Mixing Type Checking and Symbolic Execution

Khoo Yit Phang
University of Maryland, College Park
khooyp@cs.umd.edu

Bor-Yuh Evan Chang
University of Colorado, Boulder
bec@cs.colorado.edu

Jeffrey S. Foster
University of Maryland, College Park
jfoster@cs.umd.edu

Abstract

Static analysis designers must carefully balance precision and efficiency. In our experience, many static analysis tools are built around an elegant, core algorithm, but that algorithm is then extensively tweaked to add just enough precision for the coding idioms seen in practice, without sacrificing too much efficiency. There are several downsides to adding precision in this way: the tool's implementation becomes much more complicated; it can be hard for an end-user to interpret the tool's results; and as software systems vary tremendously in their coding styles, it may require significant algorithmic engineering to enhance a tool to perform well in a particular software domain.

In this paper, we present MIX, a novel system that mixes type checking and symbolic execution. The key aspect of our approach is that these analyses are applied independently on disjoint parts of the program, in an off-the-shelf manner. At the boundaries between nested type checked and symbolically executed code regions, we use special mix rules to communicate information between the off-the-shelf systems. The resulting mixture is a provably sound analysis that is more precise than type checking alone and more efficient than exclusive symbolic execution. In addition, we also describe a prototype implementation, MIXY, for C. MIXY checks for potential null dereferences by mixing a null/non-null type qualifier inference system with a symbolic executor.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging—Symbolic execution; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Verification

Keywords Mix, mixed off-the-shelf analysis, symbolic execution, type checking, mix rules, false alarms, precision

1. Introduction

All static analysis designers necessarily make compromises between precision and efficiency. On the one hand, static analysis must be precise enough to prove properties of realistic software systems, and on the other hand, it must run in a reasonable amount of time and space. One manifestation of this trade-off is that, in our experience, many practical static analysis tools begin with a relatively straightforward algorithm at their core, but then gradually accrete a multitude of special cases to add just enough precision without sacrificing efficiency.

Some degree of fine tuning is inevitable—undecidability of static analysis means that analyses must be targeted to programs of interest—but an ad-hoc approach has a number of disadvantages: it significantly complicates the implementation of a static analysis algorithm; it is hard to be sure that all the special cases are handled correctly; and it makes the tool less predictable and understandable for an end-user since the exact analysis algorithm becomes obscured by the special cases. Perhaps most significantly, software systems are extremely diverse, and programming styles vary greatly depending on the application domain and the idiosyncrasies of the programmer and her community's coding standards. Thus an analysis that is carefully tuned to work in one domain may not be effective in another domain.

In this paper, we present MIX, a novel system that trades off precision and efficiency by mixing type checking—a coarse but highly scalable analysis—with symbolic execution [King 1976], which is very precise but inefficient. In MIX, precision versus efficiency is adjusted using *typed blocks* $\{_t\ e\ _t\}$ and *symbolic blocks* $\{_s\ e\ _s\}$ that indicate whether expression e should be analyzed with type checking or symbolic execution, respectively. Blocks may nest arbitrarily to achieve the desired level of precision versus efficiency.

The distinguishing feature of MIX is that its type checking and symbolic execution engines are *completely standard, off-the-shelf implementations*. Within a typed or symbolic block, the analyses run as usual. It is only at the boundary between blocks that we use special *mix rules* to translate information back-and-forth between the two analyses. In this way, MIX gains precision at limited cost, while potentially avoiding many of the pitfalls of more complicated approaches.

As a hypothetical example, consider the following code:

```
1 {s
2   if (multithreaded) {t fork(); t}
3   {t ... t}
4   if (multithreaded) {t lock(); t}
5   {t ... t}
6   if (multithreaded) {t unlock(); t}
7 s}
```

This code uses multiple threads only if multithreaded is set to true. Suppose we have a type-based analysis that checks for data races. Then assuming the analysis is *path-insensitive*, it cannot tell whether a thread is created on line 2, and it does not know the lock state after lines 4 and 6—all of which will lead to false positives.

Rather than add path-sensitivity to our core data race analysis, we can instead use MIX to gain precision. We wrap the program in a symbolic block at the top level so that the executions for each setting of multithreaded will be explored independently. Then for performance, we wrap all the other code (lines 3 and 5 and the calls to fork, lock, and unlock) in typed blocks, so that they are analyzed with the type-based analysis. In this case, these block annotations

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

effectively cause the type-based analysis to be run twice, once for each possible setting of multithreaded; and by separating those two cases, we avoid conflation and eliminate false positives.

While MIX cannot address every precision/efficiency tradeoff issue (for example, the lexical scoping of typed and symbolic blocks is one limitation), there are nonetheless many potential applications. Among other uses, MIX can encode forms of flow-sensitivity, context-sensitivity, path-sensitivity, and local type refinements. MIX can also use type checking to overcome some limitations of symbolic execution (Section 2). Also, for the purposes of this paper, we leave the placement of block annotations to the programmer, but we envision that an automated refinement algorithm could heuristically insert blocks as needed. In this scenario, MIX becomes an intermediate language for modularly combining off-the-shelf analyzer implementations.

In this paper, we formalize MIX for a small imperative language, mixing a standard type checking system with symbolic execution to yield a system to check for the absence of run-time type errors. Thus, rather than checking for assertion failures, as a typical symbolic executor might do, our formal symbolic executor reports any type mismatches it detects. To mix these two systems together, we introduce two new rules: one rule in the type system that "type checks" blocks $\{_s\ e\ _s\}$ using the symbolic executor; and one rule in the symbolic executor that "executes" blocks $\{_t\ e\ _t\}$ using the type checker. We prove that the type system, symbolic executor, and mix of the two systems are sound. The soundness proof of MIX uses the proofs of type soundness and symbolic execution soundness essentially as-is, which provides some additional evidence of a clean modularization. Additionally, two features of our formalism for symbolic execution may be of independent interest: we discuss the tradeoff between "forking" the symbolic executor and giving more work to the solver; and we provide a soundness proof, which, surprisingly, we have been unable to find for previous symbolic execution systems (Section 3).

Finally, we describe MIXY, a prototype implementation of MIX for C. MIXY combines a simple, monomorphic type qualifier inference system (a reimplementation of Foster et al. [2006]) with a C symbolic executor. There are two key challenges that arise when mixing type inference rather than checking: we need to perform a fixed-point computation as we switch between typed and symbolic blocks since data values can pass from one to the other and back; and we need to integrate aliasing information into our analysis so that pointer manipulations performed within symbolic blocks correctly influence typed blocks. Additionally, we extend MIXY to support caching block results as well as recursion between blocks. We use MIXY to look for null pointer errors in a reasonably-sized benchmark `vsftpd`; we found several examples where adding symbolic blocks can eliminate false positives compared to pure type qualifier inference (Section 4).

We believe that MIX provides a promising new approach to trading off precision and efficiency in static analysis. We expect that the ideas behind MIX can be applied to many different combinations of many different analyses.

2. Motivating Examples

Before describing MIX formally, we examine some coding idioms for which type inference and symbolic execution can profitably be mixed. Our examples will be written in either an ML-like language or C-like language, depending on which one is more natural for the particular example.

Path, Flow, and Context Sensitivity. In the introduction, we saw one example in which symbolic execution introduced a small amount of path sensitivity to type inference. There are several potential variations on this example where we can locally add a little bit of path sensitivity to increase the precision of type checking. For example, we can avoid analyzing unreachable code:

$$\{_t\ \ldots \{_s\ \textbf{if true then}\ \{_t\ 5\ _t\}\ \textbf{else}\ \{_t\ \text{"foo"} + 3\ _t\}\ _s\}\ \ldots\ _t\}$$

This code runs without errors, but pure type checking would complain about the potential type error in the false branch. However, with these block annotations added in MIX, the symbolic executor will only invoke the type checker for the true branch and hence will avoid a false positive.

We can also use symbolic execution to gain some flow sensitivity. For example, in a dynamically-typed imperative language, programmers may reuse variables as different types, such as in the following:

$$\{_t\ \ldots \{_s\ \textbf{var}\ \mathsf{x} = 1;\ \{_t\ \ldots\ _t\}\ ;\ \mathsf{x} = \text{"foo"};\ _s\}\ \ldots\ _t\}$$

Here the local variable x is first assigned an integer and is later reused to refer to a string. With the annotations above, we can successfully statically check such code using the symbolic executor to distinguish the two different assignments to x, then type check the code in between.

Similar cases can occur if we try to apply a non-standard type system to existing code. For example, in our case study (Section 4.5), we applied a nullness checker based on type qualifiers to C. We found some examples like the following code:

$$\begin{array}{l}\{_t\ \ldots \{_s\ \mathsf{x}{\rightarrow}\mathsf{obj} = \mathsf{NULL};\\ \qquad\quad \mathsf{x}{\rightarrow}\mathsf{obj} = (\ldots)\mathsf{malloc}(\ldots);\ _s\}\ \ldots\ _t\}\end{array}$$

Here x→obj is initially assigned to NULL, immediately before being assigned a freshly allocated location. A flow-insensitive type qualifier system would think that x→obj could be NULL after this pair of assignments, even though it cannot be.

Finally, we can also use symbolic execution to gain context-sensitivity, though at the cost of duplicate work. For example, in the following:

$$\{_s\ \textbf{let}\ \mathsf{id}\ \mathsf{x} = \mathsf{x}\ \textbf{in}\ \{_t\ \ldots \{_s\ \mathsf{id}\ 3\ _s\}\ \ldots \{_s\ \mathsf{id}\ 3.0\ _s\}\ \ldots\ _t\}\ _s\}$$

the identity function id is called with an **int** and a float. Rather than adding parametric polymorphism to type check this example, we could wrap those calls in symbolic blocks, which in MIX causes the calls to be checked with symbolic execution. While this is likely not useful for standard type checking, for which parametric polymorphism is well-understood, it could be very useful for a more advanced type system for which fully general parametric polymorphic type inference might be difficult to implement or perhaps even undecidable.

