{c_0 = j_0(a_0, b_0) \cdots c_n = j_n(a_n, b_n)\}$$

and add the rule (§ 4.2.2)

$$P(x_0 \cdots x_m, c_0 \cdots c_n) \leftarrow P(x_0 \cdots x_m, a_0 \cdots a_n), P(x_0 \cdots x_m, b_0 \cdots b_n) + j$$

where $+j$ as in the informal Holmes description means to attach the function j to run after the match to generate a set of assignments to variables in the head not bound by the body.

We define a partial ordering over facts in aggregate predicates based on the lattices used to combine their fields. For a predicate P defined as above, and

$$P_a = P(x_0 \cdots x_m, y_0 \cdots y_n) \tag{5.1}$$

$$P_b = P(x_0 \cdots x_m, z_0 \cdots z_n) \tag{5.2}$$

we define the partial ordering as

$$P_a \leq P_b \leftrightarrow \bigwedge_{i=0}^{n} y_i \leq z_i$$

where the lattice defined by the join operator j_i defines the partial order for the aggregate field with index i. If the first $m+1$ fields do not match, then P_a and P_b are incomparable.

For each aggregated predicate P, introduce an additional predicate P_c with the same field types. P_c will act as a concrete representation that P at the same fields as P_c would be a maximum on the ordering we defined. Essentially, it says that if $P_c(\vec{x}, \vec{y})$, then if $P(\vec{a}, \vec{b})$ is true, $P(\vec{a}, \vec{b}) \leq P(\vec{x}, \vec{y})$, or there is no ordering between them.

Beginning with the Holmes program after translating aggregations as above, replace of all circumscripted matches to P with matches against P_c *and* P at the same indices. Finally, for each P_c, add a rule

$$\bot \leftarrow P_c(\vec{x}, \vec{y}) \wedge P(\vec{x}, \vec{z}) + \text{P-less-than}$$

where P-less-than checks whether $P_c(\vec{x}, \vec{y}) < P(\vec{x}, \vec{z})$ (defined via $\leq$ and $\neq$) and returns a singleton list of no assignments if so, and an empty list otherwise. Here, we are using the $\bot$ value added to the Herbrand base to indicate that the fact matched by the P portion contradicts the circumscriptive guess matched by the P_c portion. This essentially adds a rule saying that an upper bound proposed by P_c must not contradict known information about P.

After translating aggregation and circumscription, the program looks like normal Datalog but with functions attached to some rules. For every element of the Herbrand universe, instantiate the rule, run the function concretely on the variables bound in the body, and instantiate the head term. This will result in a rules of the form

$$p \leftarrow \bigwedge q_i$$

where $p, q_i \in \mathbb{B}$.

Example We now focus on instantiating the program given in Listing 5.1 based on the universe from the previous section.

Instantiating inc and promote_R gives

$$\begin{aligned} Q(2) &\leftarrow P(1) \\ Q(3) &\leftarrow P(2) \\ &\vdots \\ R(\{1\}) &\leftarrow Q(1) \\ R(\{2\}) &\leftarrow Q(2) \\ &\vdots \end{aligned}$$

which is analogous to what we did in the finite example.

The guards and combination rules for our aggregated predicate R look like

$$\begin{aligned} R(\{1,2\}) &\leftarrow R(\{1\}) \wedge R(\{2\}) \\ R(\{1,3\}) &\leftarrow R(\{1\}) \wedge R(\{3\}) \\ &\vdots \\ \bot &\leftarrow R_c(\{1\}) \wedge R(\{2\}) \\ \bot &\leftarrow R_c(\{1\}) \wedge R(\{3\}) \\ &\vdots \end{aligned}$$

The first half describes the instantiation of the "union" aggregator on R. The second half has the guard rules which derive $\bot$ if the program exceeds a bound.

The rule check_threshold translates to

$$\begin{gathered}
P(101) \leftarrow R(\{8\}) \\
P(101) \leftarrow R(\{9\}) \\
\vdots \\
P(101) \leftarrow R(\{1,7\}) \\
\vdots
\end{gathered}$$

Since assignments are not generated when the condition is not met (value sums to > 7), this is a rule for each set which has a sum more than 7.

Finally, force_odd instantiates as

$$\begin{aligned}
&Q(3) \leftarrow R(\{2\}) \wedge R_c(\{2\}) \\
&Q(5) \leftarrow R(\{4\}) \wedge R_c(\{4\}) \\
&Q(5) \leftarrow R(\{2,4\}) \wedge R_c(\{2,4\}) \\
&\quad\vdots
\end{aligned}$$

It again only generates rules if there was an output from the function, and adds an odd number to R if it doesn't already have one by adding the max plus 1.

5.4 Semantics

We begin by defining a few extra sets relative to the initial program which we will use for interpretation. Let $K(\Pi) \subseteq \mathbb{B}(\Pi)$ be the set of circumscripted facts added, e.g. they were of the form $P_c(\cdot)$.

Let $A(\Pi)$ be a set of tuples of an aggregated predicate and all of its non-aggregated inputs. For example, for $P(x_0, \cdots x_n, y_0, \cdots y_m)$ and all y are aggregate fields, $P(x_0, \cdots x_n)$

for each possible value of x_0 through x_n would be present in $A(\Pi)$. We will refer to members of $A(\Pi)$ as *aggregates*.

Let $K_a(\Pi)$ where $a \in A(\Pi)$ be the set of circumscribed facts which correspond to the aggregation a. For example, if $a = P(x_0, \cdots x_n)$, then $K_a(\Pi)$ would contain $P_c(x_0, \cdots x_n, y_0, \cdots y_m)$ for all possible values of y_i.

Let $D(\Pi, c)$ where $c \in K(\Pi)$ be the non-circumscribed version of the fact, e.g. if c corresponds to $P_c(x_0 \cdots x_n)$, then $D(\Pi, c) \in \mathbb{B}$ corresponds to $P(x_0 \cdots x_n)$.

Most of this construction should look familiar from the simple negation semantics.

$$\frac{\Pi \not\models \bot}{C(\Pi)}\ \text{consistent}$$

Like previously, consistency here requires that none of the negations assumed by a circumscription also be present in the model in a non-negated form. Since we added special guard rules when introducing the P_c predicates, we can just check that $\bot$ cannot be directly derived to check that the program does not violate any circumscriptive assumption.

Rather than deciding individual facts as we did previously, we now decide aggregates.

$$\frac{\Pi \models c \quad \Pi \models D(\Pi, c)}{\Pi \lhd a}\ \text{bounded} \qquad \frac{\forall c \in K_a(\Pi). \Pi \not\models D(\Pi, c)}{\Pi \lhd a}\ \text{no-base}$$

The bounded case indicates that we have circumscribed this aggregate, and so have decided it. The no-base case indicates that this aggregate is totally unpopulated by the program. As we cannot circumscribe over it, the program decides the aggregate. Specifically, the program has decided on the negation of all values which could make up the aggregate.

Our *world-base* rule is similar to the finite case. If a program decides all aggregates (rather than all predicates) and it is consistent, then it is a world.

$$\frac{\forall a \in A(\Pi). \Pi \lhd a \quad C(\Pi)}{W(\Pi)}\ \text{world-base}$$

As before, any subset of a world is also a world

$$\frac{\Pi' \subseteq \Pi \quad W(\Pi)}{W(\Pi')}\ \text{world-subs}$$

We define accessibility stepwise again. If an aggregate a isn't decided yet, and circumscribing it would not cause any inconsistency, we may do it.

$$\frac{\Pi \not\vartriangleleft a \quad c \in K_a(\Pi) \quad W(\Pi \cup c)}{\Pi \rightsquigarrow_a \Pi \cup c}\ \text{circ}$$

The *circ* rule is analogous to *assume-false* because adding c to Π is essentially adding $\forall p \in V(\Pi, c).\neg p$.

To support the `call/cc` feature of Holmes (§4.2.5), let Π_c be $\Pi \cup c$ with all rules of the form $\bot \leftarrow c \wedge p$ removed. In other words, Π_c is $\Pi \cup c$, but will ignore inconsistencies derived from the addition of c.

$$\frac{\Pi \not\vartriangleleft a \quad W(\Pi \cup p) \quad \nexists c' \in K_a(\Pi).\Pi \rightsquigarrow_a \Pi \cup c' \quad \Pi_c \models \diamond p}{\Pi \rightsquigarrow_a \Pi \cup p}\ \text{call/cc}$$

This rule says that if we cannot construct $\rightsquigarrow_a$ using the *circ* rule, but Π_c would extend the aggregate a with new information, then we can extend a with the new information even though the proposed circumscription would not result in a world. The *call/cc* rule is analogous to the *assume-true* rule. It is only accessible when *circ* (which was similar to *assume-false*) is not, and allows adding a single $p \in V(\Pi, c)$ as an otherwise-unsupported choice to allow deduction to proceed.

We define $\leq$ based on $\rightsquigarrow$ inductively as before. Finally, we hold a formula to be true for a Holmes program if after translating it as above to a program Π, $\Pi \models \diamond F$.

