Blaze: A Framework for Interprocedural Binary Analysis

Matthew Revelle, Matt Parker, Kevin Orr
Kudu Dynamics
Chantilly, VA, USA
{mattr, mp, kevino}@kududyn.com

***Abstract*—Blaze is an open-source binary analysis framework that supports the construction and manipulation of interprocedural control-flow graphs (ICFGs) and type checking on a lifted representation of program binaries. All analyses in Blaze are implemented in terms of a typed intermediate language—Path Intermediate Language (PIL). Blaze includes a unification-based type checker for PIL which is used to support the generation of SMT formulas and type inference. Blaze has been used to develop tools for reverse engineering and vulnerability discovery and provides a foundation for exploring the use of type systems and higher-level abstractions in the analysis of program binaries. This paper provides an overview of Blaze's implementation, capabilities, and applications.**

I. Introduction

Manual reverse engineering is an often necessary step in the process of understanding vulnerabilities in program binaries, even as advances in automated binary analysis continue to be made. The tools available to reverse engineers can be improved by adding support for semi-automated analysis workflows which provide opportunities for reverse engineers to inform and direct automated analyses. We have developed a framework for interprocedural binary analysis—Blaze—which supports this goal of human-computer collaboration.

Blaze is a static analysis framework for program binaries that operates on interprocedural control-flow graphs (ICFGs), which are well-suited for presentation to—and interaction with—reverse engineers. While similar to the control-flow graphs (CFGs) [1] used in many binary analysis and reverse engineering tools, the basic blocks in Blaze ICFGs are defined such that all call sites are placed in separate basic block nodes known as *call nodes*. Call nodes may be *expanded* by substituting the CFG of a call target function in place of the call node. For indirect calls, which may have many possible target functions, Blaze supports user-specified targets for the expansion.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and the Naval Information Warfare Center (NIWC) under Contract No. N66001-22-C-4018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA and NIWC. Distribution A (approved for public release, distribution unlimited).

Workshop on Binary Analysis Research (BAR) 2023
3 March 2023, San Diego, CA, USA
ISBN 1-891562-84-3
https://dx.doi.org/10.14722/bar.2023.23009
www.ndss-symposium.org

ICFGs in Blaze assist with reverse engineering tasks in two ways. First, considering a function's CFG within a specific calling context may provide information that can be used to simplify that CFG by identifying and removing infeasible paths through *constraint-driven transformations*. This manual task, frequently performed by vulnerability researchers when investigating potential vulnerabilities, is automated by Blaze.

Second, reverse engineers frequently cycle between visualizations of function CFGs related to the current portion of the program they are reviewing as they form and test hypotheses of program functionality. Blaze ICFGs provide a single workspace for analyzing all of the connected, individual function CFGs. Additionally, ICFG *snapshots* can be created, which allows reverse engineers to explore the binary through modification of an ICFG and later revert to an earlier version. A user interface for this functionality is provided through a Blaze plugin for Binary Ninja.

Blaze imports basic program information from several established external code representations: Binary Ninja's Medium-Level Intermediate Language (BN MLIL) [33] and Ghidra's P-Code and High P-Code representations [25]. Those external formats are lifted into Blaze's intermediate language, Path Intermediate Language (PIL), which provides a common target representation for importing as well as for the implementation of analysis algorithms in Blaze.

Blaze defines a type system for PIL which is used to assist in the translation of PIL statements to satisfiability modulo theories (SMT) formulas [12] for reasoning about program constraints recovered from ICFGs. A unification-based type checker is provided that performs type inference and assigns types to PIL expressions. In addition to supporting the generation of SMT formulas, inferred PIL types are directly useful for reverse engineering in describing the sorts of values a variable may represent.

The main contributions of this paper are:

- The introduction of an open-source, static analysis framework—Blaze—that is focused on the analysis of program binaries through interprocedural analyses and type checking.
- A detailed overview of the framework implementation, including its architecture, integration with existing reverse engineering tools, intermediate representation, type system, and constraint-driven transformations on interprocedural control-flow graphs (ICFGs).
- A presentation of several applications of Blaze for reverse engineering tasks, including interactive modification of

ICFGs and type inference.

Blaze is an actively-developed, open-source project[1] used to support semi-automated binary analysis and the development of reverse engineering tools. All Blaze tools used in this paper will be made publicly available at: https://github.com/kudu-dynamics.

II. Related Work

Binary analysis frameworks provide a set of consistent APIs and analyses for performing program analysis on binary images. Many of these frameworks support common capabilities such as instruction disassembly, control-flow graph recovery, and data-flow analysis. Some frameworks provide additional capabilities, such as symbolic execution, path analysis, and type recovery, which are related to Blaze's characteristic features.

Symbolic execution [7], [17] is a popular technique for modeling the execution of a program by using symbolic values instead of concrete values. Individual instructions are interpreted by a symbolic execution engine and introduce constraints on symbolic values. An SMT solver is used to check these constraints and may either provide a satisfiable interpretation which includes assignments of concrete values to symbols, or an indication that the constraints are unsatisfiable. Symbolic execution is frequently used to find program inputs which lead to a certain execution path. angr [30], BAP [5], BINSEC [11], and Miasm2 [13] are binary analysis frameworks which include symbolic execution engines. KLEE [6] (when used with McSema [15]), Manticore [24], S2E [8], and SAGE [17] are all standalone symbolic execution engines with support for program binaries.

While not a binary analysis framework or symbolic execution engine, SENinja [4] is a tool which integrates angr's symbolic execution engine with Binary Ninja's graphical user interface [33]. This integration is similar to that provided by the Blaze plugin for Binary Ninja.

Blaze does not perform symbolic execution, but it does use symbolic constraints recovered from branch conditions and static single assignment (SSA) variables to automatically simplify interprocedural control-flow graphs through a context- and path-sensitive analysis. In contrast to symbolic execution, individual instructions or statements are not interpreted, and the goal is to assist a user in focusing on the relevant portions of a program given their current assumptions.

Pharos [29] and its Java implementation—Kaiju [34]—use interprocedural analyses similarly to Blaze. Kaiju has been used to implement the GhiHorn [16] tool which is capable of determining the feasibility of a path through a program. While GhiHorn and Blaze both use SMT solvers to check path feasibility, Blaze supports an interactive construction and modification of interprocedural control-flow graphs which allows a gradual refinement of the program to a set of feasible paths. In contrast, GhiHorn provides an ability to search for a specific path matching a user-provided query.

There is existing work in type recovery from program binaries for the purposes of program variable recovery [2], [20] and decompilation [9]. The binary analysis frameworks angr [30], BAP [5], BinCAT [3], CodeSurfer/x86 [26], and REV.NG [14] perform type recovery as a part of one or both of these tasks. In program variable recovery, algorithms have been proposed to group related abstract locations [2] used to store program variables and then infer both the presence of a unique program variable and the associated type which approximates the program variables found in the source code for the program. There has also been work on type recovery apart from any particular binary analysis framework. These techniques include static [35], [2] and dynamic [31], [22] approaches.