A combination of context sensitivity and path sensitivity is possible with MIX. For example, consider the following:

$$\begin{array}{l}\{_s\\ \quad \textbf{let}\ \mathsf{div}\ \mathsf{x}\ \mathsf{y} = \textbf{if}\ \mathsf{y} = 0\ \textbf{then}\ \text{``err''}\ \textbf{else}\ \mathsf{x}\ /\ \mathsf{y}\ \textbf{in}\\ \quad \{_t\ \ldots + \{_s\ \mathsf{div}\ 7\ 4\ _s\}\ _t\}\\ _s\}\end{array}$$

where the div function may return an **int** or a string, but it returns a string (indicating error) only when the second argument is 0. Note that this level of precision would be out of the reach of parametric polymorphism by itself.

Local Refinements of Data. Symbolic execution can also potentially be used to model data more precisely for non-standard type systems. As one example, suppose we introduce a type qualifier system that distinguishes the sign of an integer as either positive, negative, zero, or unknown. Then we can use symbolic execution to refine the type of an integer after a test:

$$\begin{array}{l}\{_t\\ \quad \textbf{let}\ \mathsf{x} : \textbf{unknown int} = \ldots \textbf{in}\\ \quad \{_s\end{array}$$

```
      if x > 0 then {t (* x : pos int *) . . . t}
      else if x = 0 then {t (* x : zero int *) . . . t}
      else {t (* x : neg int *) . . . t}
   s}
t}
```

Here on entry to the symbolic block, x is an unknown integer, so the symbolic executor will assign it an initial symbolic value α_{x} ranging over all possible integers. Then at the conditional branches, the symbolic executor will fork and explore the three possibilities: $\alpha_{\mathsf{x}} > 0$, $\alpha_{\mathsf{x}} = 0$, and $\alpha_{\mathsf{x}} < 0$. On entering the typed block in each branch, since the value of x is constrained in the symbolic execution, the type system will start with the appropriate type for x, either pos, zero, or neg **int**, respectively.

As another example, suppose we have a type system to prevent data races in C. Then a common problem that arises is analyzing local initialization of shared data [Pratikakis et al. 2006]. Consider the following code:

```
{t
   {s
      x = (struct foo *) malloc(sizeof(struct foo));
      x→bar = . . .;
      x→baz = . . .;
      x→qux = . . .;
   s}
   insert(shared_data_structure, x);
t}
```

Here we allocate a new block of memory and then initialize it in several steps before it becomes shared. A flow-insensitive type-based analysis would report an error because the writes through x occur without a lock held. On the other hand, if we wrap the allocation and initialization in a symbolic block, as above, symbolic execution can easily observe that x is local during the initialization phase, and hence the writes need not be protected by a lock.

Helping Symbolic Execution. The previous examples considered adding precision in type checking through symbolic execution. Alternatively, typed blocks can potentially be used to introduce conservative abstraction in symbolic execution when the latter is not viable. For example:

```
{s
  let x = {t unknown_function() t} in . . .
  let y = {t 2**z (* operation not supported by solver *) t} in . . .
  {t while true do {s loop_body s} t}
s}
```

The first line contains a call to a function whose source code is not available, so we cannot symbolically execute the call. However, if we know the called function's type, then we can wrap the call in a typed block (assuming the function has no side effects), conservatively modeling its return value as any possible member of its return type. Similarly, on the second line, we are performing an exponentiation operation, and let us suppose the symbolic executor's solver cannot model this operation if z is symbolic. Then by wrapping the operation in a typed block, we can continue symbolic execution, again conservatively assuming the result of the exponentiation is any member of the result type. The third line shows how we could potentially handle long-running loops by wrapping them in typed blocks, so that symbolic execution would effectively skip over them rather than unroll them (infinitely). We can also recover some precision within the loop body by further wrapping the loop body with a symbolic block.

3. The MIX System

In the previous section, we considered a number of idioms that motivate the design of MIX. Here, we consider a core language, shown in the top portion of Figure 1, with which we study the essence of switching blocks for mixing analyses. Our language includes variables x; integers n; booleans true and false; selected arithmetic and boolean operations $+$, $=$, $\neg$, and $\wedge$; conditionals with if; let bindings; and ML-style updatable references with ref (construction), ! (dereference), and := (assignment). We also include two new block forms, *typed blocks* $\{_{\mathsf{t}}\ e\ _{\mathsf{t}}\}$ and *symbolic blocks* $\{_{\mathsf{s}}\ e\ _{\mathsf{s}}\}$, which indicate e should be analyzed with type checking or symbolic execution, respectively. We leave unspecified whether the outermost scope of a program is treated as a typed block or a symbolic block; MIX can handle either case.

Source Language.

e	$::= x \mid v$	variables, constants
	$\mid\ e + e$	arithmetic
	$\mid\ e = e \mid \neg e \mid e \wedge e$	predicates
	$\mid\ \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$	conditional
	$\mid\ \mathsf{let}\ x = e\ \mathsf{in}\ e$	let-binding
	$\mid\ \mathsf{ref}\ e \mid !e \mid e := e$	references
	$\mid\ \{_{\mathsf{t}}\ e\ _{\mathsf{t}}\}$	type checking block
	$\mid\ \{_{\mathsf{s}}\ e\ _{\mathsf{s}}\}$	symbolic execution block
v	$::= n \mid \mathsf{true} \mid \mathsf{false}$	concrete values

Types, Symbolic Expressions, and Environments.

τ	$::= \mathsf{int} \mid \mathsf{bool} \mid \tau\ \mathsf{ref}$	types
Γ	$::= \emptyset \mid \Gamma, x : \tau$	typing environment
s	$::= u{:}\tau$	typed symbolic expressions
g	$::= u{:}\mathsf{bool}$	guards
u	$::= \alpha \mid v$	symbolic variables, constants
	$\mid\ u{:}\mathsf{int} + u{:}\mathsf{int}$	arithmetic
	$\mid\ s = s \mid \neg g \mid g \wedge g$	predicates
	$\mid\ m[u{:}\tau\ \mathsf{ref}]$	memory select
m	$::= \mu$	arbitrary memory
	$\mid\ m, (s \rightarrow s)$	memory update
	$\mid\ m, (s \xrightarrow{\mathsf{a}} s)$	memory allocation
Σ	$::= \emptyset \mid \Sigma, x : s$	symbolic environment

Figure 1. Program expressions, types, and symbolic expressions.

3.1 Type Checking and Symbolic Execution

Type checking for our source language is entirely standard, and so we omit those rules here. Our type checking system proves judgments of the form $\Gamma \vdash e : \tau$, where Γ is the type environment and τ is e's type. Grammars for Γ and τ are given in the bottom portion of Figure 1.

The remainder of this section describes a generic symbolic executor. While the concept of symbolic execution is widely known, there does not appear to be a clear consensus of its definition. Thus, we make explicit our definition of symbolic execution here through a formalization similar to an operational semantics. Such a formalization enables us to describe the switching between type checking and symbolic execution in a uniform manner.

Symbolic Expressions, Memories, and Environments. The remainder of Figure 1 describes the symbolic expressions and environments used by our symbolic executor. Symbolic expressions are used to accumulate constraints in deferral rules. For example, the symbolic expression $(\alpha{:}\mathsf{int} + 3{:}\mathsf{int}){:}\mathsf{int}$ represents a value that is three more than the unknown integer α.

Because we are concerned with checking for run-time type errors, in our system symbolic expressions s have the form $u{:}\tau$, where u is a bare symbolic expression and τ is its type. With these type annotations, we can immediately determine the type of a symbolic expression, just like in a concrete evaluator with values. As a shorthand, we use g to represent conditional guards, which are just symbolic expressions with type bool. Bare symbolic expressions u may be symbolic variables α (e.g., α:int is a symbolic integer, and α:bool is a symbolic boolean); known values v; or operations $+$, $=$, $\neg$, $\wedge$ applied to symbolic expressions of the appropriate type. Notice that our syntax forbids the formation of certain ill-typed symbolic expression (e.g., $\alpha_1{:}\mathsf{int} + \alpha_2{:}\mathsf{bool}$ is not allowed).

Symbolic expressions also include symbolic memory accesses $m[u{:}\tau\ \mathsf{ref}]$, which represents an access through pointer u in symbolic memory m. A symbolic memory may be μ, representing an arbitrary but well-typed memory; $m,(s \to s')$, a memory that is the same as m except location s is updated to contain s'; or $m,(s \overset{\text{a}}{\to} s')$, which is the same as m except newly allocated location s points to s'. These are essentially McCarthy-style sel and upd expressions that allow the symbolic executor to accumulate a log of writes and allocations while deferring alias analysis. An allocation always creates a new location that is distinct from the locations in the base unknown memory, so we distinguish them from arbitrary writes.

Finally, symbolic environments Σ map local variables x to (typed) symbolic expressions s.

Symbolic Execution for Pure Expressions. Figure 2 describes our symbolic executor on pure expressions using what are essentially big-step operational semantics rules. The rules in Figure 2 prove judgments of the form

$$\Sigma \vdash \langle S\,;e\rangle \Downarrow \langle S'\,;s\rangle$$

meaning with local variables bound in Σ, starting in state S, expression e evaluates to symbolic expression s and updates the state to S'. In our symbolic execution judgment, a *state* S is a tuple $\langle g\,; m\rangle$, where g is a *path condition* constraining the current state and m is the current symbolic memory. The path condition begins as true, and whenever the symbolic executor makes a choice at a conditional, we extend the path condition to remember that choice (more on this below). We write $X(S)$ for the X component of S, with $X \in \{g, m\}$, and similarly we write $S[X \mapsto Y]$ for the state that is the same as S, except its X component is now Y.