Example In the previous section, we finished instantiating Listing 5.1 as normal Datalog. Call that base program Π_0. We have extensionally that $\Pi_0 \models P(1) \wedge P(3) \wedge P(5)$. Applying inc and promote_R gives us that $\Pi_0 \models$

$$Q(2), Q(4), Q(6)$$
$$R(\{2\}), R(\{4\}), R(\{6\}), R(\{2,4\}), R(\{2,6\}), R(\{4,6\}), R(\{2,4,6\})$$

Adding check_threshold gives additionally that $\Pi_0 \models$

$$\begin{aligned} P(101) \\ Q(102) \\ R(\{102\}), R(\{2,102\}) \cdots R(\{2,4,6,102\}) \end{aligned}$$

At Π_0, force_odd does nothing, since R_c has no members.

In this case, we have only one aggregate: $R(\cdot)$. As a result, there are only two ways forwards - circumscribe or call/cc that aggregate. To use *call/cc*, we need to know that circumscribing does not result in a world, so we start by circumscribing. Define $\Pi_1 = \Pi_0 \cup R_c(\{2,4,6,102\})$. Since both $\Pi_1 \models R_c(\{2,4,6,102\})$ and $\Pi_1 \models R(\{2,4,6,102\})$, $\Pi_1 \triangleleft R(\cdot)$ by the bounded rule. However, with the addition of the circumscription, force_odd now adds $\Pi_1 \models$

$$\begin{aligned} Q(103) \\ R(\{103\} \cdots R(\{2,4,6,102,103\}) \\ \bot \end{aligned}$$

One of the guard rules for the circumscription applied because $\{2,4,6,102,103\} > \{2,4,6,102\}$ in the ordering induced by union. Π_1 is inconsistent because $\Pi_1 \models \bot$. As a result, Π_1 is not a world.

Circumscription failed in this case, so it is time to apply *call/cc*. We construct Π_{1c} as in the rule - we delete all the guard rules for $R_c(\{2,4,6,102\})$ from Π_1. Since those were the only rules which could derive $\bot$, and Π_1 decided $R(\cdot)$, we can say that by *world-base*, $W(\Pi_{1c})$. In order to apply *call/cc*, we now need to pick a conclusion of Π_{1c} to pull out, identified as p in the rule. In this case, we will choose $Q(103)$. We define

$$\Pi_2 = \Pi_0 \cup \{Q(103)\}$$

At Π_2, we still don't decide $R(\cdot)$, so we try to circumscribe it closed again, this time with $R_c(\{2,4,6,102,103\})$. Define

$$\Pi_3 = \Pi_2 \cup \{R_c(\{2,4,6,102,103\})\}$$

This time, no new rules fire, since 103 is odd. Since Π_3 decides $R(\cdot)$ and is consistent (no guard rules fire), $W(\Pi_3)$.

Now that we have our final world, we need to work our way back to Π_0. Since $\Pi_0 \subseteq \Pi_2 \subseteq \Pi_3$, and $W(\Pi_3)$, by *world-subs* we get $W(\Pi_0)$ and $W(\Pi_2)$. Applying *circ*, we see that since Π_2 and Π_3 are both worlds, Π_2 does not decide$R(\cdot)$, and Π_3's change from Π_2 is to add the circumscription,

$$\Pi_2 \rightsquigarrow_{R(\cdot)} \Pi_3$$

To apply *call/cc*, we need to demonstrate that $R(\cdot)$ cannot be circumscribed at Π_0. Π_1 is not a world, so it cannot circumscribe $R(\cdot)$ at $\{2, 4, 6, 102\}$. Picking any other value to circumscribe at would either not decide $R(\cdot)$ because it cannot prove $\Pi_0 \models D(\Pi_0, c)$ if bigger or incomparable, and would trigger an inconsistency rule if smaller. As a result,

$$\not\exists c' \in K_{R(\cdot)}.\Pi_0 \rightsquigarrow_a \Pi_0 \cup c'$$

Since $\Pi_0 \not\Delta R(\cdot)$, $W(\Pi_2)$, the above, and $\Pi_{1c} \models Q(103)$, we can apply *call/cc* to get

$$\Pi_0 \rightsquigarrow_{R(\cdot)} \Pi_2$$

Since $\Pi_0 \rightsquigarrow \Pi_2 \rightsquigarrow \Pi_3$, $\Pi_0 \leq \Pi_3$. We get as our final answer that $\Pi_0 \models \diamond$

$$\begin{aligned} P(1), P(3), P(5), P(101) \\ Q(2), Q(4), Q(6), Q(102), Q(103) \\ R(\{2\}), \cdots R(\{2, 4, 6, 102, 103\}) \end{aligned}$$

Chapter 6

Alias Analysis

In order to show that Holmes is useful in practice, we implement a concrete system with it. We chose to focus on the problem of use-after-free, as it is a little-explored area for static binary analysis. We used Holmes here to link together control flow analysis, alias analysis, string recovery, and general function loading in order to build a working use-after-free engine.

There were 238 use-after-free (UaF) vulnerability disclosures (CWE-416) issued in 2017 alone, with 36.2% given a critical security rating. Use-after-free bugs happen when a program frees a pointer and subsequently writes to or reads from the memory pointed at. Use-after-free bugs can lead to DoS, control flow hijack, and information leaks.

Despite the number of CVEs, few tools exist that can automatically and statically detect such bugs in off-the-shelf *binary* code. However, there are tools for finding such bugs in *source* code [13, 27]. Requiring source code limits the applicability of these techniques to developers with full source access.

In this chapter, we focus on the question "Can we use Holmes to bridge the gap between analysis for UaF bugs in source code versus compiled code?" In particular, previous work has been unable to apply source code techniques [6, 38, 42, 71] to compiled code. Can we adapt such techniques to be effective even without source?

At a high level, UaF bugs require reasoning about memory allocation and memory aliasing. Source code techniques are more plentiful due to the rich and mature research area of alias, points-to, and similar schemes for reasoning about memory over the lifetime of a program. In comparison, at the binary level the primary approach for reasoning about memory is Value Set Analysis [11], which is less mature and has limitations in practice such as inability to reason about all arithmetic operations (e.g., bit-shifts and division) and the fact it may not terminate without ad-hoc widening in the presence of loops. For example, GUEB [28] proposed to detect UaF bugs using VSA, but is handicapped by disallowing cyclic paths to allow rapid termination of VSA.

We present a new binary-level static analysis approach for detecting UaF bugs in executable programs. One of the main technical challenges we address is showing how to adapt source-code memory alias analysis to compiled code, where previous work has instead created all-new binary-only approaches to alias analysis like VSA. We experiment with two classes of analysis: flow insensitive alias analysis via a Steensgaard-type [71] algorithm, and flow-sensitive alias analysis using a data flow approach adapted from Andersen [5] style analysis with added rules to handle binary-specific details such as calling conventions and computed addresses. We also add context-sensitivity by allowing the analysis to reason about the call-stack discipline followed by executable code, and a type of field sensitivity appropriate for direct pointer arithmetic without type information. To the best of our knowledge, there is little work in applying the literature in source alias analysis to compiled code, and no previous work has shown how to then use such techniques to find UaF bugs statically in compiled code.

We have built a tool called Marduk that uses Holmes to drive the different levels and co-dependencies in binary analysis, alias analysis, and UaF detection. Taking this approach allows us to have an end-to-end reasoning chain from input binary to why this particular candidate use-after-free could not be disproven with a given sensitivity.

We evaluated Marduk over 7 real CVEs and the Juliet test suite released by IARPA for purposes of verifying our detection capabilities and measuring false positives in the face of bugs. Additionally, we measured false positive rates against a background of expected-good binaries (we assume no true positives): a random sampling from the `$PATH` of a default

Ubuntu installation.

Marduk is available at `https://github.com/maurer/marduk`.

6.1 Analysis Design

First, our system transforms the input binary into structured semantics we can work with throughout the rest of the process. Second, we perform alias analysis over the structured representation. This allows us to know all the pointers which may point to freed memory after each free. Finally, we look for reads and writes through potentially freed pointers to create our list of candidate use-after-frees.

6.1.1 Alias Analysis

Alias analysis consists of computing the possible ways to access different variables. If dereferencing two expressions may access the same variable, they alias.

Our machine-level alias analysis involves three main design choices:

1. Selecting variables. We describe alias analysis over program variables, but unlike C, the assembly from a binary does not contain variable information.

2. Selecting sensitivity. We can return points-to information parameterized on different pieces of context. Common examples of parameters are flow sensitivity (program location), context sensitivity (call stack), field sensitivity (offsets within memory regions), and object sensitivity (possible construction sites for a "this" pointer). In this work, we examine, flow, context, field, and recency [11], how to implement them on compiled code, and their effects on precision and performance.