Similarly, decompilation includes the recovery of source program variables and their types as part of the more general task of reproducing source code that corresponds to the program binary. Type recovery aims to assign types from the source programming language to recovered program variables. In contrast, Blaze's PIL has its own type system, and Blaze provides a unification-based type checker that directly infers types for PIL terms. PIL types are sufficiently expressive to avoid ambiguities found in the type system of source languages such as C.

III. Interprocedural Binary Analysis

A. Framework Overview

Blaze is a binary analysis framework that provides control- and data-flow analyses, constraint modeling, an interprocedural program representation, and transformations on that representation. Blaze is implemented in Haskell and integrates with two popular reverse engineering platforms, Binary Ninja and Ghidra. PIL is Blaze's intermediate language for analysis and is used as a common representation for Binary Ninja's MLIL and Ghidra's P-Code and High P-Code. Interprocedural analysis in Blaze is performed on ICFGs which contain PIL statements. These graphs are similar to standard CFGs but permit the substitution of function calls with the control-flow graphs of the call target.

The Blaze framework is organized into multiple components that provide capabilities such as importing of program information, graph analyses, PIL analyses, constraint modeling, and type checking, as shown in Figure 1. The *Importer* component includes abstract interface definitions as Haskell type classes for importing function control-flow graphs, call graphs, and PIL instructions. Support for importing from both Binary Ninja and Ghidra is included with Blaze, but the design of the Importer allows Blaze users to add support for other sources of program information.

The *PIL Analysis* component includes support for analyzing and rewriting PIL statements. Specifically, this component provides data dependence analysis, copy propagation, constant propagation, and simplifying rewrites for various PIL expressions based on analysis results.

The *Type Checker* component defines a type system for PIL and implements a unification-based type checking and inference algorithm. Blaze can propagate type information within an ICFG, infer PIL types, and utilize type annotations for known functions.

[1] https://github.com/kudu-dynamics/blaze

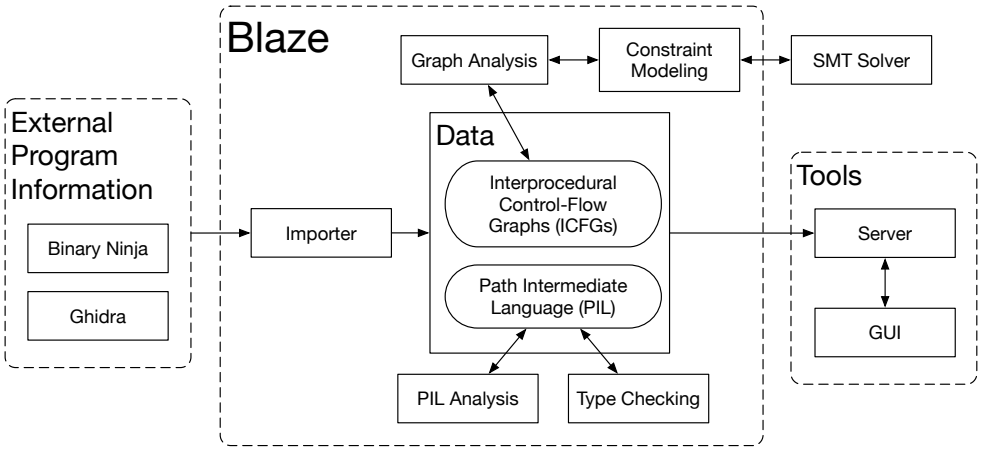


Fig. 1. Architecture of the Blaze framework and related external components.

Blaze's *Graph Analysis* component provides standard analyses for control-flow graphs as well as support for the construction and manipulation of ICFGs. The *Constraint Modeling* component converts ICFGs and PIL statements to SMT formulas and is used by the *Graph Analysis* component to automatically simplify ICFGs.

We developed a Blaze Analysis *Server* that provides access to Blaze capabilities and a Blaze Analysis *Graphical User Interface* as a Binary Ninja plugin that communicates with the server and allows users to construct and transform ICFGs within Binary Ninja. The Server also provides the ability to persist and load snapshots of ICFGs.

Some details of these components are discussed in the following subsections under Section III.

B. PIL and ICFGs

PIL includes over 90 operations and 20 types of statements, and we present an abbreviated grammar in Figure 2. Note that some PIL terms rendered by the Blaze plugin have a slightly different surface syntax. Memory store operations and all operations that affect control flow—calling a procedure, returning from a called procedure, and jumping—are made explicit at the statement level. PIL expressions include integer, floating point, boolean, and pointer constants, as well as composite expressions made from fixed-arity operations.

Though not a part of this presentation of the PIL syntax, each abstract syntax tree (AST) node within an expression can be annotated with a PIL type as described in Section III-C. As Blaze is implemented in Haskell, this is accomplished without adding complexity to the core syntax through the use of the *data types à la carte* technique [32]. In fact, using this approach, all the nodes of a PIL AST can be annotated with any kind of data.

Blaze control-flow graphs are flow graphs—rooted, directed graphs—containing basic block nodes and function call nodes. PIL statements in Blaze CFGs are in static single assignment (SSA) form. Basic block nodes contain a linear sequence of PIL statements, and function call nodes contain a single PIL call statement. Each CFG corresponds to a single function. Basic blocks always have at least one statement; their final statement may be a conditional branch, indirect jump, or switch statement. CFG nodes are connected by control-flow edges, and if a node has no out-edges, then it is considered a terminal node in the CFG. Otherwise, a node will either lead unconditionally to a single successor node or will branch to two or more successor nodes. In the case of a conditional branch, each out-edge is labeled with either `True` or `False`, indicating in which case that edge will be followed.

While CFGs only contain basic block nodes and function call nodes, *interprocedural CFGs* can also contain *enter nodes* and *leave nodes*. Each enter and leave node represents a change

Statements σ ::=

$x := e$	Assignment
$\mid x := \phi(\vec{x})$	Phi assignment
$\mid e_1 \leftarrow e_2$	Memory store
$\mid$ CALL f $(\vec{e})$ $\mid$ RET e	Procedure call, return
$\mid$ JUMP e $\mid$ IF e	Indirect jump, conditional branch
$\mid$ JUMPTO e_1 $(\vec{e})$	Switch statement