Most of the rules in Figure 2 are straightforward and intend to summarize typical symbolic executors. Rule SEVAR evaluates a local variable by looking it up in the current environment. Notice that, as with standard operational semantics, there is no reduction possible if the variable is not in the current environment. Rule SEVAL reduces values to themselves, using the auxiliary function $\mathrm{typeof}(v)$ that examines the value form to return its type (i.e., $\mathrm{typeof}(n) = \mathsf{int}$ and $\mathrm{typeof}(\mathsf{true}) = \mathrm{typeof}(\mathsf{false}) = \mathsf{bool}$).

Rules SEPLUS, SEEQ, SENOT, and SEAND execute the subexpressions and then form a new symbolic expression with $+$, $=$, $\neg$, or $\wedge$, respectively. Notice that these rules place requirements on the subexpressions—for example, SEPLUS requires that the subexpressions reduce to symbolic integers, and SENOT requires that the subexpression reduces to a guard (a symbolic boolean). If the subexpression does not reduce to an expression of the right type, then symbolic execution fails. Thus, these rules form a symbolic execution engine that does very precise dynamic type checking.

Rule SELET symbolically executes e_1 and then binds e_1 to x for execution of e_2. The last two rules, SEIF-TRUE and SEIF-FALSE, model a pure, non-deterministic version of the kind of symbolic execution popularized by DART [Godefroid et al. 2005], CUTE [Sen et al. 2005], EXE [Cadar et al. 2006], and KLEE [Cadar et al. 2008]. When we reach a conditional, we conceptually fork execution, extending the path condition with g_1 or $\neg g_1$ to indicate the branch taken. EXE and KLEE would both invoke an SMT solver at this point to decide whether one or both branches are feasible, and then try all feasible paths. DART and CUTE, in contrast, would continue down one path as guided by an underlying concrete run

Symbolic Execution. $\boxed{\Sigma \vdash \langle S\,;e\rangle \Downarrow \langle S'\,;s\rangle \qquad S = \langle g\,;m\rangle}$

SEVAR
$$\frac{}{\Sigma, x : s \vdash \langle S\,;x\rangle \Downarrow \langle S\,;s\rangle}$$

SEVAL
$$\frac{}{\Sigma \vdash \langle S\,;v\rangle \Downarrow \langle S\,;(v{:}\,\mathrm{typeof}(v))\rangle}$$

SEPLUS
$$\frac{\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;u_1{:}\mathsf{int}\rangle \qquad \Sigma \vdash \langle S_1\,;e_2\rangle \Downarrow \langle S_2\,;u_2{:}\mathsf{int}\rangle}{\Sigma \vdash \langle S\,;e_1 + e_2\rangle \Downarrow \langle S_2\,;(u_1{:}\mathsf{int} + u_2{:}\mathsf{int}){:}\mathsf{int}\rangle}$$

SEEQ
$$\frac{\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;u_1{:}\tau\rangle \qquad \Sigma \vdash \langle S_1\,;e_2\rangle \Downarrow \langle S_2\,;u_2{:}\tau\rangle}{\Sigma \vdash \langle S\,;e_1 = e_2\rangle \Downarrow \langle S_2\,;(u_1{:}\tau = u_2{:}\tau){:}\mathsf{bool}\rangle}$$

SENOT
$$\frac{\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;g_1\rangle}{\Sigma \vdash \langle S\,;\neg e_1\rangle \Downarrow \langle S_1\,;\neg g_1{:}\mathsf{bool}\rangle}$$

SEAND
$$\frac{\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;g_1\rangle \qquad \Sigma \vdash \langle S_1\,;e_2\rangle \Downarrow \langle S_2\,;g_2\rangle}{\Sigma \vdash \langle S\,;e_1 \wedge e_2\rangle \Downarrow \langle S_2\,;(g_1 \wedge g_2){:}\mathsf{bool}\rangle}$$

SELET
$$\frac{\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;s_1\rangle \qquad \Sigma, x : s_1 \vdash \langle S_1\,;e_2\rangle \Downarrow \langle S_2\,;s_2\rangle}{\Sigma \vdash \langle S\,;\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\rangle \Downarrow \langle S_2\,;s_2\rangle}$$

SEIF-TRUE
$$\frac{\begin{array}{c}\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;g_1\rangle \\ \Sigma \vdash \langle S_1[g \mapsto g(S_1) \wedge g_1]\,;e_2\rangle \Downarrow \langle S_2\,;s_2\rangle\end{array}}{\Sigma \vdash \langle S\,;\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \Downarrow \langle S_2\,;s_2\rangle}$$

SEIF-FALSE
$$\frac{\begin{array}{c}\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;g_1\rangle \\ \Sigma \vdash \langle S[g \mapsto g(S_1) \wedge \neg g_1]\,;e_3\rangle \Downarrow \langle S_3\,;s_3\rangle\end{array}}{\Sigma \vdash \langle S\,;\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \Downarrow \langle S_3\,;s_3\rangle}$$

Figure 2. Symbolic execution for pure expressions.

(so-called "concolic execution"), but then would ask an SMT solver later whether the path not taken was feasible and, if so, come back and take it eventually. All of these implementation choices can be viewed as optimizations to prune infeasible paths or hints to focus the exploration. Since we are not concerned with performance in our formalism, we simply extend the path condition and continue—eventually, when symbolic execution completes, we will check the path condition and discard the path if it is infeasible. To get sound symbolic execution, we will compute a set of symbolic executions and require that all feasible paths are explored (see Section 3.2).

Sometimes, the symbolic executor may want to throw away information (e.g., replace a symbolic expression for a complicated memory read with a fresh symbolic variable). Such a rule is straightforward to add, but as discussed in Section 3.2, a nested typed block $\{_t\ e\ _t\}$ serves a similar purpose.

Deferral Versus Execution. Consider again the rules for symbolic execution on pure expressions in Figure 2. Excluding the trivial SEVAL rule, the first set of rules (SEPLUS, SEEQ, SENOT, and SEAND) versus the second set (SELET, SEVAR, SEIF-TRUE, SEIF-FALSE) seem qualitatively different. The first set simply get symbolic expressions for their subexpressions and form a new symbolic expression of the corresponding operator, essentially deferring any reasoning about the operation (e.g., to an SMT solver). In contrast, the second set does not accumulate any such symbolic expression but rather chooses a possible concrete execution to follow. For example, we can view SEIF-TRUE as choosing to assume that g_1 is concretely true and proceeding to symbolically execute e_2. This assumption is recorded in the path condition. (The SELET and SEVAR rules are degenerate execution rules where no assumptions need to be made because there is only one possible concrete execution for each.) Alternatively, we see that there are symbolic expression forms for $+$, $=$, $\neg$, and $\wedge$ but not for let, program variables, and if.

Although it is not commonly presented as such, the decision of deferral versus execution is a design choice. For example, let us include an if-then-else symbolic expression $g?s_1{:}s_2$ (using a C-style conditional syntax) that evaluates to s_1 if g evaluates to true and s_2 otherwise. Then, we could defer to the evaluation of the conditional to the solver with the following rule:

SEIF-DEFER

$$\frac{\begin{array}{c}\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;g_1\rangle \\ \Sigma \vdash \langle S[g \mapsto g(S_1) \wedge g_1]\,;e_2\rangle \Downarrow \langle S_2\,;u_2{:}\tau\rangle \\ \Sigma \vdash \langle S[g \mapsto g(S_1) \wedge \neg g_1]\,;e_3\rangle \Downarrow \langle S_3\,;u_3{:}\tau\rangle \\ S' = \langle (g_1?g(S_2){:}g(S_3))\,;(g_1?m(S_2){:}m(S_3))\rangle\end{array}}{\Sigma \vdash \langle S\,;(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)\rangle \Downarrow \langle S'\,;(g_1?(u_2{:}\tau){:}(u_3{:}\tau)){:}\tau\rangle}$$

Here notice we also have to extend the $\cdot ? \cdot : \cdot$ relation to operate over memory as well. With this rule, we need not "fork" symbolic execution at all. However, note that even with conditional symbolic expressions and condition symbolic memory, this rule is more conservative than the SEIF-TRUE and SEIF-FALSE execution rules, as it requires both branches to have the same type.

Conversely, other rules may also be made non-deterministic in manner similar to SEIF-*. For example, SEVAR may instead return an arbitrary value v and add $\Sigma(x) = v$ to the path condition, a style that resembles hybrid concolic testing [Majumdar and Sen 2007]. A special case of execution rules are ones that apply only when we have concrete values during symbolic execution and thus do not need to "fork." For example, we could have a SEPLUS-CONC that applies to two concrete values n_1, n_2 and returns the sum. This approach is reminiscent of partial evaluation.

These choices trade off the amount of work done between the symbolic executor and the underlying SMT solver. For example, SEIF-DEFER introduces many disjunctions into symbolic expressions, which then may be hard to solve efficiently. To match current practice, we stick with the forking variant for conditionals, but we believe our system would also be sound with SEIF-DEFER.

Symbolic References. Figure 3 continues our symbolic executor definition with rules for updatable references. We use deferral rules for all aspects of references in our formalization. Rule SEREF evaluates e_1 and extends $m(S_1)$ with an allocation for fresh symbolic pointer α. Similarly, rule SEASSIGN extends S_2 to record that s_1 now points to s_2. Observe that allocations and writes are simply logged during symbolic execution for later inspection. Also, notice that we allow *any* value to be written to s_1, even if it does not match the type annotation on s_1. In contrast, standard type systems require that any writes to memory must preserve types since the type system does not track enough information about pointers to be sound if that property is violated. Symbolic execution tracks every possible program execution precisely, and so it can allow arbitrary memory writes.