3. Solving for points-to relationships. The generation of the aliasing information is undertaken differently for flow-insensitive vs flow-sensitive varieties of alias analysis. The flow-insensitive variety uses a constraint satisfaction technique and simultaneous solv-

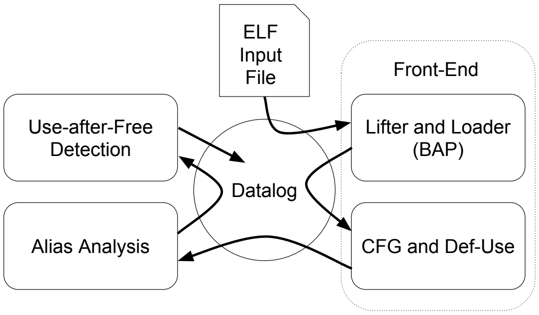


Figure 6.1: System Diagram

ing via Steensgaard [71]. Flow-sensitive varieties use the inter-procedural dataflow described in §6.1.5.

6.1.2 Front-End

Before we can apply our analysis, we first need to transform the raw bytes of the input file into a form similar to the AST[1] received after parsing. Techniques in this section are not novel; we describe them for completeness to understand our overall system.

The front end:

- Parses the executable file container, e.g., ELF on Linux
- Finds potential function entry points
- Builds a control flow graph
- Provides an intermediate language with the semantics for every reachable instruction
- Builds a call graph and call-site information

We use the BAP [16] framework to provide ELF parsing, entry point identification, and instruction semantics.

---

[1] Abstract Syntax Tree

Variables. In our approach, we use 3 kinds of variables:

- Stack slots, parameterized by which function they are in. Written `sp+off@func`
- Registers
- Dynamic Allocations, parameterized by their allocation site. Written `dyn@loc`

We use an abstract location in these parameterizations. In the context-insensitive case it is simply the address of the instruction, and in the context-sensitive case it is the pair of the address with the return stack.

Our choice of dynamic allocation variables defines our heap model. We assume that two memory regions may only alias if they share an allocation site. This assumption matches reality unless the program releases pointer back to the allocator via free, but then uses it afterwards (a use-after-free bug). This may violate the assumption because a write to or read from the now freed pointer may alias with newly returned memory. As a result, this heap model is correct at least until the first use-after-free in an execution. As we are not performing a value analysis, we also assume all accesses are in-bounds. Essentially, if a use-after-free is the *first* memory violation to occur, we will locate it.

Calculating Update Summaries. In order to avoid analyzing the full complexity of IL instructions within alias analysis, we first transform them to contain only the relevant dataflow information. Our update summaries are then a description of the action of an IL instruction on the points-to relationship, where `a` and `b` are variables as defined above:

- `a = b`
- `a = *b`
- `a = &b`
- `*a = b`
- `*a = *b`

- `*a = &b`

We generate a list of these constraints for each instruction and associated with the location of the instruction. At allocation sites we emit the summary `a = &dyn@loc`, meaning that the variable `a` (usually `RAX`) takes on the address of the allocation region corresponding to that location.

6.1.3 Insensitive Analysis

In the flow-insensitive case, we modify the summaries before use to adapt our variable selection to better fit the problem. Specifically, we annotate registers with definition sites, similar to SSA form.

We compute the possible definition sites of each register on the right side of a summary, and clone the summary for every possible definition. If the left side of a summary contains a register, we parameterize it with the location it came from. We do this because a single register may hold multiple different logical variables at different points of time. The register location parameterization allows us to avoid every definition of a register being potentially readable from every other site in the program. Without our approach, the resulting alias sets would be too imprecise to be useful as a register's alias set (e.g., `RAX`) would include information about all variables ever assigned to `RAX` by register allocation. Notably, this would include every call to `malloc`. This parameterization adds a little bit of flow sensitivity even in otherwise insensitive analysis.

We then aggregate all update summaries from the program and solve them by equality via Steensgaard's algorithm. Steensgaard runs in almost linear time [71] making it possible to compute over nearly any binary, though is less precise than flow-sensitive analysis, described below. We use insensitive analysis as a baseline to help quantify the additional precision derived by adding additional sensitivity.

6.1.4 Adding Flow Sensitivity

We structure our flow sensitive analysis as a dataflow problem. At each assembly instruction, we use a transfer function based on Andersen's [5] inclusion-style analysis. We use the interprocedural dataflow adaptation described next (§6.1.5). The same rules and functions handle both context-sensitive and context-insensitive analysis, as the Location's stack context is optional.

A dataflow analysis is defined by a transfer function which calculates how a statement should update the dataflow facts (alias sets in our case), a set of control flow edges to walk from a set of starting points, and a meet function with specifies how to merge information sets at control flow graph confluence points. We describe these below.

By default, alias analysis calculates alias sets even when a variable is dead. We run a pass before alias analysis to determine which variables are live. We use this information to remove dead points-to information to improve performance and precision. This technique increases performance because we do not waste time and space updating and tracking points-to sets which cannot affect the outcome. It also increases precision because if no pointers to a given allocation exist any longer, we know any new pointer to that allocation points to copy of that region which does not need the information from the old copy.

Transfer Function. The transfer function processes the update summaries. We follow Andersen as shown in Listing 6.2:

- Definitions of variables are destructive
- Apply writes through variables to each value they may point to.
- Right hand sides go through 0, 1, or 2 levels of dereference for `&b`, `b`, and `*b` respectively to generate the set to update with

Figure 6.2 shows the transfer function rules for each statement type. The "Initial" section shows a sample initial configuration a points-to relation might have. Both `a` and `b` are two-level pointers, with `x` and `y` representing the possible things pointed to at each

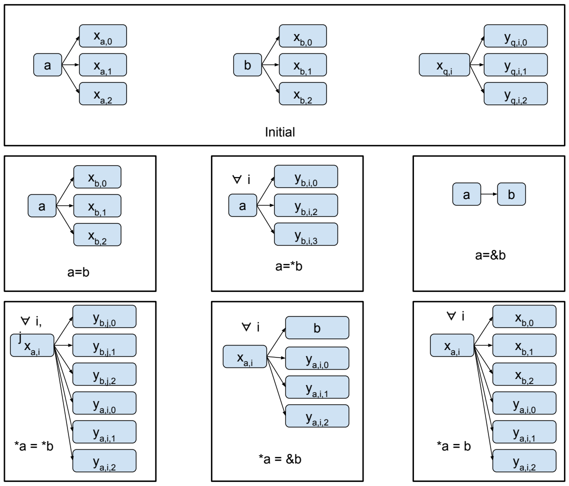


Figure 6.2: Points-to Updates

level. The chart labels each other box by an update summary, and shows what would change about the points-to relation. The shown transfer function is for field-insensitive analysis; as each summary is not tracking offsets, we cannot perform a destructive update to anything dereferenced on the left hand side.

For updates to stack slots or registers, we can perform a "destructive" update. The `a =` examples in Figure 6.2 demonstrate this style of update. This means that since we know the new value is the only value the variable could have, we can *replace* the points-to set. In the case of pointer write though, we cannot do this, because our analysis is not field sensitive.

A field sensitive analysis extends this destructive update logic to writes through pointers with a points-to size of one, allowing for more destructive updates and increasing the precision of our analysis. We will explain and add field sensitivity shortly in section 6.1.8, but until then, pointer updates need to remain non-destructive.

After we complete the updates to the points-to structure, we remove all variables which are no longer live according to the previously computed information. The program will not use these variables, so tracking them is imprecise and expensive.

Finally, we perform a mark-and-sweep garbage collection of the tracked pointers, using stack slots, registers, and anything reachable at the entrance of the function as roots. This

Listing 6.1: Flow Sensitive Pointer Analysis Rules

```
flow_in(Location, PointsTo^pt_union)
flow_out(Location, PointsTo)

flow_init: flow_in(loc, {}) <- malloc_call {loc}

flow_step: flow_in(dst, pts) <-
  succ {src, dst, is_call: false, is_ret: false} & flow_out(src, pts)

flow_xfer: flow_out(loc, pts2) <-
  flow_in(loc, pts) & updates(loc, us) & kill(loc, ks) & live {loc,
     live}
  & flow::xfer(pts, us, ks, pts2)
```

Listing 6.2: Process Update

```
match update {
  *a = &b =>
    for a_target in pts[a] {
      pts_out.add(a_target, b)
    }
   a = &b =>
     pts_out.replace(a, {b})
   a = b =>
     pts_out.replace(a, pts[b])
   a = *b =>
     pts_out.replace(pts[pts[b]])
  *a = b =>
    for a_target in pts[a] {
      pts_out.add(a_target, pts[b])
    }
  *a = *b =>
    for a_target in pts[a] {
      pts_out.add(a_target,
                  pts[pts[b]])
    }
}
```

last caveat is necessary so that if a function writes through one of its arguments, then never uses it again, the parent will still see the update even though the callee no longer knows how to reference the region. This allows us to *forget* about allocations which are no longer accessible.

Dataflow Listing 6.1 shows the overall flow rules we use. Andersen-style analysis formulates updates as inclusion constraints, rather than equality constraints like Steensgaard. In the initial declaration of predicates, we declare `flow_in` to aggregate via set union on each points-to set. Since each `flow_in` value at a location produces a unique `flow_out` value, aggregation is not necessary on that predicate. We create an empty input map at every allocation site, because allocation is the only action which will add information to an empty points-to relation. `flow_step` propagates points-to relations along normal succession edges. Finally, `flow_xfer` performs the meat of the operation, absorbing the update summaries into the current context by applying the transfer function.