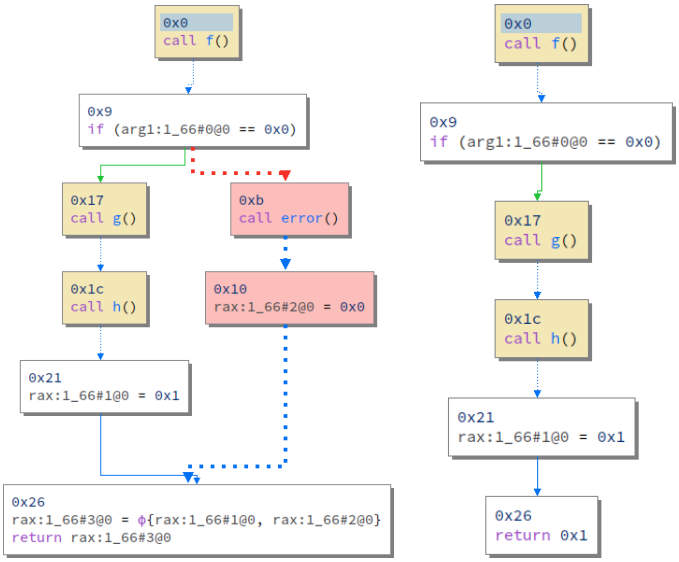
Expressions e, P, Q ::=

$n_i \mid n_f \mid b \mid p$	Integer, float, bool, pointer constants
$\mid x$	Variables
$\mid \&x$	Addresses of stack locals and parameters
$\mid e_1 \oplus e_2$	Binary operations
$\mid \lceil e \rceil \mid \lfloor e \rfloor$	Float truncation
$\mid \neg e$	Bitwise logic
$\mid [e]$	Memory load

Binary operators $\oplus$ ::=

$+_i \mid -_i \mid *_i \mid \div_s \mid \div_u$	Integer arithmetic
$\mid +_f \mid -_f \mid *_f \mid \div_f$	Float arithmetic
$\mid \gg_a \mid \gg_l \mid \ll$	Bit-shifting
$\mid =_i \mid \neq_i \mid =_f \mid \neq_f$	Integer, float equality
$\mid <_s \mid \leq_s \mid >_s \mid \geq_s$	Signed integer comparison
$\mid <_u \mid \leq_u \mid >_u \mid \geq_u$	Unsigned integer comparison
$\mid <_f \mid \leq_f \mid >_f \mid \geq_f$	Float comparison
$\mid$ AND $\mid$ OR $\mid$ XOR	Bitwise logic

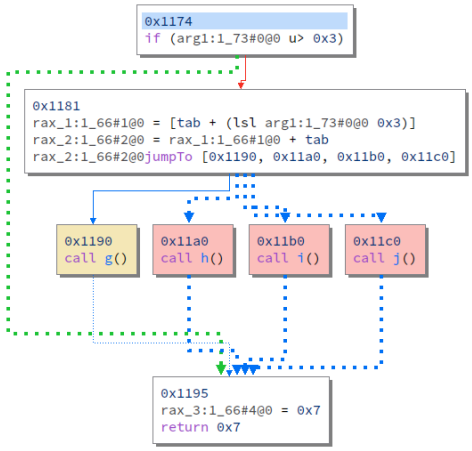
Fig. 2. Abbreviated grammar for PIL statements and expressions.



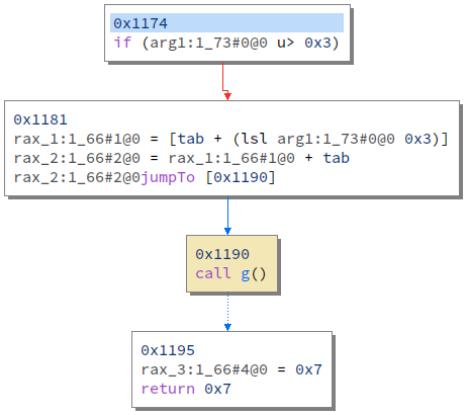
(a) Before. A conditional edge has been selected by the user to be pruned; nodes that will be removed after carrying out the operation are highlighted in red, and edges to be removed are dotted.

(b) The ICFG after the prune operation is completed.

Fig. 3. A demonstration of the *prune* operation.



(a) Before. The left-most successor of the switch statement block has been selected by the user to be focused on; nodes that will be removed are highlighted in red, and edges to be removed are dotted.



(b) The CFG after the focus operation is completed. All immediate successors of the switch statement block (except for the focused block) have been removed, and the switch statement has been simplified.

Fig. 4. A demonstration of the *focus* operation.

of context between a caller and callee. An enter node represents switching from a caller's context to its callee's context, and includes assignment statements that link the caller's call site arguments to the callee's function parameters. A leave node represents returning back to a caller's context and includes an assignment statement that captures the callee's return value.

Blaze implements a number of transformations on ICFGs that can be directed either by a user or by an automated analysis:

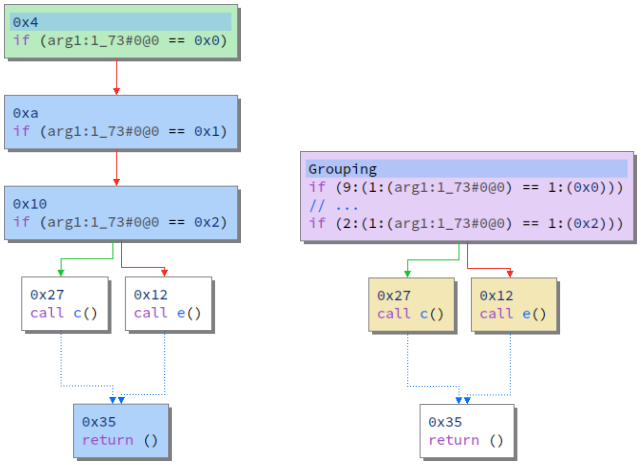
Call-node expansion A function call node can be *expanded*, which replaces the function call node with the CFG of the callee. At the boundary between the caller and callee, two new nodes are inserted: an enter node and a leave node. The subgraphs of expanded call nodes may introduce new call sites which the user may also expand. The user can expand recursive and mutually-recursive procedures an arbitrary number of times.

Conditional edge pruning Conditional edges can be selected by the user to be *pruned*. The edge will be removed, and any blocks which become unreachable as a result will also be removed. Figure 3 shows the result of pruning a conditional edge. If an ICFG is thought of as a compact representation of all paths through a region of the program being analyzed, then pruning removes any path that includes the targeted edge.

Node focusing While pruning removes any path that includes a specific edge, *focusing* removes any path which excludes a specific node. When the user focuses on a node, Blaze will remove all nodes except those which are either reachable from the focused node or can reach the focused node. In other words, focusing removes any nodes which do not coexist with the focused node on any path through the ICFG. Focusing allows the user to mark a node of interest and remove all paths in the ICFG which do not include the focused node. Figure 4 shows an example of focusing on a node.

Node grouping Certain subgraphs—called *groups*—within an ICFG can be collapsed into a compact *grouping node*. Intuitively, a group is a subgraph where all nodes are only connected to other nodes in the group except for a *start node* and an *end node*. The start node may have in-edges whose source node is not a member of the group, and the end node may have out-edges whose destination node is not a member of the group. These are analogous to the requirements that are imposed on basic blocks: there must be no stray control-flow edges into or out of the group except at its start and end. After replacing the group with a grouping node, any in-edges to the start node and out-edges from the end node are redirected to the new grouping node. This grouping node contains the original group which can can be restored by expanding the grouping node.