In SEDEREF, we evaluate e_1 to a pointer $u_1{:}\tau$ ref and then produce the symbolic expression $m(S_1)[u_1{:}\tau\ \mathsf{ref}]{:}\tau$ to represent the contents of that location. However, here we are faced with a challenge: we are not actually looking up the contents of memory; rather, we are simply forming a symbolic expression to represent

Symbolic Execution for References. $\boxed{\Sigma \vdash \langle S\,;e\rangle \Downarrow \langle S'\,;s\rangle}$

SEREF

$$\frac{\begin{array}{c}\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;u_1{:}\tau\rangle \qquad \alpha \notin \Sigma, S, S_1, u_1 \\ S' = S_1[m \mapsto (m(S_1), (\alpha{:}\tau\ \mathsf{ref} \xrightarrow{\text{a}} u_1{:}\tau))]\end{array}}{\Sigma \vdash \langle S_1\,;\mathsf{ref}\ e_1\rangle \Downarrow \langle S'\,;\alpha{:}\tau\ \mathsf{ref}\rangle}$$

SEASSIGN

$$\frac{\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;s_1\rangle \qquad \Sigma \vdash \langle S_1\,;e_2\rangle \Downarrow \langle S_2\,;s_2\rangle}{\Sigma \vdash \langle S\,;e_1 := e_2\rangle \Downarrow \langle S_2[m \mapsto (m(S_2), (s_1 \to s_2))]\,;s_2\rangle}$$

SEDEREF

$$\frac{\Sigma \vdash \langle S\,;e_1\rangle \Downarrow \langle S_1\,;u_1{:}\tau\ \mathsf{ref}\rangle \qquad \vdash m(S_1)\ \mathsf{ok}}{\Sigma \vdash \langle S\,;!e_1\rangle \Downarrow \langle S_1\,;m(S_1)[u_1{:}\tau\ \mathsf{ref}]{:}\tau\rangle}$$

Memory Type Consistency. $\boxed{\vdash m\ \mathsf{ok}\ U \qquad \vdash m\ \mathsf{ok}}$

EMPTY-OK

$$\frac{}{\vdash \mu\ \mathsf{ok}\ \emptyset}$$

ALLOC-OK

$$\frac{\vdash m\ \mathsf{ok}\ U}{\vdash m, (\alpha{:}\tau\ \mathsf{ref} \xrightarrow{\text{a}} u_2{:}\tau)\ \mathsf{ok}\ U}$$

OVERWRITE-OK

$$\frac{\vdash m\ \mathsf{ok}\ U \quad U' = U \setminus \{s_1 \to s_2 \mid s_1 \equiv u_1{:}\tau\ \mathsf{ref} \wedge s_1 \to s_2 \in U\}}{\vdash m, (u_1{:}\tau\ \mathsf{ref} \to u_2{:}\tau)\ \mathsf{ok}\ U'}$$

ARBITRARY-NOTOK

$$\frac{\vdash m\ \mathsf{ok}\ U}{\vdash m, (s_1 \to s_2)\ \mathsf{ok}\ (U \cup \{s_1 \to s_2\})}$$

M-OK

$$\frac{\vdash m\ \mathsf{ok}\ \emptyset}{\vdash m\ \mathsf{ok}}$$

Figure 3. Symbolic execution for updatable references.

the contents. How, then, do we determine the type of the pointed-to value? We need the type so that we can halt symbolic execution later if that value is used in a type-incorrect manner. That is, we do not want to defer the discovery of a potential type error.

Our solution is to use the type annotation on the pointer to get the type of the contents—but above we just explained that SEASSIGN allows writes to violate those type annotations. There are many potential ways to solve this problem. We could invoke an SMT solver to compute the actual set of addresses that could be dereferenced and fork execution for each one. Or we could proceed as our implementation and use an external alias analysis to conservatively model all possible locations that could be read to check that the values at all locations have the same type (Section 4). However, to keep the formal system simple, we choose a very coarse solution: we require that *all* pointers in memory are well-typed with the check $\vdash m(S_1)$ ok.

This judgment is defined in the bottom portion of Figure 3 in terms of the auxiliary judgment $\vdash m$ ok U, which means memory m is consistently typed (pointers point to values of the right type), except for mappings in U. There are four cases for this judgment. EMPTY-OK says that arbitrary well-typed memory μ is consistently typed. Similarly, ALLOC-OK says that if m is consistently typed except for potentially inconsistent writes in U, then adding an allocation preserves consistent typing up to U. Rule OVERWRITE-OK says that if $\vdash m$ ok U and we extend m with a well-typed write to u_1, then any previous, inconsistent writes to locations $s_1 \equiv u_1{:}\tau$ ref can be ignored. Here by $\equiv$ we mean syntactic equivalence, but in practice we could query a solver to validate such an equality given the current path condition. Rule ARBITRARY-NOTOK says that any write can be added to U and viewed as potentially inconsistent. Finally, M-OK says that $\vdash m$ ok if m has no

Block Typing. $\boxed{\Gamma \vdash e : \tau}$

$$\textsc{TSymBlock}\quad \frac{\begin{array}{c}\Sigma(x) = \alpha_x{:}\Gamma(x) \qquad (\text{for all } x \in \mathit{dom}(\Gamma)) \\ \Sigma \vdash \langle S\, ; e\rangle \Downarrow \langle S_i\, ; u_i{:}\tau\rangle \qquad S = \langle \mathsf{true}\, ; \mu\rangle \qquad \mu \notin \Sigma \\ \vdash m(S_i)\ \mathsf{ok} \qquad \mathit{exhaustive}(g(S_1), \ldots, g(S_n)) \qquad (i \in 1..n)\end{array}}{\Gamma \vdash \{_{\mathsf{s}}\ e\ _{\mathsf{s}}\} : \tau}$$

$$\mathit{exhaustive}(g_1, \ldots, g_n) \iff (g_1 \vee \ldots \vee g_n \text{ is a tautology})$$

Block Symbolic Execution. $\boxed{\Sigma \vdash \langle S\, ; e\rangle \Downarrow \langle S'\, ; s\rangle}$

$$\textsc{SETypBlock}\quad \frac{\vdash \Sigma : \Gamma \qquad \vdash m(S)\ \mathsf{ok} \qquad \Gamma \vdash e : \tau \qquad \mu', \alpha \notin \Sigma, S}{\Sigma \vdash \langle S\, ; \{_{\mathsf{t}}\ e\ _{\mathsf{t}}\}\rangle \Downarrow \langle S[m \mapsto \mu']\, ; \alpha{:}\tau\rangle}$$

Symbolic and Typing Environment Conformance. $\boxed{\vdash \Sigma : \Gamma}$

$$\frac{\begin{array}{c}\mathit{dom}(\Sigma) = \mathit{dom}(\Gamma) \\ \Sigma(x) = u{:}\Gamma(x) \qquad (\text{for all } x \in \mathit{dom}(\Gamma))\end{array}}{\vdash \Sigma : \Gamma}$$

Figure 4. Mixing symbolic execution and type checking.

inconsistent writes that persist. Together, these rules ensure that the type assigned to the result of a dereference is sound. We can also see how the SEDEREF may be made more precise by only requiring consistency up to a set of writes U and querying a solver to show that $u_1{:}\tau$ ref are disequal to all the address expressions in U.

3.2 Mixing

In the previous section, we considered type checking and symbolic execution separately, ignoring the blocks that indicate a switch in analysis. Figure 4 shows the two *mix rules* that capture switching between analyses.

Rule TSYMBLOCK describes how to type check a symbolic block $\{_{\mathsf{s}}\ e\ _{\mathsf{s}}\}$, that is, how to apply symbolic execution to derive a type of a subexpression for a type checker. First, we construct an environment Σ that maps each variable x in Γ to a fresh symbolic variable α_x, whose type is extracted from Γ. Then we run the symbolic execution under Σ, starting in a state with true for the path condition and a fresh symbolic variable μ to stand for the current memory. Recall that, because of SEIF-TRUE and SEIF-FALSE, symbolic execution is actually non-deterministic—it conceptually can branch at conditionals. If we want to *soundly* model the entire possible behavior of e, we need to execute all paths. Thus, we run the symbolic executor n times, yielding final states $\langle S_i\, ; u_i{:}\tau\rangle$ for $i \in 1..n$, and we require that the disjunction of the guards from all executions form a tautology. This constraint ensures that we exhaustively explore every possible path (see Section 3.3 about soundness). And if all those paths executed successfully without type errors and returned a value of the same type τ, then that is the type of expression e. We also check that all paths leave memory in a consistent state.

Symbolic execution has typically been used as an unsound analysis where there is no exhaustiveness check like $\mathit{exhaustive}(\ldots)$ in the TSYMBLOCK. We can also model such unsound analysis by weakening $\mathit{exhaustive}(\ldots)$ to a "good enough check."

The other rule, SETYPBLOCK, describes how to symbolically execute a typed block $\{_{\mathsf{t}}\ e\ _{\mathsf{t}}\}$, that is, how to apply the type checker in the middle of a symbolic execution. We begin by deriving a type environment Γ that maps local variables to the types of the symbols they are mapped to in Σ. This mapping is described precisely by the judgment $\vdash \Sigma : \Gamma$, which is straightforward. We also require that the current symbolic memory state be consistent, since the typed block relies purely on type information (rather than tracking pointer values as symbolic execution does). Then we type check e in Γ, yielding a type τ. The typed block itself symbolically evaluates to a fresh symbolic variable α of type τ. Since the typed block may have written to memory, we conservatively set the memory of the output state to a fresh μ', indicating we know nothing about the memory state at that point except that it is consistent.