6.1.5 Inter-procedural Dataflow

Our approach to inter-procedural dataflow follows the example of Reps [63] with some modifications. Some optimizations described in that work are inapplicable to our domain due to source-level assumptions. However, their general structure for procedural dataflow still applies. At the site of a call, we add an inter-procedural edge from the call site to the target function. Following this edge elides stack slots which are not currently pointed to. This is a departure from Reps in that their restricted domain, all local variables were not passed through. We add an intra-procedural edge which skips over the call, applying effects (see §6.1.9) and clobbering variables not preserved by the function. Finally, at each return site, we add an inter-procedural edge which forgets the stack slots local to that function if not pointed to. We do not report an error even if a local stack slot is reachable in the parent to, as this is a legal possibility in the case of a recursive call which passes one of its stack variables by reference. Again, we depart from Reps here by keeping around what would normally be local variables if they are reachable through dereferences.

`flow_call` and `flow_ret` apply at function entry and exit to cut down on unnecessary information propagation, applying the restrictions for our inter-procedural edges, as described earlier.

Listing 6.3: Inter-procedural Rules

```
flow_call: flow_in(dst, pts2) <-
  succ {src, dst, is_call: true} & flow_out(src, pts)
  & flow::call(pts, dst, pts2)
flow_ret: flow_in(dst, pts2) <-
  succ {src, dst, is_ret: true} & flow_out(src, pts) & func {base, contains: dst}
  & flow::ret(pts, base, pts2)
flow_call_over: flow_in(dst, pts2) <-
  call_over {src, func, dst} & flow_out(src, pts)
  & flow::over(pts, pts2)
```

Listing 6.4: Example (C)

```
char* g() {
        return malloc(1);
}
void f() {
        char* x = malloc(1);
        *x = 'a'; // Safe
        char* g_a = g();
        *g_a = 'a'; // Safe
        char* g_b = g();
        *g_b = 'a'; // Safe
        free(x);
        free(g_a);
        *x = 'b'; // UaF
        *g_b = 'b'; // Safe, but needs context
}
```

`flow_call_over` propagates information at a call site *over* a function call, skipping it but removing definitions for variables known to be overwritten by the function. This corresponds to the special intra-procedural edge added to our dataflow when processing a call. This rule also enables analysis through functions which were not provided, albeit with the assumption that the function was effectively a no-op.

6.1.6 Inter-procedural Flow-Sensitive Example

In this section, we provide an example of the context- and flow- sensitive analysis on a simple example shown in Listing 6.4. To provide clarity on how updating alias sets work, this example does not show data-flow merges, which is a set union operation as previously described.

Listing 6.4 contains one real use-after-free bug. We also show one location that is safe, but without context-sensitivity the analysis will raise an additional false positive. Without context-sensitivity, both the allocations for `g_a` and `g_b` get merged, making the analysis believe `g_a` aliases `g_b` when it does not.

Listing 6.5: Annotated Flow-Insensitive Analysis

```
1   g:
2     subq    $8, %rsp
3     movl    $1, %edi
4      { RAX -> {dyn@5, dyn@13}; sp+24@f -> dyn@1 }
5     call    malloc  ; &dyn@5
6      { RAX -> dyn@5; sp+24@f -> dyn@13 }
7     addq    $8, %rsp
8     ret
9
10  f:
11    subq    $40, %rsp
12    movl    $1, %edi
13    call    malloc  ; &dyn@13
14     { RAX -> dyn@13 }
15    movq    %rax, 24(%rsp)
16     { RAX -> dyn@13; sp+24@f -> dyn@1}
17    movb    $97, (%rax)  ; Safe
18    call    g
19     { RAX -> dyn@5; sp+24@f -> dyn@13}
20    movq    %rax, 16(%rsp)
21     { RAX -> dyn@5; sp+24@f -> dyn@13;
22         sp+16@f -> dyn@5}
23    movb    $97, (%rax)  ; Safe
24    call    g
```

```
25      { RAX -> dyn@5; sp+24@f -> dyn@13;
26         sp+16@f -> dyn@5}
27     movq      %rax, 8(%rsp)
28      { RAX -> dyn@5; sp+24@f -> dyn@13;
29         sp+16@f -> dyn@5; sp+8@f -> dyn@0}
30     movb      $97, (%rax) ; Safe
31     movq      24(%rsp), %rdi
32      { RAX -> dyn@5; sp+24@f -> dyn@13;
33         sp+16@f -> dyn@5; sp+8@f -> dyn@0;
34         RDI -> dyn@13 }
35     call      free ;; frees memory pointed to by RDI
36      { RAX -> dyn@5; sp+24@f -> dyn@13;
37         sp+16@f -> dyn@5; sp+8@f -> dyn@0;
38         RDI -> dyn@13; dyn@1 -> free@35 }
39     movq      16(%rsp), %rdi
40      { RAX -> dyn@5; sp+24@f -> dyn@13;
41         sp+16@f -> dyn@5; sp+8@f -> dyn@0;
42         RDI -> dyn@5; dyn@13 -> free@35 }
43     call      free
44      { RAX -> dyn@5; sp+24@f -> dyn@13;
45         sp+16@f -> dyn@5; sp+8@f -> dyn@0;
46         RDI -> dyn@5; dyn@13 -> free@35;
47         dyn@5 -> free@43 }
48     movq      24(%rsp), %rax
49      { RAX -> dyn@13; sp+24@f -> dyn@1;
50         sp+16@f -> dyn@5; sp+8@f -> dyn@0;
51         RDI -> dyn@5; dyn@13 -> free@35;
52         dyn@5 -> free@43}
53     movb      $98, (%rax) ; UaF
54     movq      8(%rsp), %rax
55      { RAX -> dyn@5; sp+24@f -> dyn@13;
56          sp+16@f -> dyn@5; sp+8@f -> dyn@0;
57          RDI -> dyn@5; dyn@13 -> free@35;
58          dyn@5 -> free@43 }
59    movb      $98, (%rax) ; Safe, but needs context
60    addq      $40, %rsp
61    ret
```

We show the corresponding assembly, annotated with comments on the location of the UaF and alias sets, in Listing 6.5. Portions of the alias set highlighted in green are new bindings. Portions in blue are bindings which have been destructively updated. In the example, the variables are:

- The stack slots of f (g has none): sp+24@f, sp+16@f, sp+8@f
- All used registers (other than the stack): RDI, RAX
- Allocations split on sensitivity, using the form dyn@addr{stack}
  - Context-insensitive: dyn@5 and dyn@13
  - Context-sensitive: dyn@5{19}, dyn@5{25}, dyn@13{}

Comments show the alias sets after each step. For example, right before the free on line 35, the alias set is:

`{ RAX -> dyn@0; sp+24@f -> dyn@1; sp+16@f -> dyn@0; sp+8@f -> dyn@0; RDI -> dyn@0 }`

This shows that `RDI` is pointing to the memory location `dyn@1`, the allocation site for `x` in the source code. Right after the `free` on line 35, the alias set changes to show `dyn@1` is now free. The UAF detector would say any pointer that resolves to `dyn@1` is therefore a use-after-free bug, which happens on line 53.

If we had been context-sensitive, the points-to relation at 59, the false positive, would instead be

`{ RAX -> dyn@5{25}; sp+24@f -> dyn@13{}; sp+16@f -> dyn@5{19}; sp+8@f -> dyn@5{25}; RDI -> dyn@5{19}; dyn@13{} -> free@35; dyn@5{19} -> free@43 }`

The key difference here is that `RAX` points to `dyn@5{25}` rather than just `dyn@5`, so we can distinguish it from the freed allocation. This allows context sensitivity to weed out more false positives.

6.1.7 Adding Context Sensitivity

Context-sensitivity requires we only change our `Location` values to contain information about the stack. We add an empty stack to entry points to initialize the new stack-enhanced CFG. Called functions must copy their control flow graphs to separate versions for every stack the program calls them at. When generating the target of a call function, we now add the current instruction's fall-through to the stack as a return address. If the return address is already on the stack, we truncate the stack so that it is the topmost.

Unfortunately, with an unbounded stack, of our real-world samples (§6.3.2), this approach exhausts available resources (128G RAM) for all but `gnome-nettool`. To remedy this, we use a k-stack approach combined with the stack truncation above. If a call's location has a stack, the stack is first checked to see if the call site is present. If so, we truncate the stack as before. Otherwise, we push the call site onto the stack, and if the stack is longer than k, we remove the oldest entry. In the case of our example, a stack size of 1 would be sufficient, and we'd see the allocations `dyn@0{10}`, `dyn@0{11}`, and `dyn@1{}`.

6.1.8 Adding Field Sensitivity

Up until now, we've been treating each memory region like a single homogeneous bag: Anything written into it ever is a possible result on any subsequent read. This reduces precision.

In traditional alias analysis, simply treating each variable with a struct type as though it were multiple variables, one for each field, can implement field sensitivity. Unfortunately, in the binary case, things are slightly messier for two reasons: overlapping fields, and variable offset writes. Since this is only a pointer analysis, not a general value analysis, we only consider fields with size equal to the pointer size.