Blaze identifies candidate end nodes which are dominated by the start node and post-dominate all group nodes. Figure 5 shows an example of grouping and the identification



(a) Before. The user has selected the first block as the group start node. Blaze highlights candidate end nodes in blue. Note that the two call nodes are not candidate end nodes because the preceding node at `0x10` would then have a stray out-edge.

(b) After the user selects the node at `0x10` as the group end node, Blaze collapses the group containing the first, second, and third nodes into a grouping node, which displays a summary of the group it contains. The out-edges from the end node are replaced with out-edges from the grouping node.

Fig. 5. A demonstration of the *group* operation.

of possible end nodes.

C. Type Checking

We designed a type system for PIL and implemented a type checker for it in Blaze. A goal of the PIL type system is to provide types for the program as lifted from the binary that are more expressive than those available in the original source language and assist in reverse engineering. We hope this may ultimately lead to type checking as automated vulnerability discovery.

The PIL type checker produces an assignment of types to PIL expressions. The type checker is typically used to check the PIL statements associated with an ICFG, but any collection of statements can be provided as input. Blaze's type checker uses unification—a common approach for type checking in functional programming languages [19], [23], [21]—to resolve type constraints and find appropriate substitutions. Type inference occurs as unifying substitutions are selected based on the constraints provided by the PIL statements being checked. The type checker is used for three capabilities in Blaze: 1) inference and assignment of types to expressions; 2) conversion of PIL statements to SMT formulas; and 3) checking for type errors.

The PIL types and *type constructors* are listed in Table I and provide ways to represent common values that occur in programs. Type constructors refer to partially-defined types and are used to distinguish a set of possible types, `Int w s`, from a concrete type such as `Int 32 Signed`. The `w` and `s` symbols in `Int w s` denote metavariables which may be replaced with a concrete value that provide information about the bit-width and signedness. Support for C structs, C++ classes, and general data structures can be implemented in terms of the `Record m` and `Array n T` type constructors.

Record types have an associated map that describes the fields associated with the type. The map keys are field offsets

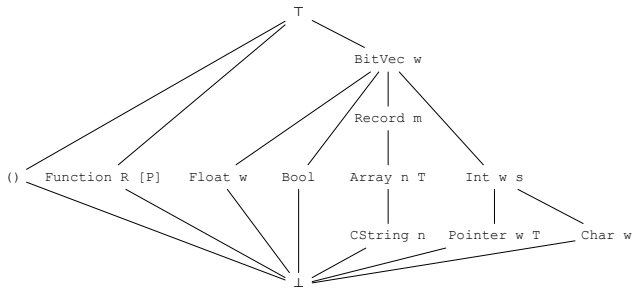


Fig. 6. The type lattice depicting the subtyping relation for PIL types. Metavariables are used as placeholders to reduce the visual complexity from instantiating possible types.

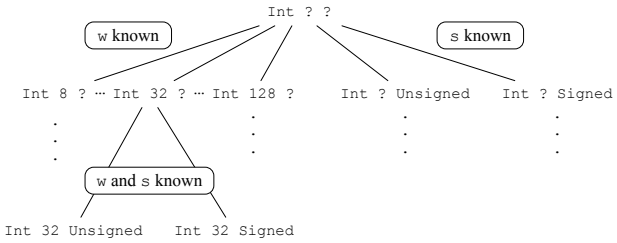


Fig. 7. An expanded view of the `Int w s` type constructor and the subtypes defined by adding information about the bit-width and signedness of an integer. Unknown values are represented with `?`.

expressed in bits. For example, a record type for a two-dimensional point may be denoted as: `Record {0: Float 32, 32: Float 32}` and may correspond to a C struct, such as: `struct {float x; float y;};`.

The type lattice for PIL types is shown in Figure 6 and depicts the subtype relation between the available types. In Figure 7, an expanded view of the `Int w s` type constructor is shown with the bit-width `w` and signedness flag `s` metavariables instantiated with concrete values. This figure shows how the incorporation of additional information about a type is captured through subtyping. We use `?` to denote unknown values. As information is added, unification between various integer types becomes infeasible. For example, the types in the constraint `(Int 32 ?) = (Int ? Signed)` will unify to `Int 32 Signed`, but the types in the constraint `(Int 32 ?) = (Int 8 ?)` cannot unify because neither type is a subtype of the other.

Similarly, not all instantiations of `Int w s` are subtypes of all instantiations of `BitVec w`. Consider a bit vector with a known bit-width of 32, `BitVec 32`; only 32-bit integers are subtypes and will unify with it. An example of the subtype

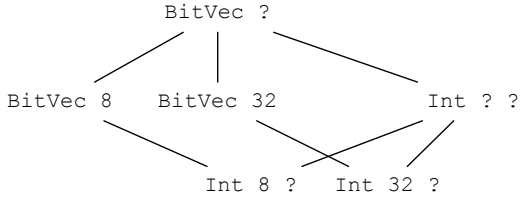


Fig. 8. The subtype relationship between several instantiations of the `BitVec w` and `Int w s` type constructors.

TABLE I. THE TYPES AND TYPE CONSTRUCTORS AVAILABLE IN THE PIL TYPE SYSTEM. SINGLE-LETTER METAVARIABLES ARE USED AS PLACEHOLDERS FOR UNSPECIFIED VALUES (LOWERCASE) AND TYPES (UPPERCASE).

Name	Description
`BitVec w`	Bit vector types specified by bit-width `w`.
`Int w s`	Integer types specified by bit-width `w` and signedness flag `s`.
`Pointer w T`	Pointer types specified by the pointer bit-width `w` and pointee type `T`.
`Float w`	Float types specified by bit-width `w`.
`Char w`	Character types specified by bit-width `w`.
`Bool`	The Boolean type.
`Record m`	Record types specified by a map `m` of field offsets to field types.
`Array n T`	Array types specified by a length `n` and element type `T`.
`CString n`	Null-terminated sequence of `Char 8` values specified by length `n`.
`Function R [P]`	Function types specified by return type `R` and variable number of parameter types `[P]`.
`()`	The Unit type inhabited by the single value `()`.
$\top, \bot$	The Top and Bottom types.

relation between instantiated types is provided in Figure 8.

There are two distinct phases in a unification approach for type checking and type inference. In the first phase, Blaze traverses all the PIL statements and expressions in an ICFG and generates unification variables for PIL terms, such as PIL variables, constants, and other expressions. These unification variables are assigned type constraints. Blaze's type checker uses two types of constraints: type-equality constraints and subtype constraints. The type-equality constraints state that the types assigned to two unification variables must match—or unify—with each other. As an example, a type-equality constraint between two unification variables would be added to the PIL variables in an assignment statement. The statement `x := y` assigning the value of PIL variable `y` to PIL variable `x` would introduce the constraint that the type of `x` and the type of `y` must be the same: $\texttt{x:T}_1\texttt{, y:T}_2\texttt{, T}_1 = \texttt{T}_2$, where $\texttt{T}_1$ and $\texttt{T}_2$ are unification variables.