Note that in our formalism, we do not have typed blocks within typed blocks, or symbolic blocks within symbolic blocks, though these would be trivial to add (by passing-through).

Why Mix? The mix rules are essentially as precise as possible given the strengths and limitations of each analysis. The nested analysis starts with the maximum amount of information that can be extracted from the other static analysis—for symbolic blocks, the only available information for symbolic execution is types, whereas for typed blocks, the type checker only cares about types of variables and thus abstracts away the symbolic expressions. After the nested analysis is complete, the result is similarly passed back to the enclosing analysis as precisely as possible.

For this paper, we deliberately chose two analyses at opposite ends of the precision spectrum: type checking is cheap, flow-insensitive with a rather coarse abstraction, while symbolic execution is expensive, flow- and path-sensitive (and context-sensitive if we add functions) with a minimal amount of abstraction (i.e., it is not even a proper program analysis per se, as there are no termination guarantees). They also work in such a different manner that it does not seem particularly natural to combine them in tighter ways (e.g., as a reduced product of abstract interpreters [Cousot and Cousot 1979]). We think it is surprising just how much additional precision we can obtain and the kinds of idioms we can analyze from such a simple mixing of an entirely standard type system and a typical symbolic executor as-is (as we see in Section 2). We note that a type system capturing all of the examples in Section 2 would likely be quite advanced (involving, for example, dependent types).

However, as can be seen in Figure 4, the conversion between these two analyses may be extremely lossy. For example, in SETYPBLOCK, the memory after returning from the type checker must be a fresh arbitrary memory μ' because e may make any number of writes not captured by the type system and thus not seen by the symbolic executor. We can also imagine mixing any number of analyses in arbitrary combination, yielding different precision/efficiency tradeoffs. For example, if we were to use a type and effect system rather than just a type system, we could avoid introducing a completely fresh memory μ' in SETYPBLOCK—instead, we could find the effect of e and limit applying this "havoc" operation only to locations that could have been changed.

3.3 Soundness

In this section, we sketch the soundness of MIX, which is described in full detail in the appendix of our companion technical report [Khoo et al. 2010]. The key feature of our proof is that aside from the mix rule cases, it reuses the standalone type soundness and symbolic execution soundness proofs essentially as-is.

We show soundness with respect to a standard big-step operational semantics for our simple language of expressions. Our semantics is given by a judgment $E \vdash \langle M; e\rangle \to r$. This says that in a concrete environment E, an initial concrete memory M and an expression e evaluate to a result r. A concrete environment maps variables to values, while a concrete memory maps locations to values. The evaluation result r is either a concrete memory-value pair $\langle M'; v\rangle$ or a distinguished error token.

To prove mix soundness, we consider simultaneously type and symbolic execution soundness. While type soundness is standard, we discuss it briefly, as it is a part of mix soundness, and provides intuition for symbolic execution soundness.

For type soundness, we introduce a memory type environment Λ that maps locations to types, and we update the typing judgment to carry this additional environment, as $\Gamma \vdash_\Lambda e : \tau$ where Λ is constant in all rules. In many proofs, Λ is included in Γ rather than being called out separately, but for Mix soundness separating locations from variables makes the proof easier. To show type soundness, we need a relation between the concrete environment and memory $\langle E; M\rangle$ and the type environment and memory typing $\langle \Gamma; \Lambda\rangle$. We write this relation as $\langle E; M\rangle \sim \langle \Gamma; \Lambda\rangle$, which informally says two things: (1) the type environment Γ abstracts the concrete environment E, that is, the concrete value v mapped by each variable x in E has type $\Gamma(x)$, and (2) the memory typing Λ abstracts the concrete memory M, that is, the concrete value v at each location l in M has type $\Lambda(l)$. We also talk about the second component in isolation, in which case we write $M \sim \Lambda$ to mean memory typing Λ abstracts the concrete memory M.

Type soundness is the first part of mix soundness (statement 1 in Theorem 1, shown below). Let us consider the pieces. Suppose we have a concrete evaluation $E \vdash \langle M; e\rangle \rightarrow r$. We further suppose that e has type τ in typing environments that are sound with respect to the concrete state (i.e., $\langle E; M\rangle \sim \langle \Gamma; \Lambda\rangle$). Then, the result r must be a memory-value pair $\langle M'; v\rangle$ where the resulting concrete memory is abstracted by Λ', an extension of Λ, and the resulting value v has the same type τ in Γ with the extended memory typing Λ'. Notice this captures the notions that well-typed expressions cannot evaluate to error and that evaluation preserves typing.

For symbolic execution soundness, we need to ensure that a symbolic execution faithfully models actual concrete executions. Let V be a *valuation*, which is a finite mapping from symbolic values α to concrete values v or concrete memories M. We write $[\![s]\!]^V$, $[\![m]\!]^V$, and $[\![\Sigma]\!]^V$ for the natural extension of V to operate on arbitrary symbolic expressions, memories, and the symbolic environment. Symbolic execution begins with symbolic values α for unknown inputs and accumulates a symbolic expression s that represents the result of the program. Then at a high-level, if symbolic execution is sound, then a concrete run that begins with $[\![\alpha]\!]^V$ for inputs should produce the expression $[\![s]\!]^V$.

To formalize this notion, we need a soundness relation between the concrete evaluation state and the symbolic execution state, just as in type soundness. The form of our soundness relations for symbolic execution states is as follows:

$$\langle E; M\rangle \sim_{\Lambda_0 \cdot V \cdot \Lambda} \langle \Sigma; m\rangle$$

This relation captures two key properties. First, applying the valuation V to the symbolic state should yield the concrete state (i.e., $[\![\Sigma]\!]^V = E$ and $[\![m]\!]^V = M$). Second, the types of symbolic expressions in Σ and m must be correctly related. Recall that an additional property of our typed symbolic execution is that it tracks the type of symbolic expressions and halts upon encountering ill-typed expressions. The typing of symbolic reference expressions must be with respect to some memory typing. This memory typing is given by Λ_0 and Λ. For technical reasons, we need to separate the locations in the arbitrary memory on entry Λ_0 from the locations that come from allocations during symbolic execution Λ; to get typing for the entire memory, we write $\Lambda_0 * \Lambda$ to mean the union of submemory typings Λ_0 and Λ with disjoint domains. Analogously, we also have a symbolic soundness relation that applies to memory-value pairs: $\langle M; v\rangle \sim_{\Lambda_0 \cdot V \cdot \Lambda} \langle m; s\rangle$.

As alluded to above, we first consider a notion of symbolic execution soundness with respect to a concrete execution. This notion is what is stated in the second part of mix soundness (Theorem 1). Analogous to type soundness, it says that suppose we have a concrete evaluation $E \vdash \langle M; e\rangle \rightarrow r$ and a symbolic execution $\Sigma \vdash \langle S; e\rangle \Downarrow \langle S'; s\rangle$ such that the symbolic state is an abstraction of the concrete state (i.e., $\langle E; M\rangle \sim_{\Lambda_0 \cdot V \cdot \Lambda} \langle \Sigma; m(S)\rangle$). There is one more premise, $[\![g(S')]\!]^V$, which says that the path condition accumulated during symbolic execution holds under this valuation. This constrains the concrete and symbolic executions to follow the same path. With these premises, symbolic execution soundness says that the result of symbolic execution, that is the memory-symbolic expression pair $\langle m(S'); s\rangle$, is an abstraction of the concrete evaluation result, which must be a memory-value pair $\langle M'; v\rangle$.

Theorem 1 (MIX Soundness)

1. If

$$E \vdash \langle M; e\rangle \rightarrow r \quad \textit{and}$$
$$\Gamma \vdash_\Lambda e : \tau \quad \textit{such that}$$
$$\langle E; M\rangle \sim \langle \Gamma; \Lambda\rangle \quad ,$$

then $\emptyset \vdash_{\Lambda'} v : \tau$ *and* $M' \sim \Lambda'$ *for some* M', v, Λ' *such that* $r = \langle M'; v\rangle$ *and* $\Lambda' \supseteq \Lambda$.

2. If

$$E \vdash \langle M; e\rangle \rightarrow r \quad \textit{and}$$
$$\Sigma \vdash \langle S\, ;\, e\rangle \Downarrow \langle S'\, ;\, s\rangle \quad \textit{such that}$$
$$\langle E; M\rangle \sim_{\Lambda_0 \cdot V \cdot \Lambda} \langle \Sigma; m(S)\rangle \quad \textit{and} \quad [\![g(S')]\!]^V \quad ,$$

then $r \sim_{\Lambda_0' \cdot V' \cdot \Lambda'} \langle m(S'); s\rangle$ *for some* $V' \supseteq V$ *and some* Λ_0', Λ' *such that* $\Lambda_0' * \Lambda' \supseteq \Lambda_0 * \Lambda$.

PROOF

By simultaneous induction on the derivations of $E \vdash \langle M; e\rangle \rightarrow r$. The proof is given in the appendix of our companion technical report [Khoo et al. 2010].

This statement of symbolic execution soundness (part 2 in Theorem 1) is what we need to show MIX sound, but at first glance, it seems suspect because it does not say anything about symbolic execution being exhaustive. However, if we look at type checking a symbolic block (i.e., rule TSYMBLOCK), exhaustiveness is ensured through the $\textit{exhaustive}(\ldots)$ constraint.

In particular, we can state exhaustive symbolic execution as a corollary, and the case for TSYMBLOCK proceeds in the same manner as this corollary.