Previously, we used a simple set to denote what a variable might point to. We now replace this with a "field map". We will use the word "reference" to describe the pair of a variable and an offset. The offset may be a fixed value, or a special value indicating a computed offset.

A field map has two components: a set of references for unknown offsets, and a set of references for some subset of possible fixed offsets. If we see a fixed offset read, we return the offset's set if defined, otherwise the unknown set. If we instead see a computed offset read, we return the union of every set (both unknown and fixed) present in the map.

We split update rules into cases for a single pointer write (so we know which memory cell the write updated precisely) and for multiple possibilities.

Single Pointer. For a fixed offset write on a variable, we destructively update the set corresponding to that offset. In case of a computed offset write, we extend both the unknown set, and every tracked offset to contain the new reference. If a write targets an overlapping field, we empty the field it overlaps.

Multiple Pointers. For a fixed offset write to variable, extend the set corresponding to that offset. If we find a computed offset write, we extend both the unknown set, and every tracked offset to contain the new reference.

This structure is conservative so long as we accept the assumption that pointers are not constructed piece-wise. Specifically, multiple writes construct a single pointer, we will not know about it. This limitation was present in the previous, field insensitive formulation, but becomes more obvious in the description of the field sensitive extension. This practice is extremely uncommon (outside bulk copy functions like `memcpy`), so we accept this limitation to limit the need to reason about the exact possible values of a computed offset.

Unfortunately, this formulation gives the alias analyzer slightly too much power. Specifically, it can now count. Despite the fact that we did not model actually performing arithmetic, a incrementing a pointer in a loop will look like repeatedly examining relative offsets of a struct. To deal with this, we add a widening operator[2] which, if the same variable points to a fixed limit or more offsets in a region, will replace it with pointing to an unknown offset to that region. This reclaims termination.

6.1.9 Recent Allocation Domain

Using the site or site and size of an allocation to define the allocation domain is common. However, this can have poor behavior around loops:

```
char* x;
while(1) {
site_0:
        x = malloc(1);
        *x = 'a';
        free(x);
}
```

If running a dataflow computation to fixpoint, on the second iteration, the allocation for `site_0` will appear already freed. The usual response to this type of imprecision is to unroll loops a fixed number of times. However, we intend our tool to be complete (assuming a complete control flow graph, provided functions, etc.) so we want to avoid fixed unrolling.

To this end, we extend our allocation domain with a "recent" bit, similar to the MRAB vs NMRAB abstraction [11], though for different purposes. Relative to a concrete trace, the address most recently given by an allocation at a given site belongs to the set where "recent" is true. All other addresses issued by that site belong to the set where "recent" is false. In our static form, a value belongs to the recent set for an allocation site if there exists some trace for which it was the last allocated from that site. It belongs to the non-recent set if there exists some trace for which it was not most recently allocated. Note that once

---

[2] A widening operator is an addition to a dataflow analysis which provides termination by moving further on the lattice than is strictly necessary under certain conditions.

in static form, the recent set may have more than one member, and may even intersect with the non-recent set.

This extension is a departure from the normal Andersen alias analysis. It is only for precision, not correctness. We can consider this as 1 bit path-sensitivity, where path-sensitivity would be parameterizing on entire sequences of instructions which could reach the current point.

To implement this in our static alias analysis, we construct summaries for each function of allocations and possible allocations. When processing the edge which skips the function, we update the points-to relation using information from this summary to keep recency information accurate. We do this by modifying the behavior of `flow_call_over` to update the caller points-to for the purpose of representing what may have happened in the callee.

We define an "effect" to be a set of definite allocation sites and a set of possible allocation sites that may occur when calling the function. To apply an effect to a points-to set: For every definite allocation site, make the recent bit false. For every possible allocation site, duplicate any recent references to have both recent and non-recent values. Listing 6.6 shows this change.

There are four new functions here. `effect_merge` will merge two local effects at a confluence point. It does this by intersecting their definite allocations, and migrating all other allocations to possible allocations. `flow::over_effect` will apply the effect to the points-to set in addition to its earlier responsibilities. `effect::sequence_effect` will update the currently processed effect with the called function's effect by unioning together both possible and definite effects, then removing those possible effects which are also definite. `effect::malloc` just adds the current site as a definite allocation to the effect.

The result is that these effect summaries allow greater precision over allocation sites, similar to the benefits of loop unrolling, but without sacrificing fix-point semantics.

In our example earlier, this would give rise to three allocations - `dyn@0`, `dyn@1+old`, and `dyn@1`. This would suppress the false positive by distinguishing between the two invocations of `g`. However, were a third added, it would not be able to distinguish between the first two

Listing 6.6: Rules for Recent Domain

```
flow_call_over: flow_in(dst, pts2) <-
  call_over {src, func, dst} &
  flow_out(src, pts) &
  func_effect {func, effect} &
  flow::over_effect(pts, effect, pts2)

local_effect {base: Location, local: Location, effect: Effect}
func_effect {func: Location, effect: Effect^effect_merge}

effect_init: local_effect {base, local: base, effect: no_op} <- func {base}
effect_ret: local_ret: func_effect(base, effect) <-
  succ {src: local, is_ret: true} &
  local_effect {base, local, effect}
effect_xfer: local_effect {base, local: local2, effect} <-
  succ {src: local, dst: local2, is_call: false, is_ret: false} &
  local_effect {base, local, effect}
effect_call: local_effect {base, local: local2, effect: effect2} <-
  call_over {src: local, func: remote, dst: local2} &
  func_effect {func: remote, effect: effect_call} &
  local_effect {base, local, effect} &
  effect::sequence_effect(effect, effect_call, effect2)
effect_malloc: local_effect {base, local: local2, effect: effect2} <-
  succ_over {src: local, dst: local2} &
  local_effect {base, local, effect} &
  malloc_call {loc: local} &
  effect::malloc(effect, local, effect2)
```

invocations of `g`.

6.1.10 Use-after-Free

With alias relationship in hand, we must determine which reads and writes in the program are use-after-free candidates. For the flow and flow & context analyses, we can augment the alias analysis itself to track most of this information for us. When a free occurs, we generate a special form of the `*a = &b` summary, with `a` as the argument to free, and `b` as a special value representing things freed at that location. This is as presented in Listing 6.7. `read_vars` generates pointer access information from actual assembly instructions, and `use_vars` imports it from summaries which can do things like determine argument count to `printf`.

Listing 6.7: Tracking Frees with Flow Sensitivity

```
read_vars: deref_var(v, loc) <-
  lift {loc, bil} &
  func {base, contains: loc} &
  uaf::reads_vars(bil, v)
use_vars: deref_var(v, loc) <-
  uses(r, loc) &
  uaf::use_vars(r, v)

free_summary: summary(loc, s) <-
  free_call{loc, args} &
  uaf::free_summary(args, s)
uaf_flow: uaf_flow(v, loc, loc2) <-
  deref_var(v, loc2) &
  flow_in(loc2, pts) &
  flow::is_freed(pts, v, loc)
```

However, this approach only works with alias information that is at least flow sensitive. Without flow sensitivity, any pointer which was ever freed will appear freed everywhere in the program, even before that free occurred. As a result, the false positive rate would be absurd, and such a detector would be more a heap-access detector than a use after free detector. To help it along and make the difference in false positive rates more about the precision of the alias information rather than the ability to track state changes, we add a extra conditions to report a use after free without flow sensitivity by essentially doing flow sensitive tracking of the freed-property only.

Listing 6.8: Tracking Frees Insensitively

```
base_freed: freed_base(v, loc) <-
  free_call {loc, args} &
  uaf::free_args(args, v)
all_freed: freed_var(v2, loc) <-
  freed_base(v, loc) &
  steens_point(v, vs) &
  uaf::expand_vars(vs, v2)

path_init: path_exists(loc, loc) <-
  freed_base(_, loc)
path_step: path_exists(loc, loc3) <-
  path_exists(loc, loc2) &
  succ_any {src: loc2, dst: loc3}

uaf: uaf(v, loc, loc2) <- freed_var(v, loc) & path_exists(loc, loc2) & deref_var(v, loc2)
```

Listing 6.8 shows the implementation of this modification. In the first stanza, we mark the variable freed by the free call and everything it aliases with as potentially freed at that location. In the next, we find those parts of the program reachable from the free site. We then finalize the results by saying that if a path exists from a freed variable to a dereference of that same variable, then there is a use after free candidate. Going much further to disambiguate variables would begin to graft flow sensitive information into the otherwise insensitive analysis.

6.2 Implementation

Our implementation consists of 3k lines of Rust and 234 lines of Datalog. We implement each additional sensitivity (field, context, flow, recency) by adding an additional Datalog file. The switches to determine which sensitivities actually run operate by adding initial facts to the database to suppress unwanted rule triggers. As a result of this design, other than field sensitivity (which requires significant logic in both the constraint generator and points-to set structures), we can safely cut out these sensitivities by simply removing the rules in question (and their associated bound functions).