Additionally, Blaze supports subtype constraints that are added to unification variables based on uses of the related PIL term. For example, the $+_i$ operator represents integer addition and $\texttt{x} +_i 1$ implies that `x` is an integer. Let us assume that the bit-width size of `x` is 32 bits. Given the use of `x` in this addition operation and its known size, Blaze will add the subtype constraint `x <: (Int 32 ?)`. Note that the signedness flag cannot be inferred from this expression as indicated by `?`.

A subtype relation is reflexive; thus this subtype constraint asserts `x` must be an `Int 32 ?` or another more-specific subtype. Once these type constraints have been generated, Blaze then attempts to find the most general type assignments that meet those constraints. While solving for these constraints, Blaze is performing type checking in addition to type inference. If no valid type assignment is possible given a set of constraints then Blaze will report a type error for the constraints that could not be met.

We implemented a standard unification algorithm for Blaze to resolve the type constraints. Constraints are stored in a list and unified one at a time. Additional constraints may be introduced during unification and are added to the list of constraints. For example, resolving a constraint between two pointer types $\texttt{Pointer 64 T}_1 = \texttt{Pointer 64 T}_2$ introduces a constraint, $\texttt{T}_1 = \texttt{T}_2$, between the pointee types' unification variables.

Once all constraints have been processed, every PIL term is associated with a PIL type. If a constraint cannot be resolved because no valid substitution is possible, then the $\bot$ (`Bottom`) type is assigned and a type error is reported. The results of type checking—type assignments, errors, and other related information—are provided as a result.

D. Constraint-Driven Transformations

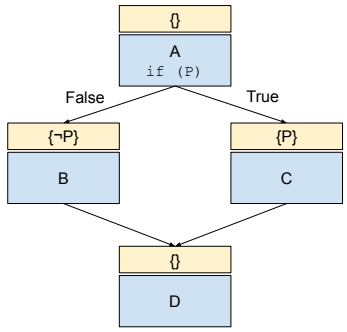
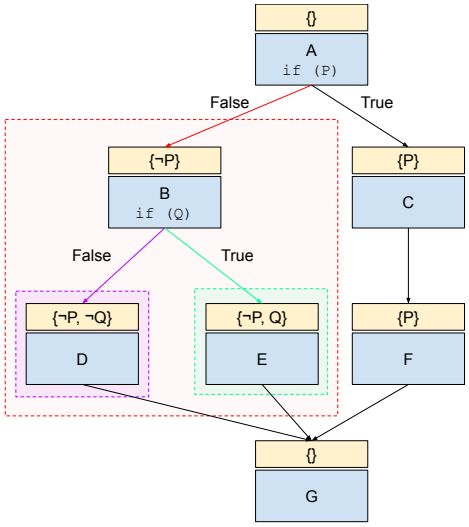


Fig. 9. A simple ICFG with a single conditional branch. Contextual constraints are shown above each basic block. For example, because node `B` is dominated by the false branch of `if (P)`, it has the constraint $\neg P$.

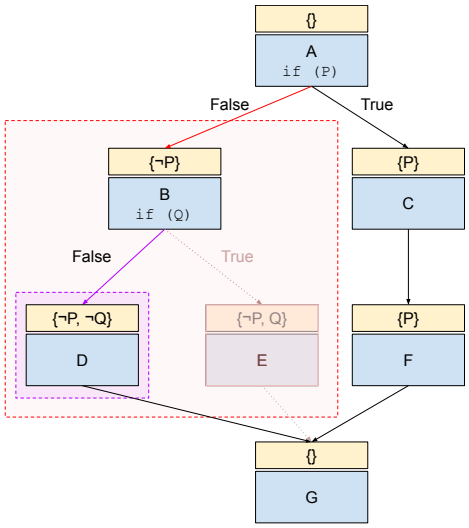
As ICFGs grow in size and complexity, the number of conditional branches typically increases, introducing constraints in each branch. For instance, a branch condition, $\texttt{x} =_i 0$, will introduce $\texttt{x} \neq_i 0$ in the false branch and $\texttt{x} =_i 0$ in the true branch, and will be in effect for all nodes dominated by that conditional edge. Constraints from branch conditions can accumulate and by testing the satisfiability of these constraints within their different contexts we can perform constraint-driven transformations. These transformations automatically eliminate infeasible conditional edges and can significantly reduce the size and complexity of an ICFG.

For example, Figure 9 shows an if-statement in node `A`. If control flow reaches `C`, then P must have been true, and if it reaches `B`, P must have been false. All nodes that are dominated by `C` will nominally only be executed if P is true, and all nodes that are dominated by `B` will only be executed if P is false. Control-flow nodes dominated by a conditional edge are said to be within the same *branch context*. A constraint

determined by the branch condition and a conditional edge is associated with each branch context.



(a) Nodes B, D, and E are all within the branch context of the false conditional edge of node A (red box), and all share the $\neg P$ constraint. Within that branch context, there are two nested branch contexts descending from node B (fuchsia and green boxes).



(b) After $\neg P \wedge Q$ is recognized as unsatisfiable, node E is automatically removed.

Fig. 10. A more complex ICFG with nested branch contexts, before and after automatic pruning of infeasible paths. For this example, P and Q are Boolean expressions such that $\neg P \wedge Q$ is unsatisfiable.

There may also be nested branch contexts, where there exists a branch context within a branch context, as in Figure 10. Because E is dominated by both the false branch of `if (P)` in node A, and dominated by the true branch of `if (Q)` in B, the branch-context constraints in E are $\{\neg P, Q\}$. In complex ICFGs, nodes may exist within deeply-nested branch contexts. Sometimes, the accumulated constraints conflict and are unsatisfiable. For instance, if the conjunction of the branch context constraints $\{\neg P, Q\}$ is unsatisfiable, then all nodes within the branch context are unreachable. Because they are unreachable, all the nodes dominated by the true edge can be eliminated from the ICFG.

Because PIL statements in ICFGs are in SSA form, the constraints produced by the assignment statements within an ICFG are not dependent on control flow. In Blaze, these constraints are called *base constraints*. Each conditional edge introduces a branch context with corresponding branch-context constraints. For each branch context, Blaze generates an SMT formula that combines the base constraints with the constraints of the current branch contexts. This formula is checked for satisfiability; if it is unsatisfiable, then the conditional edge is removed. Any nodes unreachable from the ICFG root are then removed.

IV. Applications

A. Automated ICFG Simplification

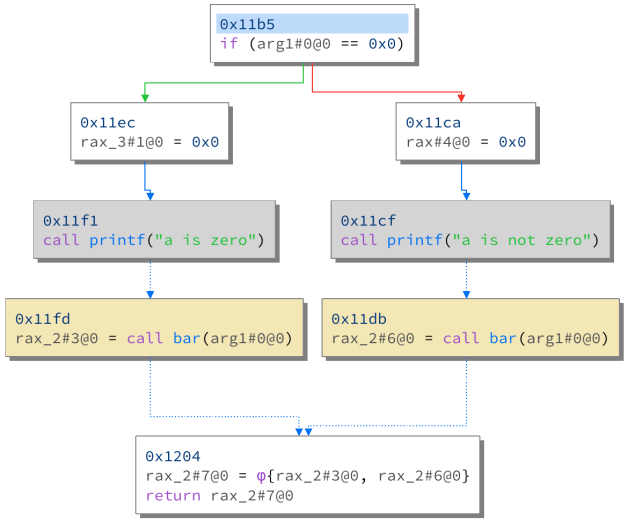


Fig. 11. A `foo` function that calls the `bar` function from two different branch contexts.