Corollary 1.1 (Exhaustive Symbolic Execution)

Suppose $E \vdash \langle M; e\rangle \rightarrow \langle M'; v\rangle$ *and we have* $n > 0$ *symbolic executions*

$$\Sigma \vdash \langle\langle \mathsf{true}; m\rangle\, ;\, e\rangle \Downarrow \langle S_i\, ;\, s_i\rangle \quad \textit{such that}$$
$$\textit{exhaustive}(g(S_1), \ldots, g(S_n)) \quad \textit{and}$$
$$\langle E; M\rangle \sim_{\Lambda_0 \cdot V \cdot \Lambda} \langle \Sigma; m\rangle \quad ,$$

then $\langle M'; v\rangle \sim_{\Lambda_0' \cdot V' \cdot \Lambda'} \langle m(S_i); s_i\rangle$ *for some* $i \in 1..n$, $V' \supseteq V$, *and some* Λ_0', Λ' *such that* $\Lambda_0' * \Lambda' \supseteq \Lambda_0 * \Lambda$.

Here we say that if we have $n > 0$ symbolic executions that each start with a path condition of true and where their resulting path conditions are *exhaustive* (i.e., $g(S_1) \vee \ldots \vee g(S_n)$ is a tautology meaning it holds under any valuation V), then one of those symbolic executions must match the concrete execution. Observe that in this statement, there is no premise on the resulting path condition, but rather that we start with a initial path condition of true.

4. MIXY: A Prototype of MIX for C

We have developed MIXY, a prototype tool for C that uses MIX to detect null pointer errors. MIXY mixes a (flow-insensitive) type qualifier inference system with a symbolic executor. MIXY is built on top of the CIL front-end for C [Necula et al. 2002], and our type qualifier inference system, CilQual, is essentially a simplified CIL reimplementation of the type qualifier inference algorithm described by Foster et al. [2006]. Our symbolic executor, Otter [Reisner et al. 2010], uses STP [Ganesh and Dill 2007] as its SMT solver and works in a manner similar to KLEE [Cadar et al. 2008].

Type Qualifiers and Null Pointer Errors. For this application, we introduce two qualifier annotations for pointers: nonnull indicates that a pointer must not be null, and null indicates that a pointer may be null. Our inference system automatically annotates uses of the NULL macro with the null qualifier annotation. The type qualifier inference system generates constraints among known qualifiers and unknown qualifier variables, solves those constraints, and then reports a warning if null values may flow to nonnull positions. Thus, our type qualifier inference system ensures pointers that may be null cannot be used where non-null pointers are required.

For example, consider the following C code:

```
1 void free(int *nonnull x);
2 int *id(int *p) { return p; }
3 int *x = NULL;
4 int *y = id(x);
5 free(y);
```

Here on line 1 we annotate free to indicate it takes a nonnull pointer. Then on line 3, we initialize x to be NULL, pass that value through id, and store the result in y on line 4. Then on line 5 we call free with NULL.

Our qualifier inference system will generate the following types and constraints (with some simplifications, and ignoring l- and r-value issues):

$$\begin{array}{rl} \mathsf{free} : \mathsf{int} * \mathsf{nonnull} \to \mathsf{void} & \mathsf{x} : \mathsf{int} * \beta \\ \mathsf{id} : \mathsf{int} * \gamma \to \mathsf{int} * \delta & \mathsf{y} : \mathsf{int} * \epsilon \end{array}$$

$$\mathsf{null} = \beta \quad \beta = \gamma \quad \gamma = \delta \quad \delta = \epsilon \quad \epsilon = \mathsf{nonnull}$$

Here β, γ, δ, and ϵ are variables that standard for unknown qualifiers. Put together, these constraints require null = nonnull, which is not allowed, and hence qualifier inference will report an error for this program.

Our symbolic executor also looks for null pointer errors. The symbolic executor tracks C values at the bit level, using a representation similar to KLEE [Cadar et al. 2008]. A null pointer is represented as the value 0, and the symbolic executor reports an error if 0 is ever dereferenced.

Typed and Symbolic Blocks. In our formal system, we allow typed and symbolic blocks to be introduced anywhere in the program. In MIXY, these blocks can only be introduced around whole function bodies by annotating a function as *MIX(typed)* or *MIX(symbolic)*, and MIXY switches between qualifier inference and symbolic execution at function calls. We can simulate blocks within functions by manually extracting the relevant code into a fresh function.

Skipping some details for the moment, this switching process works as follows. When MIXY is invoked, the programmer specifies (as a command-line option) whether to begin in a typed block or a symbolic block. In either case, we first initialize global variables as appropriate for the analysis, and then analyze the program starting with main. In symbolic execution mode, we begin simulating the program at the entry function, and at calls to functions that are either unmarked or are marked as symbolic, we continue symbolic execution into the function body. At calls to functions marked with *MIX(typed)*, we switch to type inference starting with that function.

In type inference mode, we begin analysis at the entry function f, applying qualifier inference to f and all functions reachable from f in the call graph, up to the frontier of any functions that are marked with *MIX(symbolic)*. We use CIL's built-in pointer analysis to find the targets of calls through function pointers. Finally, we switch to symbolic execution for each function marked *MIX(symbolic)* that was discovered at the frontier.

In this section, we describe implementation details that are not captured by our formal system from Section 3:

- The formal system MIX is based on a type checking system where all types are given. Since type qualifier inference involves variables, we need to handle variables that are not yet constrained to concrete type qualifiers when transitioning to a symbolic block (Section 4.1).
- We need to translate information about aliasing between blocks (Section 4.2).
- Since the same block or function may be called from multiple contexts, we need to avoid repeating analysis of the same function (Section 4.3).
- Since functions can contain blocks and be recursive, we need to handle recursion between typed and symbolic blocks (Section 4.4).

Finally, we present our initial experience with MIXY (Section 4.5), and we discuss some limitations and future work (Section 4.6).

4.1 Translating Null/Non-null and Type Variables

At transitions between typed and symbolic blocks, we need to translate null and nonnull annotations back and forth.

From Types to Symbolic Values. Suppose local variable x has type **int** ***nonnull**. Then in the symbolic executor, we initialize x to point to a fresh memory cell. If x has type **int** ***null**, then we initialize x to be $(\alpha{:}\mathsf{bool})?loc{:}0$, where α is a fresh boolean that may be either true or false, loc is a newly initialized pointer (described in Section 4.2), and 0 represents null. Hence this expression means x may be either null or non-null, and the symbolic executor will try both possibilities.

A more interesting case occurs if a variable x has a type with a qualifier variable (e.g., **int** $*\beta$). In this case, we first try to solve the current set of constraints to see whether β has a solution as either null or nonnull, and if it does, we perform the translation given above. Otherwise, if β could be either, we first optimistically assume it is nonnull.

We can safely use this assumption when returning from a typed block to a symbolic block since such a qualifier variable can only be introduced when variables are aliased (e.g., via pointer assignment), a case that is separately taken into account by the MIXY memory model (Section 4.2).

However, if we use this assumption when entering a symbolic block from a typed block, we may later discover our assumption was too optimistic. For example, consider the following code:

$$\{_t\ \textbf{int}\ {*}\mathsf{x};\ \{_s\ \mathsf{x} = \mathsf{NULL};\ _s\}\ ;\ \{_s\ \mathsf{free}(\mathsf{x});\ _s\}\ _t\}$$

In the type system, x has type **int** $*\ \beta$, where initially β is unconstrained. Suppose that we analyze the symbolic block on the right before the one on the left. This scenario could happen because the analysis of the enclosing typed block does not model control-flow order (i.e., is flow-insensitive). Then initially, we would think the call to free was safe because we optimistically treat unconstrained β as nonnull—but this is clearly not accurate here.

The solution is, as expected, to repeat our analyses until we reach a fixed point. In this case, after we analyze the left symbolic block, we will discover a new constraint on x, and hence when we iterate and reanalyze the right symbolic block, we will discover the error. We are computing a least fixed point because we start with optimistic assumptions—nothing is null—and then monotonically discover more expressions may be null.

From Symbolic Values to Types. We use the SMT solver to discover the possible final values of variables and translate those to the appropriate types. Given a variable x that is mapped to symbolic expression s, we ask whether $g \wedge (s = 0)$ is satisfiable where g is the path condition. If the condition is satisfiable, we constrain x to be null in the type system. There are no nonnull constraints to be added since they correspond to places in code where pointers are dereferenced, which is not reflected in symbolic values.

Thus, null pointers from symbolic blocks will lead to errors in typed blocks if they flow to a nonnull position; whereas null pointers from typed blocks will lead to errors in symbolic blocks if they are dereferenced symbolically.

4.2 Aliasing and MIXY's Memory Model

The formal system MIX defers all reasoning about aliasing to as late of a time as possible. As alluded to in Section 3, this choice may be difficult to implement in practice given limitations in the constraint solver. Thus in MIXY, we use a pre-pass pointer analysis to initialize aliasing relationships.

Typed to Symbolic Block. When we switch from a typed block to a symbolic block, we initialize a fresh symbolic memory, which may include pointers. We use a variant of the approach described in Section 3 that makes use of aliasing information to be more precise. Rather than modeling memory as one big array, MIXY models memory as a map from locations to separate arrays. Aliasing within arrays is modeled as in our formalism, and aliasing between arrays is modeled using Morris's general axiom of assignment [Bornat 2000; Morris 1982].

C also supports a richer variety of types such as arrays and structs, as well as recursive data structures. MIXY lazily initializes memory in an incremental manner so that we can sidestep the issue of initializing an arbitrarily recursive data structure; MIXY only initializes as much as is required by the symbolic block. We use CIL's pointer analysis to determine possible points-to relationships and initialize memory accordingly.