6.2.1 Limitations

Our system is conservative in almost all cases, but there are a few notable exceptions. If the provided file links against files which are not provided, the analysis will treat those functions as if they are no-ops. If in reality, these functions free memory or write to the heap, this may cause missed vulnerabilities. The system uses the BAP control flow graph and any indirect jump resolution and the analysis assumes they go nowhere; we do no additional resolution beyond `ret` calls. Control flow recovery is a general problem in binary analysis and not specific to our approach. Essentially, if a prerequisite step gives our algorithm incomplete information, it will produce an incomplete result. This was sufficient for the programs analyzed, which were primarily written in C, but some function pointer resolution would help to achieve good quality result on C++ programs which use vtables or programs which use a callback architecture (and so rely heavily on function pointers).

We assume the transfer of a pointer from one place to another will take place in a single assembly instruction - we do not model “the first 3 bytes of a pointer to x” for example. Finally, we assume that the program follows traditional stack discipline for purposes of points-to minimization in the sensitive cases. If it is not, aliases from stack calculations pointing above or below the current stack frame may be incomplete.

6.3 Evaluation

Our evaluation has 3 major components.

- Juliet - How do we perform on a labeled (true positives, false positives, false negatives) data set?

- Real World Bugs - Can we detect real bugs?
- Ubuntu `$PATH` - How often do we alarm on real bug-free code?

We evaluate over the Juliet test set both for comparability with other work [34, 80] and to act as a baseline for our detection power and false positive rates. It helps answer the question "In the absences of confounding factors from the real world, how well does this work?". We also evaluate over real use-after-free bugs pulled from MITRE's CVE database. Evaluating on these verifies that real world code, while potentially confounding, does not stop our technique from functioning altogether. It also provides a measurement of the false positive rate in the presence of true positives. Finally, we evaluate over a variety of believed-good binaries. The intent behind this evaluation is to get a better idea of false positive rates and analysis costs for average programs believed to be non-buggy.

6.3.1 Juliet

IARPA released the Juliet test suite [2] as a way of providing standardized examples of CWEs. By building only those corresponding to use-after-free, we get a high density test suite.

All three sensitivities find all intended bugs in Juliet. Insensitive analysis generates 39834 false positives, reducing to 0 with flow sensitivity. Run time was 19m30s for the flow sensitive version, and 30m4s for insensitive. However, the insensitive variety generates its alias information as of 3m23s. The system spent the remainder of the time generating reachability information.

Unfortunately, while Juliet serves as a good test for true positives, it does not do much to elicit false positives from our system, which is why both our performance and others' look too-good-to-be-true here. Within the negative tests, there is little in the way of things that checkers are traditionally weak against (data structures, recursion, loops, etc.). For this reason, it is important to evaluate ourselves on real world code as well.

Program	Sensitivity	Run time	Memory	False Positives	Binary Size
gnome-nettool	Insensitive	38s	1G	1851	156k
	Flow	30s	1G	0	
	Flow + Ctx	2m34s	2G	0	
goaccess	Insensitive	4m35s	15G	387459	635k
	Flow	16m14s	10G	112420	
	Flow + Ctx	43m34s	34G	87	
libarchive	Insensitive	1m23s	3G	4917	366k
	Flow	34s	1G	852	
	Flow + Ctx	22m12s	44G	7	
shadowsocks-libev	Insensitive	2m12s	5G	130760	631k
	Flow	3hr46m21s	62G	22357	
	Flow + Ctx	3hr53m26s	72G	115	
mdadm	Insensitive	16m45s	31G	1056570	768k
	Flow	2hr24m13s	42G	270683	
	Flow + Ctx	12hr10m43s	111G	14566	
isisd	Insensitive	3m46s	8G	58241	451k
	Flow	18m49s	9G	11776	
	Flow + Ctx	22m32s	25G	513	

Table 6.1: Real CVE Performance

Sensitivity	Run time			Memory			Alarms	
	Avg	Median	Stdev	Avg	Median	Stdev	Avg	Imp
Insensitive	2m26.1s	58.4s	3m38.1s	241.7M	34.1M	1.9G	73.1	
Flow	2m14.2s	54.7s	3m19.6s	236.8M	34.1M	1.9G	0.5	93.1%
Flow and Ctx	2m22.1s	55.6s	3m54.9s	349.2M	34.0M	2.3G	0.2	43.5%

Table 6.2: Ubuntu `/usr/bin` Performance

6.3.2 Live CVEs

Our system successfully detects 7 real bugs across 6 programs. All sensitivities of the checker detected all bugs. We assume that all potential use after frees which do not match the known bugs in each of these programs are false positives.

Note that while the insensitive analysis completes quickly and cheaply for every binary, the false positive rates are so high that the output would be difficult to use. Flow sensitivity reduces false positives significantly. Manual analysis reveals that most remaining false positives are either due to data structure usage (which decreases the precision of the alias analysis), confused allocation sites from wrapped malloc constructors, and infeasible paths. Context sensitivity gives additional improvements by helping to differentiate between instances of calling wrapped mallocs (e.g. `new_foo()` to allocate and initialize a `foo`).

Performance for insensitive and flow-sensitive analyses appears similar in large part because the generation of a global program reachability graph for each `free` is costly. If the analysis is instead timed in phases, the alias-analysis-only portion for the insensitive system takes seconds, while it takes the bulk of the non-CFG-recovery time in a flow-sensitive analysis.

For the known-vulnerable set, flow sensitivity reduced the false positive set by an average of 90%, and context sensitivity reduced it by an additional 84.1%. The false positive reduction for the addition of flow sensitivity is immense, and the increase in time and space needed for the alias analysis was manageable for programs in our known-vulnerable set, the largest of which was 768kb. Adding context sensitivity further increased the time and space cost, but still yielded a major increase in precision.

GUEB

The author of the GUEB [28] tool made his tool open source[3], allowing us to compare against it. We connected IDA, BinNavi, and GUEB and ran the system over the same bugs we evaluated against. As a caveat, we could not feed them the same binaries our tool consumed - their tool's stack only accepts 32-bit, so we recompiled the same vulnerable programs in 32-bit mode.

Table 6.3 shows the performance. The crashes derive from unhandled cases in the input, and not fundamental to their methods. The undetected bugs occur due to their choice to not follow back edges (either as recursion or loops) when computing their VSA. This is an understandable choice, since VSA can become slow and be difficult to force convergence for when cycles are present in the input, but in this case it caused their analysis to miss bugs. Likely due to this forwards-only approach, GUEB terminated rather quickly on all inputs.

---

[3] `https://github.com/montyly/gueb`

Program	False Positives	Bug Found?
gnome-nettool	2	Yes
goaccess	crash	crash
libarchive	222	Yes
shadowsocks-libev	crash	crash
mdadm	crash	crash
isisd	596	No

Table 6.3: GUEB Performance

Listing 6.9: `isisd` Vulnerability

```
// ... (adj is allocated and constructed here)
for (level = IS_LEVEL_1; level <= IS_LEVEL_2; level++) {
        // ...
        else if (new_state == ISIS_ADJ_DOWN) {
                // ...
                isis_delete_adj(adj);
        }
}
// ...
```

In Listing 6.9, we can see one of the real vulnerabilities the lack of a fixpoint fails to detect. The loop knows that `adj` is allocated and non-null on entry, so the first time through the loop is always fine. However, some paths through the loop free `adj`, and go around the loop again. At this point, a use-after-free can occur. If back edges are not followed, the analysis cannot detect this.

6.3.3 Ubuntu Path Sample

Now that we know that our program will alert us to real world vulnerabilities, we also want to know how it will behave in the case where no expected vulnerabilities are present. To this end, we ran our program across `/usr/bin` on a default Ubuntu Xenial installation, as shown in Table 6.2.

Adding flow sensitivity provided an average reduction in bug candidates of 93.1% in those situations where the insensitive code found at least one candidate. Then adding context sensitivity ($k = 1$) reduced it by an additional 43.5%, in those situations where the flow sensitive analysis had a bug candidate, and the context sensitive analysis terminated.

Manual auditing of the reported bugs did not reveal any true bugs, but did show that a common pattern amongst false positives was functions for whom one path freed and replaced a pointer, and the other did neither, and they rejoined. A more aggressive analysis for dead variables could remedy this by pruning them to allow the freed region to leave the points-to relationship before the paths rejoined.

The system emitted a maximum of 22 reports on individual binaries (and this worst case had most of them clustered in the same code area). This was few enough reports to enable practical manual auditing by a single individual. Unfortunately, none of these reports corresponded to real bugs upon examination. This does not guarantee these programs are bug free - while we have a conservative analysis, that is dependent on seeing the entire control flow graph. In this case this condition is not met. Some C++ programs which use vtables are present in this path - calls to member functions there will appear as no-ops. Function pointers are similarly considered to be no-ops. Calls into libraries which were not analyzed with the binary are similarly absent. Finally, some of these are GUI or threaded applications, which utilize a callback system we again do not handle control flow edges for.