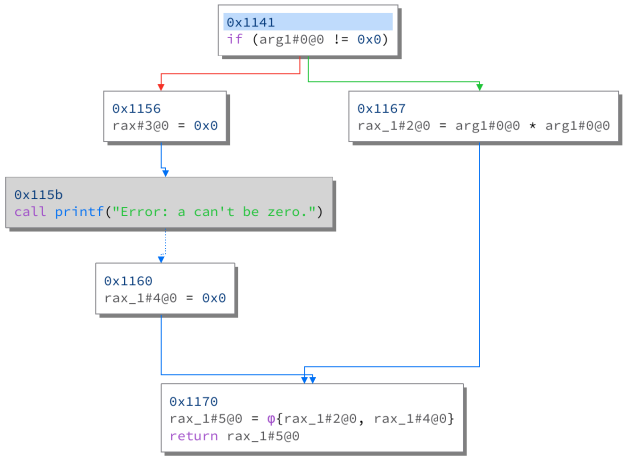


Fig. 12. The `bar` function, which tests if its first argument is zero.

Blaze can statically determine if conditional edges are infeasible under a given branch context and can therefore be pruned. For example, Figure 11 shows a function `foo` which twice calls function `bar`—shown in Figure 12. The `arg1` variable of function foo is provided as an argument at both of the `bar` call sites. At the `IF` statement in the root node of the

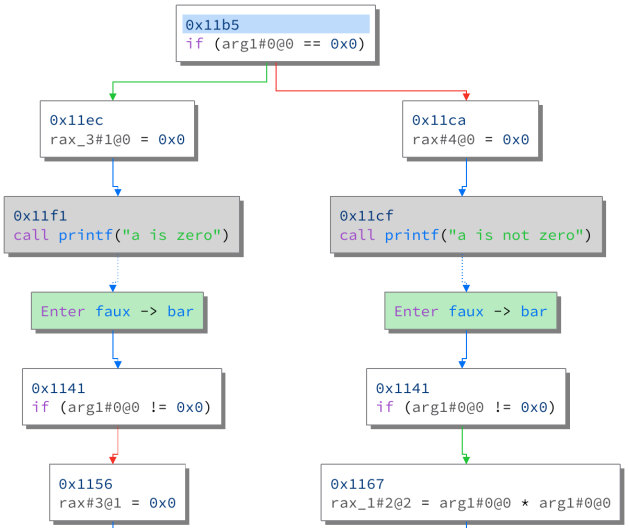


Fig. 13. The `foo` function, with both calls to `bar` expanded. Blaze is able to automatically prune the infeasible conditional edges in each instance of `bar` from the constraint on `arg1` introduced in `foo`.

ICFG, the left branch context asserts $\texttt{arg1} =_i 0$ and the right branch context asserts $\texttt{arg1} \neq_i 0$. The function being called, `bar`, checks if its argument is zero and branches accordingly. Figure 13 shows the result of expanding both call sites in a Blaze ICFG within these two calling contexts. The constraints from each of the calling contexts are used to automatically prune the respective infeasible conditional edges in `bar`.

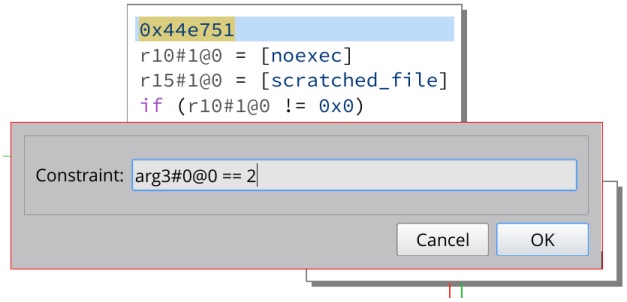


Fig. 14. Example of a user manually introducing a constraint to an ICFG created from the `server_updated` function in `cvs`.

A user may also directly specify global constraints on PIL variables in ICFGs. For example, in Figure 14, a user introduces a constraint in the root node of the `server_updated` function in a modified version of the CVS program [10]. The third argument to the `server_updated` function corresponds to a file status, and a user may be interested in only certain types of server responses. Using this constraint, Blaze is able to prune away the irrelevant portions of the ICFG, as seen in Figure 15.

B. Type Inference

The PIL type system and type checker in Blaze—described in Section III-C—provides type inference designed for binary analysis. While the PIL type system is distinct from the type system found in source languages such as C and C++, type inference in Blaze can perform a similar role to that of

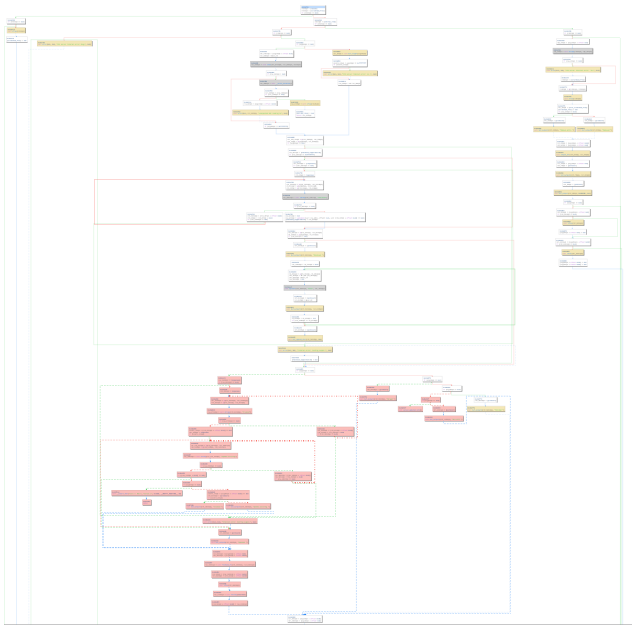


Fig. 15. Blaze uses the constraint entered manually by a user in Figure 14 to automatically prune away the red-highlighted nodes in the ICFG.