Symbolic to Typed Block. An issue arises from using type inference when we switch from a symbolic block to a typed block. Consider the following code snippets, which are identical except that y points to r on the left, and y points to x on the right:

```
{s                                  {s
  // *y not aliased to x              // *y aliased to x
  int *x = ...;                       int *x = ...;
  int *r = ..., **y = &r;             int **y = &x;
  {t // okay                          {t // should fail
    x = NULL;                           x = NULL;
    assert_nonnull(*y); t}              assert_nonnull(*y); t}
s}                                  s}
```

In both cases, at entry to the typed blocks, x and $*$y are assigned types β **ref** and γ **ref** respectively, based on their current values. Notice, however, that for the code on the right, we should also have $\beta = \gamma$. Otherwise, after the assignment x = NULL, we will not know that $*$y is also NULL.

This example illustrates an important difference between type inference and type checking. In type checking, this problem cannot arise because every value has a known type, and we only have to check that those types are consistent. However, type inference actually has to discover richer information, such as what types must be equal because of aliasing, in order to find a valid typing.

One solution to this problem would be to translate aliasing information from symbolic execution to and from type constraints. In MIXY, we use an alternative solution that is easier to implement: we use CIL's built-in may pointer analysis to conservatively discover points-to relationships. When we transition from a symbolic block to a typed block, we add constraints to require that all may-aliased expressions have the same type.

4.3 Caching Blocks

In C, a block or function may be called from many different call sites, so we may need to analyze that block in the context of each call site. Since it can be quite costly to analyze that block repeatedly, we cache the calling context and the results of the analysis for that block, and we reuse the results when the block is called again with a compatible calling context. Conceptually, caching is similar to compositional symbolic execution [Godefroid 2007]; in MIXY, we implement caching as an extension to the mix rules, using types to summarize blocks rather than symbolic constraints.

Caching Symbolic Blocks. Before we translate the types from the enclosing typed block to symbolic values, we first check to see if we have previously analyzed the same symbolic block with a compatible calling context. We define the calling context to be the types for all variables that will be translated into symbolic values, and we say two calling contexts are compatible if every variable has the same type in both contexts.

If we have not analyzed the symbolic block before with a compatible calling context, we translate the types into symbolic values, analyze the symbolic block, and translate the symbolic values to types by adding type constraints as usual. At this point, we will cache the translated types for this calling context; we cache the translated types instead of the symbolic values since the translation from symbolic values to types is expensive. Otherwise, if we have analyzed the symbolic block before with a compatible calling context, we use the cached results by adding null type constraints for null cached types in a manner similar to translating symbolic values. Finally, in both cached and uncached cases, we restore aliasing relationships and return to the enclosing typed block as usual.

Caching Typed Blocks. Caching for typed blocks is similarly implemented, but with one difference: unlike above, we first translate symbolic values into types, then use the translated types as the calling context, and finally cache the final types as the result of analyzing the typed block. We could have chosen to use symbolic values as the calling context and the result, but since translating symbolic values to types or comparing symbolic values both involve similar number of calls to the SMT solver, we chose to use types to unify the implementation.

4.4 Recursion between Typed and Symbolic Blocks

A typed block and a symbolic block may recursively call each other, and we found block recursion to be surprisingly common in our experiments. Without special handling for recursion, MIXY will keep switching between them indefinitely since a block is analyzed with a fresh initial state upon every entry. Therefore, we need to detect when recursion occurs, either beginning with a typed block or a symbolic block, and handle it specially.

To handle recursion, we maintain a block stack to keep track of blocks that are currently being analyzed. Similar to a function call stack, the block stack is a stack of blocks and their calling contexts, which are defined in terms of types as in caching (Section 4.3). We push blocks onto the stack upon entry and pop them upon return.

Before entering a block, we first look for recursion by searching the block stack for the same block with a compatible calling context. If recursion is detected, then instead of entering the block, we mark the matching block on the stack as recursive and return an assumption about the result. For the initial assumption, we use the calling context of the marked block, optimistically assuming that the block has no effect. When we eventually return to the marked block, we compare the assumption with the actual result of analyzing the block. If the assumption is compatible with the actual result, we return the result; otherwise, we re-analyze the block using the actual result as the updated assumption until we reach a fixed point.

4.5 Preliminary Experience

We gained some initial experience with MIXY by running it on `vsftpd-2.0.7` and looking for false null pointer warnings from pure type qualifier inference that can be eliminated with the addition of symbolic execution. Since MIXY is in the prototype stage, we started small. Rather than annotate all dereferences as requiring nonnull, we added just one nonnull annotation:

```
sysutil_free(void * nonnull p_ptr) MIX(typed) { ... }
```

The sysutil_free function wraps the free system call and checks, at run time, that the pointer argument is not null. In essence, our analysis tries to check this property statically. We annotated sysutil_free itself with *MIX(typed)*, so MIXY need not symbolically execute its body—our annotation captures the important part of its behavior for our analysis.

We then ran MIXY on `vsftpd`, beginning with typing at the outermost level. We examined the resulting warnings and then tried adding *MIX(symbolic)* annotations to eliminate warnings. We succeeded in several cases, discussed next. We did not fully examine many of the other cases, but Section 4.6 describes some preliminary observations about MIXY in practice. Note that the code snippets shown below are abbreviated, and many identifiers have been shortened. We should also point out that all the examples below eliminate one or more imprecise qualifier flows from type qualifier inference; this pruning may or may not suppress a given warning, depending on whether other flows could produce the same warning.

***Case 1: Flow and path insensitivity in* sockaddr_clear**

```
1 void sockaddr_clear(struct sockaddr **p_sock) MIX(symbolic) {
2   if (*p_sock != NULL) {
3     sysutil_free(*p_sock);
4     *p_sock = NULL;
5   }
6 }
```

This function is implicated in a false warning: due to flow insensitivity in the type system, the null assignment on line 4 flows to the argument to sysutil_free on line 3, even though the assignment occurs after the call. Also, the type system ignores the null check on line 2 due to path-insensitivity.

Marking sockaddr_clear with *MIX(symbolic)* successfully resolves this warning: the symbolic executor determines that *p_sock is not null when used as an argument to sysutil_free().

***Case 2: Path and context insensitivity in* str_next_dirent**

```
1 void str_alloc_text(struct mystr* p_str) MIX(typed);
2 const char* sysutil_next_dirent(...) MIX(typed) {
3   if (p_dirent == NULL) return NULL;
4 }
5 void str_next_dirent(...) MIX(symbolic) {
6   const char* p_filename = sysutil_next_dirent(...);
7   if (p_filename != NULL)
8     str_alloc_text(p_filename);
9 }
10 ...str_alloc_text(str); sysutil_free(str); ...
```

In this example, the function str_next_direct calls sysutil_next_dirent on line 6, which may return a null value. Hence p_filename may be null. The type system ignores the null check on line 7 and due to context-insensitivity, conflates p_filename with other variables, such as str, that are passed to str_alloc_text (lines 8 and 10). Hence the type system believes str may be null. However, str is used as an argument to sysutil_free (line 10), which leads the type system to report a false warning.

Annotating function str_next_dirent as symbolic, while leaving sysutil_next_dirent and str_alloc_text as typed, successfully eliminates this warning: the symbolic executor correctly determines that p_filename is not null when it is used as an argument to str_alloc_text. And although the extra precision does not matter in this particular example, notice that the call on line 8 will be analyzed in a separate invocation of the type system than the call on line 10, thus introducing some context-sensitivity.

***Case 3: Flow- and path-insensitivity in* dns_resolve *and* main**

```
1 void main_BLOCK(struct sockaddr** p_sock) MIX(symbolic) {
2   *p_sock = NULL;
3   dns_resolve(p_sock, tunable_pasv_address);
4 }
5 int main(...) {
6   ...main_BLOCK(&p_addr); ...; sysutil_free(p_addr); ...
7 }
8 void dns_resolve(struct sockaddr** p_sock,
9                  const char* p_name) {
10  struct hostent* hent = gethostbyname(p_name);
11  sockaddr_clear(p_sock);
12  if (hent→h_addrtype == AF_INET)
13    sockaddr_alloc_ipv4(p_sock);
14  else if (hent→h_addrtype == AF_INET6)
15    sockaddr_alloc_ipv6(p_sock);
16  else
17    die("gethostbyname():␣neither␣IPv4␣nor␣IPv6");
18 }
```

There are two sources of null values in the code above: *p_sock is set to null on line 2; and sockaddr_clear, which was previously marked as symbolic in *Case 1* above, also sets *p_sock to null on line 11 in dns_resolve. Due to flow insensitivity in the type system, both these null values eventually reach sysutil_free on line 6, leading to false warnings.

However, we can see that these null values are actually overwritten by non-null values on lines 13 and 15, where sockaddr_alloc_ipv4 or sockaddr_alloc_ipv6 allocates the appropriate structure and assigns it to *p_sock (not shown). We can eliminate these warnings by extracting the code in main that includes both null sources into a symbolic block.

Also, there is a system call gethostbyname on line 10 that we need to handle. Here, we define a well-behaved, symbolic model of gethostbyname that returns only AF_INET and AF_INET6 as is standard (not shown). This will cause the symbolic executor to skip the last branch on line 17, which we need to do because we cannot analyze die symbolically as it eventually calls a function pointer, an operation that our symbolic executor currently has limited support for. We also cannot put gethostbyname or die in typed blocks in this case, since *p_sock is null and will result in false warnings.