6.4 Related Work

6.4.1 Dynamic Approaches

A use-after-free bug specifically describes a temporal safety property: In a specific execution of the program, it releases a region of memory to the allocator, and uses that region afterwards. As a result, checking this property on an actual execution of the program via dynamic analysis is appealing. Assuming we have a single execution path in mind, checking for a use-after-free bug becomes a matter of remembering which addresses have are free, and watching for additional dereferences of these. [4] This approach avoids false positives

---

[4] Technically, to avoid false positives the allocator must not hand out the same region twice, or must employ some other mitigation strategy.

by using real runs, and frequently only incurs constant-factor overhead on runs, making it widely applicable. However, it also has the limitation that bugs along paths not included in a trace will remain unnoticed.

LLVM's ASAN [1] and Valgrind [3] approach this through instrumented execution while monitoring the safety property. ASAN accomplishes this via compile-time instrumentation, and Valgrind through a hooked virtual machine. Undangle [18] monitors derived pointers whose allocation the program has freed (dangling pointers) using taint analysis via TEMU (a virtual machine hooking framework on top of QEMU). DangNull [48] uses LLVM to do compile time instrumentation to track dangling pointers, writing `NULL` to them on free. This converts otherwise exploitable conditions into either a DoS (if there was no null handling) or a recoverable error. DangSan [74] follows on the work of DangNull, again doing compile time instrumentation over LLVM bitcode. DangSan distinguishes itself in better multi-threaded performance, focusing on only on detection rather than recovery.

The major limitation in dynamic analysis is code coverage. Comprehensive test suites can achieve coverage manually, but these are difficult to write and can often miss the faulty cases - if the author were thinking about that case when they wrote the code, they likely would not have written the bug. Fuzzing can allow for automated generation of interesting execution traces for the purpose of use by dynamic analysis, but relies on statistical techniques or manual guidance to achieve reasonable coverage. This approach guarantees that found bugs are real, but ties their power to whatever the mechanism used to generate paths.

6.4.2 Static Approaches

Alternatively, we can search for bugs by tracking properties that will hold across all traces, and finding uses we don't know aren't freed. This approach will not miss any bugs, and does not depend on any kind of test suites or input generators for its results. Unfortunately, this approach also brings false positives and a greater potential for scaling issues. As a result, while the core of static techniques could give the completeness property of detecting all bugs, in a final user version may drop this guarantee in order to reduce false positive rates.

METAL proposed [27] the use of simple, programmer written state machines to enforce additional properties across C code. This approach works for things like interrupts, locking, no-alloc zones, and no-side effect zones, and is low overhead. It even works for simple use-after-free bugs where no alias analysis is necessary (e.g. `free(x); *x`), and found real instances of this.

TAC [80] applies an insensitive pointer analysis to identify candidate use after frees, then runs type-state, path-sensitive analysis on the results. In the type-state analysis, when they encounter what could be a use-after-free or a double-free, rather than immediately transitioning to the error state, they query their learned model. If the bug is not likely a real bug by their model, they continue executing past it as though it had no effect. The support vector machine step, while it does eliminate a good chunk of false positives, does fundamentally move Tac from one-sided error to two-sided. If the SVM removes a bug candidate, Tac will not report it. At the same time, in the type-state phase of the algorithm, they proceed along a slice from the allocation site, meaning they fundamentally cannot know that a condition is feasible if it depends on prior code, they can only detect infeasibility. Regardless of the two-sided error, it has good true and false positive rates, so the tool is practically applicable. It is possible that if Tac used our approach to generate the bug candidates, the SVM phase might not be as necessary due to the increased precision of both selected use/free pairs, and the points-to sets provided.

For compiled code, the GUEB [28] is the existing static checker. GUEB employs VSA [11] to track values, augmented to track allocations and frees. When it finds a situation where a dereference to a chunk which is possibly freed, it extracts a subgraph of the control flow graph trying to display only the portion with the allocation, free, and use of the relevant pointer. In order to allow VSA to run quickly, it use fixed loop unrolling and disallows recursive calls. This is pragmatic to allow VSA to converge, but causes GUEB to miss some real world bugs.

One other tool [23] has the distinction of operating on object-oriented code, even in the presence of vtables. Their AODA tool has the benefit of reasoning about C++ code, but the limitation of not reasoning about pointers which have the program copies onto the heap. When they reason about a use of an object, they use the use-def chain to identify the instantiation site. They also use the use-def chain to determine what to add to the kill-set when the program destroys an object.. If a pointer to an object moves onto the heap (e.g. `vector<*SomeObject>`), then when moved off, the use-def chain will no longer trace it back to its origin - this is one of the problems alias analysis solves. They also only examine the output of MSVC, meaning they are possibly tied to some compiler idioms. However, their ability to resolve virtual function calls on C++ objects is potentially valuable.

6.4.3 Datalog Program Analysis

Datalog has played a role in program analysis before [14, 78], though only on source code or a compiler IR like LLVM bitcode or Java bytecode. These approaches transform the program into a set of input facts to combine with rules and run by a Datalog engine to receive the final results, essentially phasing the computation. Our approach uses a mixture of Datalog with traditional procedural code instead. This allows us to do things like lifting newly found instructions from within the Datalog context or hypothetical reasoning which requires a computation step. In our implementation, this occurs for every instruction other than the entry points, since we discover the location of the next instruction during lifting. However, this also means that we lose Datalog's termination guarantees and cannot use most common Datalog engines.

`bddbddb` [78] used BDDs (binary decision diagrams) to exploit symmetry in context-sensitive points-to analysis for Java programs. In order to encode this problem as a BDD, `bddbddb` first encodes the points-to problem as a Datalog program. Then, they converted each operation needed to run a Datalog program (join, substitution, extension) into operations on BDDs, one per predicate. Finally, by running these operations unto fixpoint, the user can query the resultant BDD efficiently for points-to information which if represented concretely would be far too large to manipulate.

Perhaps the closest to MARDUK is DOOP [14, 15, 69], a program analysis framework for Java bytecode based in Datalog. Like `bddbddb`, DOOP extracts from the program and analyses to a pure Datalog program, then runs to completion. Unlike previous work, it relies less on preprocessing, and performs more of the analysis within the Datalog program itself. They showed [14] that information needs to flow bidirectionally between pointer and exception analyses, a condition which Datalog is great at, since it does not require manual control flow interleaving. DOOP distinguishes itself on modeling accuracy as a result of this analysis combination approach [15], outperforming `bddbddb` even with the same sensitivity simply by discovering more of the control graph. Finally, they found that careful modification of the algorithm can recover most of the compactness wins normally acquired from a BDD representation [69].

DOOP uses LogicBlox [7] as their Datalog backing engine. LogicBlox uses Leapfrog Trie-Join [77] a novel join algorithm whose primary property is its ability to handle multi-way joins of the sort found in Datalog queries efficiently. They combine this with incremental maintenance [76] to provide a framework to compute their Datalog dialect, LogiQL. LogicBlox is disk-backed, meaning they might be able to avoid some of the memory issues our implementation hit for high sensitivity outputs.

Chapter 7

Conclusion

We have presented Holmes, a new dialect of Datalog for orchestrating binary analyses. We described the language informally (§ 4) and formally (§ 5). We demonstrated the concrete use of Holmes to detect real-world bugs as a use-after-free detector (§ 6). We showed that some analyses, such as type recovery, written without Holmes in mind match its execution paradigm regardless (§ 3). We conclude that the Holmes approach improves upon the current ad-hoc model by representing the interplay between analyses.

We described a new EDSL based on a dialect of Datalog (§ 4). We designed Holmes as an orchestrater of procedural analyses, rather than an individual analysis itself. We show a novel form of negation via exact aggregation, a useful tool in the domain of binary analysis (§ 4.2.4). We extend that negation to make progress from failed hypotheses (§ 4.2.5) as is the case when a control flow graph is incrementally recovered through use of analyses which assume a complete graph. We describe and evaluate implementation approaches for this language (§ 4.3).

To enable reasoning about programs written in Holmes, we give a formal semantics (§ 5). We present the Holmes' unique extended negation notion in the context of a finite Datalog world (§ 5.1), showing how it connects with stratified, stable set, and well-founded semantics. We show how to rewrite a Holmes program into a Datalog program in an infinite Herbrand universe (§ 5.3). Finally, we show how to augment the negation concept from finite Datalog to one which may perform an infinite number of negations in a single step to enable circumscription (§ 5.4).

We showed that Holmes can function as a framework for real world analyses by creating a use-after-free detector built on it which leverages Holmes to explore different sensitivities of alias analysis (§ 6). We translate alias analyses from the traditional programming language world for use in the binary analysis world (§ 6.1). We quantify the cost/benefit tradeoff for precision in alias analysis for bug finding (§ 6.3). We show that Holmes assists in the ability to write such a system concisely and to allow modularity between analyses (§ 6.2).

We show an approach to binary type recovery, BiTR, which performs better than previous work and would be well suited to Holmes integration (§ 3). We define a descriptive type system for recovering compiled C code, allowing a more flexible type recovery than is possible either using only C types or a prescriptive system (§ 3.2). We show how to generate constraints in this system and solve them in a way that tolerates partially typed programs (§ 3.3). We provide a novel measurement system for type recovery evaluation based on its probability of correctness, showing how the previously used distance mechanism falls short (§ 3.4.1).

7.1 Availability

Much of the work described in this thesis is available for download and use under a BSD license.