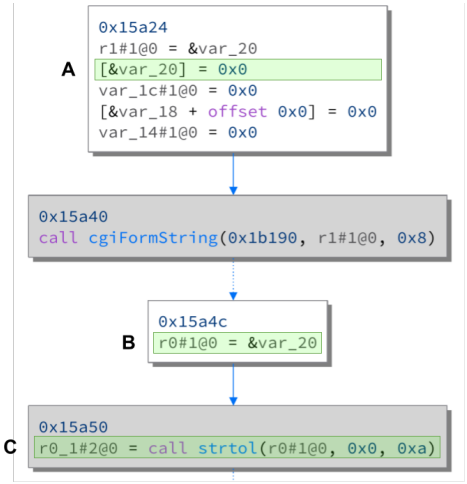


Fig. 16. An example scenario of type unification in a binary from an IoT device. Statement **A** is a store to an address of a PIL variable. It provides an incomplete hint about the size of `var_20`. At **B**, an assignment statement produces a type-equality constraint between `r0#1@0` and the address of `var_20`. At **C**, a call to a Standard C Library function with known PIL types provides a subtype constraint on `r0#1@0`.

type recovery when reverse engineering a program binary. We present two scenarios of how type inference in Blaze can be used to produce additional type information by unifying a set of related constraints and recognizing recursive types.

All PIL terms are associated with type constraints that link related type hints across statements. As these constraints are resolved, more specific types for every PIL term may be inferred. In the scenario shown in Figure 16, Blaze infers `var_20` is a `CString 4` by using the available subtype constraints. Figure 17 depicts the unification of these constraints. Statement **A** provides hints that `var_20` is a 32-bit bit vector and is referenced through a 32-bit pointer. At Statement

```
&var_20  : T1
     T1 <: Pointer 32 T2
A  var_20  : T3
     T2  = T3
     T3 <: BitVec 32

B    r0#1  : T4
       T1  = T4

   strtol  : T5
     T5 <: Function T6 [T7 T8 T9]
C      T4  = T7
     T7 <: Pointer 32 T10
    T10 <: CString ?
```

(a) First, type constraints are generated from all PIL statements and expressions within the ICFG. For this example, we only need to examine constraints from the statements labeled **A**, **B**, and **C** in Figure 16. The relevant type constraints introduced by those statements are listed here with the corresponding statement label.

$$\frac{T_2 = T_3 \quad T_1 = T_4 \quad T_4 = T_7}{\begin{array}{l} T_1 <: \text{Pointer } 32\ T_2 \\ T_2 <: \text{BitVec } 32 \\ T_1 <: \text{Pointer } 32\ T_{10} \\ T_5 <: \text{Function } T_6\ [T_1\ T_8\ T_9] \\ T_{10} <: \text{CString ?} \end{array}}$$

(b) All unification variables that occur on the right-hand side of a type-equality constraint from (a) are substituted with the corresponding left-hand side in all constraints.

$$\frac{\begin{array}{l} T_1 <: \text{Pointer } 32\ T_2 \\ T_1 <: \text{Pointer } 32\ T_{10} \end{array}}{T_2 = T_{10}}$$

(c) The right-hand sides of two subtype constraints on T_1 are unified together to produce an additional type-equality constraint. This corresponds to the requirement that the pointee types must unify.

$$\frac{T_2 = T_{10}}{T_2 <: \text{CString ?}}$$

(d) The new $T_2 = T_{10}$ type-equality constraint is substituted against a subtype constraint derived in (b).

$$\frac{\begin{array}{l} T_2 <: \text{BitVec } 32 \\ T_2 <: \text{CString ?} \end{array}}{T_2 <: \text{CString } 4}$$

(e) A subtype constraint from (b) is unified against the new constraint from (d). Recall that `BitVec` is parameterized over a bit-width while `CString` is parameterized over a string length. After this step, all constraints have been unified. This step unifies two subtype constraints on `var_20`.

Fig. 17. A unification example for variable `var_20` based on the PIL statements from Figure 16. The unification algorithm terminates once all generated constraints have been resolved through substitution.

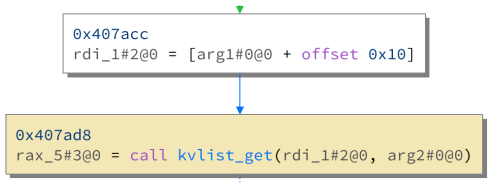


Fig. 18. A recursive call within the function `kvlist_get` from the `bryant` challenge program [10]. The `arg1#0@0` variable has the same type as the field at `arg1#0@0` $+_i$ `0x10`.

B, an assignment produces a type-equality constraint between `r0#1@0` and the address of `var_20`. Finally, in Statement **C**, a call to a `strtol`, a Standard C Library Function with a known PIL type, provides a subtype constraint on `r0#1@0`.

The result of unifying the type constraints from these statements is that a 32-bit bit vector (`BitVec 32`) unifies with a C string of unknown length (`CString ?`) through type equality constraints on pointers. This results in `CString 4` as the inferred type of `var_20`. Unfortunately, this inferred type is not entirely correct. There are an additional four bytes of adjacent storage on the stack which should be associated with `var_20`. If the type checker were provided with accurate storage information then it could correctly infer the `CString 8` type. Future improvements to Blaze's static analysis of storage locations can provide more accurate information.

Blaze is also able to infer recursive data types, such as linked lists. In Figure 18, a function named `kvlist_get` from the `bryant` challenge program [10] recursively calls itself, passing in the contents of the field at offset `0x10` bytes—128 bits—as the first argument of its recursive call. From this, constraints between the arguments of the recursive call and the function parameters are formed. Blaze then is able to infer the recursive field in the linked-list node structure:

```
T = Pointer 64
      (Record
       { 0   : Pointer 64 (CString ?)
       , 64  : BitVec 64
       , 128: T })
```

The type `T` is a record with a pointer to a C string of an unknown length in the first field and a recursive reference to its own type in the third field. The second field is another pointer to a C string, but within the context of `kvlist_get` there is only enough information provided to infer it is a 64-bit bit vector.

C. Guided Search

Blaze is particularly well-suited for exploring ways to reach points of interest (POIs) in a program, such as possible vulnerabilities found by automated analysis. For example, if there is a call to `sprintf` that appears to use an attacker-controlled format string, then it may lead to a format string vulnerability. Whether or not that vulnerability can be exercised depends on several factors, one of which is whether the location is reachable from the attack surface. Blaze's guided search feature can help a vulnerability researcher construct an ICFG between an attack vector and a POI. Blaze's focus and pruning features, as well as the accompanying automated

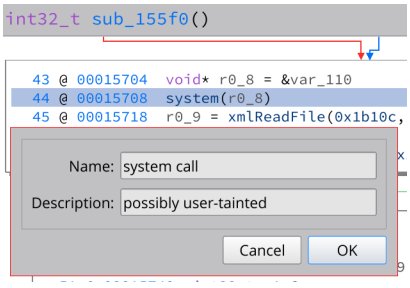


Fig. 19. A user inputs a point of interest through the Blaze plugin in Binary Ninja.