Case 4: Helping symbolic execution with symbolic function pointers

```
1 void sysutil_exit_BLOCK(void) MIX(typed) {
2   if (s_exit_func) (*s_exit_func)();
3 }
4 void sysutil_exit(int exit_code) {
5   sysutil_exit_BLOCK();
6   exit(exit_code);
7 }
```

In several instances, we would like to evaluate symbolic blocks that call sysutil_exit, defined on line 4, which in turn calls exit to terminate the program. However, before terminating the program, sysutil_exit calls the function pointer s_exit_func on line 2. Our symbolic executor does not support calling symbolic function pointers (i.e., which targets are unknown), so instead, we extract the call to s_exit_func into a typed block to analyze the call conservatively.

4.6 Discussion and Future Work

Our preliminary experience provides some real-world validation of MIX's efficacy in removing false positives. However, there are several limitations to be addressed in future work.

Most importantly, the overwhelming source of issues in MIXY is its coarse treatment of aliasing, which relies on an imprecise pointer analysis. One immediate consequence is that it impedes performance in the symbolic executor: if an imprecise pointer analysis returns large points-to sets for pointers, translating symbolic pointers to type constraints becomes slow because we first need to check if each pointer target is valid in the current path condition by calling the SMT solver, then determine if any valid targets may be null. This leads to a significant slowdown: our small examples from Section 4.5 take less than a second to run without symbolic blocks, but from 5 to 25 seconds to run with one symbolic block, and about 60 seconds with two symbolic blocks. This issue is further compounded by the fixed-point computation that repeatedly analyzes symbolic blocks nested in typed blocks or for handling recursion.

We also noticed several cases in `vsftpd` where calls to symbolic blocks would help introduce context sensitivity to distinguish calls to `malloc`. However, since we rely on a context-insensitive pointer analysis to restore aliasing relationships when switching to typed blocks, these calls will again be conflated. The issue especially affects the analysis of typed-to-symbolic-to-typed recursive blocks because the nested typed blocks are polluted by aliasing relationships from the entire program. A similar issue occurs with symbolic blocks, as pointers are initialized to point to targets from the entire program, rather than being limited to the enclosing context.

Just as in the formalism, MIXY has to consider the entire memory when switching from typed to symbolic or vice-versa. Since this was a deliberate design decision, we were not surprised to find out that this has an impact on performance and leads to many limitations in practice. Any temporary violation of type invariants from symbolic blocks would immediately be flagged when switching to typed blocks, even if they have no effect on the code in the typed blocks. In the other direction, symbolic blocks are forced to start with a fresh memory when switching from typed blocks even if there were no effects.

Ultimately, we believe that these issues can be addressed with more precise information about aliasing as well as effects, perhaps extracted directly from the type inference constraints and symbolic execution.

In addition to checking for null pointer errors, we plan to extend MIXY to check other properties, such as data races, and to mix other types of analysis together. We also plan to investigate automatic placement of type/symbolic blocks, i.e., essentially using MIX as an intermediate language for combining analyses. One idea is to begin with just typed blocks and then incrementally add symbolic blocks to refine the result. This approach resembles abstraction refinement (e.g., Ball and Rajamani [2002]; Henzinger et al. [2004]), except the refinement can be obtained using completely different analyses instead of one particular family of abstractions.

5. Related Work

There are several threads of related work. There have been numerous proposals for static analyses based on type systems; see Palsberg and Millstein [2008] for pointers. Symbolic execution was first proposed by King [1976] as an enhanced testing strategy, but was difficult to apply for many years. Recently, SMT solvers have become very powerful, making symbolic execution much more attractive as even very complex path conditions can be solved surprisingly fast. There have been many recent, impressive results using symbolic execution for bug finding [Cadar et al. 2006, 2008; Godefroid et al. 2005; Sen et al. 2005]. These systems use symbolic execution to explore a small subset of the possible program paths, since in the presence of loops with symbolic bounds, pure symbolic execution will not terminate in a reasonable amount of time (unless loop invariants are assumed). In the MIX formalism, in contrast, we use symbolic execution in a sound manner by exploring all paths, which is possible because we can use type checking on parts of the code where symbolic execution takes too long. Of course, it is also possible to mix unsound symbolic execution with type checking, to gain whatever level of assurance the user desires.

There are several static analyses that can operate at different levels of abstraction. Bandera [Corbett et al. 2000] is a model checking system that uses abstraction-based program specialization, in which the user specifies the exact abstractions to use. System Z is an abstract interpreter generator in which the user can tune the level of abstraction to trade off cost and precision [Yi and Harrison 1993]. Tuning these systems requires a deep knowledge of program analysis. In contrast, we believe that MIX's tradeoff is easier to understand—one selects between essentially no abstraction (symbolic execution), or abstraction in terms of types, which are arguably the most successful, well-understood static analysis.

MIX bears some resemblance to static analysis based on abstraction refinement, such as SLAM [Ball and Rajamani 2002], BLAST [Henzinger et al. 2004], and client-driven pointer analysis [Guyer and Lin 2005]. These tools incrementally refine their abstraction of the program as necessary for analysis. Adding symbolic blocks to a program can be seen as introducing a very precise "refinement" of the program abstraction.

There are a few systems that combine type checking or inference with other analyses. Dependent types provide an elegant way to augment standard type with very rich type refinements [Xi and Pfenning 1999]. Liquid types combines Hindley-Milner style type inference with predicate abstraction [Rondon et al. 2008, 2010]. Hybrid types combines static typing, theorem proving, and dynamic typing [Flanagan 2006]. All of these systems combine types with refinements at a deep level—the refinements are placed "on top of" the type structure. In contrast, MIX uses a much coarser approach in which the precise analysis is almost entirely separated from the type system, except for a thin interface between the two systems.

Many others have considered the problem of combining program analyses. A reduced product in abstract interpretation [Cousot and Cousot 1979] is a theoretical description of the most precise combination of two abstract domains. It is typically obtained via manually defined reduction operators that depend on the domains being combined. Another example of combining abstract domains is the logical product of Gulwani and Tiwari [2006]. Combining program analyses for compiler optimizations is also well-studied (e.g., Lerner et al. [2002]). In all of these cases, the combinations strengthen the kinds of derivable facts over the entire program. With MIX, we instead analyze separate parts of the program with different analyses. Finally, MIX was partially inspired by Nelson-Oppen style cooperating decision procedures [Nelson and Oppen 1979]. One important feature of the Nelson-Oppen framework is that it provides an automatic method for distributing the appropriate formula fragments to each solver (if that the solvers match certain criteria). Clearly MIX is targeted at solving a very different problem, but it would be an interesting direction for future work to try to extend MIX into a similar framework that can automatically integrate analyses that have appropriately structured interfaces.

6. Conclusion

We presented MIX, a new approach for mixing type checking and symbolic execution to trade off efficiency and precision. The key feature of our approach is that the mixed systems are essentially completely independent, and they are used in an off-the-shelf manner. Only at the boundaries between typed blocks—which the user inserts to indicate where type checking should be used—and symbolic blocks—the symbolic checking annotation—do we invoke special mix rules to translate information between the two systems. We proved that MIX is sound (which implies that type checking and symbolic execution are also independently sound). We also described a preliminary implementation, MIXY, which performs null/non-null type qualifier inference for C. We identified several cases in which symbolic execution could eliminate false positives from type inference. In sum, we believe that MIX provides a promising new approach to trade off precision and efficiency in static analysis.

Acknowledgments

We would like to thank the anonymous reviewers and Patrice Godefroid for their helpful comments and suggestions. This research was supported in part by DARPA ODOD.HR00110810073, NSF CCF-0541036, and NSF CCF-0915978.

References

Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages (POPL)*, pages 1–3, 2002.

Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction (MPC)*, pages 102–126, 2000.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Computer and Communications Security (CCS)*, pages 322–335, 2006.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering (ICSE)*, pages 439–448, 2000.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, pages 269–282, 1979.

Cormac Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, pages 245–256, 2006.

Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6): 1035–1087, 2006.

Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer-Aided Verification (CAV)*, pages 519–531, July 2007.

Patrice Godefroid. Compositional dynamic test generation. In *Principles of Programming Languages (POPL)*, pages 47–54, 2007.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *Programming Language Design and Implementation (PLDI)*, pages 376–386, 2006.

Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2):83–114, 2005.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Principles of Programming Languages (POPL)*, pages 232–244, 2004.

Khoo Yit Phang, Bor-Yuh Evan Chang, and Jeffrey S. Foster. Mixing type checking and symbolic execution (extended version). Technical Report CS-TR-4954, Department of Computer Science, University of Maryland, College Park, 2010.

James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Principles of Programming Languages (POPL)*, pages 270–282, 2002.

Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *International Conference on Software Engineering (ICSE)*, pages 416–426, 2007.

Joe M. Morris. A general axiom of assignment. Assignment and linked data structure. A proof of the Schorr-Waite algorithm. In *Theoretical Foundations of Programming Methodology*, pages 25–51, 1982.

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction (CC)*, pages 213–228, 2002.

Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

Jens Palsberg and Todd Millstein. Type Systems: Advances and Applications. In *The Compiler Design Handbook: Optimizations and Machine Code Generation*, chapter 9. 2008.

Polyvios Pratikakis, Jeffrey S. Foster, and Michael W. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Programming Language Design and Implementation (PLDI)*, pages 320–331, 2006.

Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *International Conference on Software Engineering (ICSE)*, 2010. To appear.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI)*, pages 159–169, 2008.

Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *Principles of Programming Languages (POPL)*, pages 131–144, 2010.

Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Foundations of Software Engineering (FSE)*, pages 263–272, 2005.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Principles of Programming Languages (POPL)*, pages 214–227, 1999.

Kwangkeun Yi and Williams Ludwell Harrison, III. Automatic generation and management of interprocedural program analyses. In *Principles of Programming Languages (POPL)*, pages 246–259, 1993.