- A legacy implementation of Holmes (feature incomplete) based on PostgreSQL is available at `https://github.com/maurer/holmes`
- An in-memory implementation of Holmes, as used for Marduk, is available at `https://github.com/maurer/mycroft`
- Marduk, a static binary use-after-free detector is available at `https://github.com/maurer/marduk`

Bibliography

[1] AddressSanitizer — Clang 7 documentation. https://clang.llvm.org/docs/AddressSanitizer.html. 6.4.1

[2] Juliet Test Suite (1.3). NIST, NSA. 6.3.1

[3] Valgrind. 6.4.1

[4] María Alpuente, Marco Antonio Feliú, Christophe Joubert, and Alicia Villanueva. Datalog-based program analysis with BES and RWL. In *Datalog Reloaded*, pages 1–20. Springer, 2011. 1

[5] Lars Andersen. *Program Analysis and Specialization for the c Programming Language*. PhD thesis, University of Cophenhagen. 6, 6.1.4

[6] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD Thesis, University of Cophenhagen, 1994. 2.1.4, 6

[7] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and Implementation of the LogicBlox System. pages 1371–1382. ACM Press, 2015. 4.3.2, 6.4.3

[8] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *In FOCS (2008*, pages 739–748. 4.3.2

[9] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic Exploit Generation. *Commun. ACM*, 57(2):74–84, February 2014. 2.1.5

[10] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM. 2.1.5

[11] Gogul Balakrishnan. *WYSINWYX: What You See Is Not What You Execute*. PhD, University of Wisconsin-Madison, 2007. 1, 2.1.1, 2.1.2, 2.1.4, 3.3.7, 4.1.2, 6, 2, 6.1.9, 6.4.2

[12] Gogul Balakrishnan and Thomas Reps. Divine: Discovering variables in executables. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 1–28. Springer, 2007. 2.1.4, 3.3.2

[13] Al Bessey, Dawson Engler, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, and Scott McPeak. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, February 2010. 6

[14] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 1–12. ACM, 2009. 1, 4.1.2, 6.4.3

[15] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices*, 44(10):243–262, 2009. 6.4.3

[16] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011. 1, 3.2.3, 3.3, 3.5.1, 4.1.1, 6.1.2

[17] David Brumley and James Newsome. Alias analysis for assembly. Technical report, Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 2006. 1

[18] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143. ACM, 2012. 6.4.1

[19] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. 2.1.5

[20] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, May 2012. 2.1.5

[21] Anthony Cozzie, Frank Stratton, Hui Xue, and ST King. Digging for Data Structures. *OSDI*, pages 255–266, 2008. 3.1, 3.5.1

[22] K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. Talx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, pages 25–35, 1999. 3.5.2

[23] David Dewey, Bradley Reaves, and Patrick Traynor. Uncovering Use-After-Free Conditions in Compiled Code. In *Availability, Reliability and Security (ARES), 2015 10th International Conference On*, pages 90–99. IEEE, 2015. 6.4.2

[24] EN Dolgova and AV Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 35(2):105–119, 2009. 3.5.3

[25] Michael Eddington. Peach. 2.1.5

[26] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 51–60. ACM, 2013. 3.2.1, 3.3.2, 3.4.1, 3.4.3, 3.5.1

[27] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-specific, Programmer-written Compiler Extensions. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association. 6, 6.4.2

[28] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, August 2014. 6, 6.3.2, 6.4.2

[29] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Interval Analysis and Machine Arithmetic: Why Signedness Ignorance Is Bliss. *ACM Transactions on Programming Languages and Systems*, 37(1):1–35, January 2015. 2.1.2

[30] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics For Logic Programming. pages 1070–1080. MIT Press, 1988. 5.1.1

[31] Michael Gelfond and Vladimir Lifschitz. Logic Programming. pages 579–597. MIT Press, Cambridge, MA, USA, 1990. 5.1.1

[32] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM. 2.1.5

[33] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *ACM Sigplan Notices*, volume 45, pages 43–56. ACM, 2010. 2.1.1

[34] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, December 2015. 6.3

[35] Ilfak Guilfanov. IDA Pro: Hex-Rays. 1, 3.1, 3.4.2, 3.5.3

[36] Brian Hackett and Shu-yu Guo. Fast and Precise Hybrid Type Inference for JavaScript. In *In PLDI*, 2012. 2.1.3

[37] Istvan Haller, Asia Slowinska, and Herbert Bos. MemPick: High-level data structure detection in C/C++ binaries. *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 32–41, October 2013. 3.5.1

[38] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium On*, pages 289–298. IEEE, 2011. 2.1.4, 6

[39] Sam Hocevar. Zzuf. 2.1.5

[40] Trevor Jim. What are principal typings and what are they good for? Technical Report November, 1995. 3.3, 3.3.3, 3.5.2

[41] Changhee Jung and Nathan Clark. DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage. *Proceedings of the 42nd Annual IEEE/ACM . . .*, 2009. 3.5.1

[42] George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. An efficient data structure for must-alias analysis. pages 48–58. ACM Press, 2018. 6

[43] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 161–174, New York, NY, USA, 1999. ACM. 3.5.2

[44] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via Geometric Resolutions: Worst-case and Beyond. *arXiv:1404.0703 [cs]*, April 2014. 4.3.2

[45] Johannes Kinder. Static analysis of x86 executables. Technical report, Technische Universität Darmstadt, 2010. 1, 2.1.2

[46] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–12. ACM, 2005. 1

[47] Chris Arthur Lattner. *LLVM: An Infrastructure for Multi-Stage Optimization.* PhD Thesis, University of Illinois at Urbana-Champaign, 2002. 1, 3.5.1

[48] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing Use-after-free with Dangling Pointers Nullification. Internet Society, 2015. 6.4.1

[49] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium*, January 2011. 2.1.1, 2.1.3, 3.1, 3.3.2, 3.4.1, 3.4.3, 3.4.4, 3.5.1, 3.5.3

[50] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. 2010. 3.1, 3.4.1, 3.5.1

[51] Matthew Maurer. {Bi}nary {T}ype {R}ecovery, 2014. 2.1.1, 2.1.3

[52] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, July 2002. 1

[53] John McCarthy. Circumscription—a form of non-monotonic reasoning. In *Readings in Artificial Intelligence*, pages 466–472. Elsevier, 1981. 1, 2.2.3

[54] Bogdan Mihaila. Bindead. 1

[55] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. *Stack-based typed assembly language*. Springer, 1998. 3.5.2

[56] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, pages 208–223, London, UK, UK, 1999. Springer-Verlag. 3.5.3

[57] Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In *Asian Symposium on Programming Languages and Systems*, pages 115–130. Springer, 2012. 2.1.1, 2.1.2

[58] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 37–48. ACM, 2012. 4.3.2

[59] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, and Ian Quah. Self-Driving Database Management Systems. In *CIDR*, 2017. 4.3.1

[60] G. Ramalingam. On loops, dominators, and dominance frontier. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 233–241. ACM Press, 2000. 2.1.1

[61] E Raman and DI August. Recursive data structure profiling. *Proceedings of the 2005 workshop on Memory . . .* , 2005. 3.5.1

[62] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 861–875, Berkeley, CA, USA, 2014. USENIX Association. 2.1.5

[63] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995. 6.1.5

[64] Edward J. Schwartz, J. Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, volume 16, 2013. 3.5.3

[65] Z Shao and AW Appel. Smartest recompilation. *Proceedings of the 20th ACM SIGPLAN-SIGACT . . .*, pages 439–450, 1993. 3.3, 3.3.3, 3.5.2

[66] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, Giovanni Vigna, and UC Santa Barbara. (State of) The Art of War: Offensive Techniques in Binary Analysis. page 20. 2.1.4

[67] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *NDSS*. Citeseer, 2011. 2.1.3

[68] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2011. 3.1, 3.5.1

[69] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog Reloaded*, pages 245–251. Springer, 2011. 6.4.3

[70] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze:

A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, pages 1–25. Springer, 2008. 1

[71] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41. ACM, 1996. 2.1.4, 4.1.2, 6, 3, 6.1.3

[72] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1994. 2.2.2, 4.2.4

[73] Katerina Troshina, Yegor Derevenets, and Alexander Chernov. Reconstruction of composite types for decompilation. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 179–188. IEEE, 2010. 3.5.1

[74] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable Use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 405–419, New York, NY, USA, 2017. ACM. 6.4.1

[75] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. Unfounded Sets and Well-founded Semantics for General Logic Programs. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '88, pages 221–230, New York, NY, USA, 1988. ACM. 5.1.1

[76] Todd L. Veldhuizen. Incremental Maintenance for Leapfrog Triejoin. *arXiv:1303.5313 [cs]*, March 2013. 6.4.3

[77] Todd L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, 2014. 4.3.2, 6.4.3

[78] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer, 2005. 1, 2.1.1, 6.4.3

[79] DH White and G Lüttgen. Identifying dynamic data structures by learning evolving patterns in memory. *Tools and Algorithms for the Construction and . . .*, 2013. 3.5.1

[80] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection. pages 42–54. ACM Press, 2017. 6.3, 6.4.2