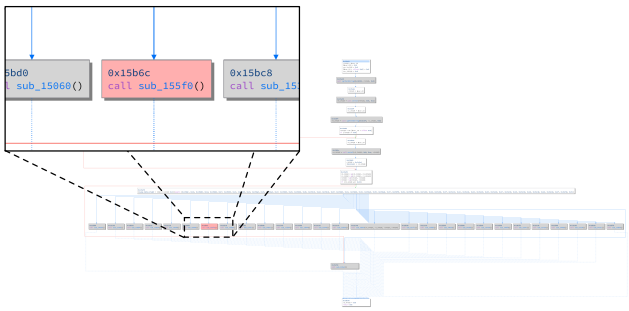


Fig. 20. When guided search is enabled, Blaze highlights all call sites that reach the selected target POI. In this example, a single call in the `cgiMain` function of a web server from an IoT device reaches the POI location defined in Figure 19 and is highlighted in red.

simplification, complement guided search by narrowing the code which must be reviewed.

A POI is required to use the guided search feature. Users can mark a particular location as a POI, giving it an optional name and description, as seen in Figure 19. This POI marks a possible command injection vulnerability in a web service running on an IoT device. The program's entry function, `cgiMain`, calls many functions that never reach the POI. However, when the POI is enabled in Blaze's guided search, all call sites that lead to the POI are highlighted. Figure 20 shows the `cgiMain` function with guided search enabled. A single call site among the 28 expandable calls is highlighted in red, making it clear to the user which path to follow. Using Blaze's *focus* feature, then expanding the call site, results in a simplified ICFG that links the entry function and the POI.

V. Limitations and Future Work

Development of Blaze is ongoing and we plan to improve the framework in several ways. Our immediate plans include support for interprocedural path analyses, extensions to the PIL type system, and the addition of a functional-style code representation designed for program analysis.

Blaze was originally designed for interprocedural paths and earlier analyses were written in terms of these paths rather than interprocedural control-flow graphs. Previously, Blaze supported testing path feasibility, path sampling, and path differencing. We plan to rewrite these analysis algorithms in terms of ICFGs where an interprocedural path can be represented as an ICFG in which all conditional branches have been pruned to have a single conditional edge. We have found these features useful in practice and they provide a foundation for other capabilities.

We are developing improvements to the PIL type system that we expect will support using type checking as a means to discover vulnerabilities in program binaries. The most notable planned improvements are support for refinement types and sum types in PIL. Refinement types [27], [28] will permit adding inferred predicates to existing PIL types and support the use of PIL type checking as a method for finding vulnerabilities that occur from incorrect bounds checking. The addition of sum types [18] will support the lifting of low-level dispatch code to structural pattern matching in PIL. We are also considering how to incorporate type annotations supplied interactively by a user; providing another opportunity for user-directed automation.

Additionally, we are considering how a functional-style code representation for program analysis could be well-suited for composable, partial analysis of a program binary and improved reasoning over memory and loops.

VI. Conclusion

Blaze is an open-source, binary analysis framework written in Haskell which provides a foundation for research and tool development that leverages interprocedural analysis and type systems. Interprocedural CFGs provide reverse engineers with additional context and opportunities to focus and simplify the region of the program being analyzed, and typed PIL statements assist in recovering and communicating program semantics. Interprocedural CFGs and typed PIL statements are the prominent code representations in Blaze and together make Blaze well-suited for developing interactive reverse engineering and vulnerability discovery tools.

In this paper, we presented a technical description of the functionality and implementation of Blaze. We also demonstrated the use of Blaze for interaction with ICFGs and PIL type inference through a Binary Ninja plugin. The source code of Blaze is available from: https://github.com/kudu-dynamics/blaze.

Acknowledgments

We would like to thank Evyatar Ben-Asher and Julian Chan who provided their feedback on an earlier version of this paper.

The original development of Blaze was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No. FA8750-19-C-0005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA and AFRL.

References

[1] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[2] G. Balakrishnan and T. Reps, "DIVINE: DIscovering Variables IN Executables," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2007, pp. 1–28.

[3] P. Biondi, R. Rigo, S. Zennou, and X. Mehrenberger, "BinCAT: purrfecting binary static analysis," in *Symposium sur la sécurité des technologies de linformation et des communications*, 2017.

[4] L. Borzacchiello, E. Coppa, and C. Demetrescu, "SENinja: A symbolic execution plugin for Binary Ninja," *SoftwareX*, vol. 20, p. 101219, 2022.

[5] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[6] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[7] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[8] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–49, 2012.

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

[10] Cromulence, "CHESS Challenges," 2021, [Online; accessed 29-December-2022]. [Online]. Available: https://github.com/cromulencellc/chess-aces

[11] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion, "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 653–656.

[12] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[13] F. Desclaux, "Miasm2," 2012, [Online; accessed 29-December-2022]. [Online]. Available: https://github.com/cea-sec/miasm

[14] A. Di Federico, M. Payer, and G. Agosta, "rev.ng: a unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 131–141.

[15] A. Dinaburg and A. Ruef, "McSema: Static translation of x86 instructions to LLVM," in *ReCon 2014 Conference, Montreal, Canada*, 2014.

[16] J. Gennari, "GhiHorn: Path analysis in Ghidra using SMT solvers," 2021, [Online; accessed 29-December-2022]. [Online]. Available: https://insights.sei.cmu.edu/blog/ghihorn-path-analysis-in-ghidra-using-smt-solvers/

[17] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[18] M. Greenberg, S. Malewski, and E. Tanter, "Gradual algebraic data types," in *Informal Proceedings of the ACM SIGPLAN Workshop on Gradual Typing (WGT20)*, 2020.

[19] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson *et al.*, "Report on the programming language Haskell: a non-strict, purely functional language version 1.2," *ACM SIGPLAN Notices*, vol. 27, no. 5, pp. 1–164, 1992.

[20] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," 2011.

[21] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon, "The Objective Caml System release 3.11," *Documentation and user's manual. INRIA*, 2008.

[22] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution." in *NDSS*, vol. 8, 2008, pp. 1–15.

[23] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The definition of standard ML: revised*. MIT press, 1997.

[24] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[25] National Security Agency, "Ghidra," 2019, [Online; accessed 29-December-2022]. [Online]. Available: https://ghidra-sre.org

[26] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum, "A next-generation platform for analyzing executables," in *Asian Symposium on Programming Languages and Systems*. Springer, 2005, pp. 212–229.

[27] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 159–169.

[28] P. M. Rondon, M. Kawaguchi, and R. Jhala, "Low-level liquid types," *ACM SIGPLAN Notices*, vol. 45, no. 1, pp. 131–144, 2010.

[29] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, "Using logic programming to recover C++ classes and methods from compiled executables," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 426–441.

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[31] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *NDSS*, 2011.

[32] W. Swierstra, "Data types à la carte," *Journal of functional programming*, vol. 18, no. 4, pp. 423–436, 2008.

[33] Vector 35, "Binary Ninja," 2022, [Online; accessed 29-December-2022]. [Online]. Available: https://binary.ninja

[34] G. Wassermann and J. Gennari, "Introducing CERT Kaiju: Malware Analysis Tools for Ghidra," 2021, [Online; accessed 29-December-2022]. [Online]. Available: https://insights.sei.cmu.edu/blog/introducing-cert-kaiju-malware-analysis-tools-for-ghidra/

[35] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "OSPREY: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 813–832.