Active Property Checking

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Michael Y. Levin
Microsoft Center for Software Excellence
mlevin@microsoft.com

David Molnar
UC Berkeley (Visiting Microsoft)
t-davidm@microsoft.com
dmolnar@eecs.berkeley.edu

Abstract

Runtime property checking (as implemented in tools like Purify or Valgrind) checks whether a program execution satisfies a property. *Active property checking* extends runtime checking by checking whether the property is satisfied by *all* program executions that follow the same program path. This check is performed on a symbolic execution of the given program path using a constraint solver. If the check fails, the constraint solver generates an alternative program input triggering a new program execution that follows the same program path but exhibits a property violation. Combined with systematic dynamic test generation, which attempts to exercise all feasible paths in a program, active property checking defines a new form of dynamic software model checking (program verification). In this paper, we formalize and study active property checking. We show how static and dynamic type checking can be extended with active type checking. Then, we discuss how to implement active property checking efficiently. Finally, we discuss results of experiments with large shipped Windows applications, where active property checking was able to detect several new security-related bugs.

1. Introduction

During the last decade, code inspection for standard programming errors has largely been automated with static code analysis. Commercial static program analysis tools (e.g., [6, 18, 12, 24, 30]) are now routinely used in many software development organizations. These tools are popular because they find many (real) software bugs, thanks to three main ingredients: they are *automatic*, they are *scalable*, and they check *many properties*. Intuitively, any tool that is able to check automatically (with good enough precision) millions of lines of code against hundreds of coding rules and properties is bound to find on average, say, one bug every thousand lines of code.

Our research goal is to automate, as much as possible, an even more expensive part of the software development process, namely *software testing*, which usually accounts for about 50% of the R&D budget of software development organizations. In particular, we want to automate *test generation* by leveraging recent advances in program analysis, automated constraint solving, and the increasing computation power available on modern computers. To replicate the success of static program analysis in the testing space, we need the same key ingredients: automation, scalability and the ability to check many properties.

Automating test generation from program analysis is an old idea [23, 27], and work in this area can roughly be partitioned into two groups: static versus dynamic test geneneration. *Static test generation* [23, 3, 33, 10] consists of analyzing a program statically to attempt to compute input values to drive its executions along specific program paths. In contrast, *dynamic test generation* [25, 14, 7, 15] consists in executing the program, typically starting with some random inputs, while simultaneously performing a symbolic execution to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer program executions along alternative program paths. Since dynamic test generation extends static test generation with additional runtime information, it can be more powerful [14]. Scalability in this context has been recently addressed in [13, 15]. The motivation of the present work is to address the third challenge, which has been largely unexplored so far: how to dynamically check many properties simultaneously, thoroughly and efficiently, in order to maximize the chances of finding bugs during an automated testing session?

Traditional runtime checking tools like Purify [20], Valgrind [29] and AppVerifier [8] check a *single* program execution against a set of properties (such as the absence of buffer overflows, uninitialized variables or memory leaks). We show in this paper how such traditional *passive* runtime property checkers can be extended to *actively* search for property violations. Consider the simple program:

```
int divide(int n,int d) { // n and d are inputs
  return (n/d); // division-by-zero error if d==0
}
```

The program `divide` takes two integers `n` and `d` as inputs and computes their division. If the denominator `d` is zero, an error occurs. To catch this error, a traditional runtime checker for division-by-zero would simply check whether the concrete value of `d` satisfies `(d==0)` just before the division is performed in that specific execution, but would not provide any insight or guarantee concerning other executions. Testing this program with random values for `n` and `d` is unlikely to detect the error, as `d` has only one chance out of 2^{32} to be zero if `d` is a 32-bit integer. Static and dynamic test generation techniques that attempt to cover specific or all feasible paths in a program will also likely miss the error since this program has a single program path which is covered no matter what inputs are used. However, the latter techniques could be helpful provided that a test `if (d==0) error()` was inserted before the division `(n/d)`: they could then attempt to generate an input value for `d` that satisfies the constraint `(d==0)`, now present in the program path, and detect the error. This is essentially what *active property checking* does: it injects *at runtime* additional symbolic constraints

MSR-TR-2007-91, July 2007.

on inputs that, when solvable by a constraint solver, will generate new test inputs leading to property violations.

In other words, *active property checking* extends runtime checking by checking whether the property is satisfied by *all* program executions that follow the same program path. This check is performed on a dynamic symbolic execution of the given program path using a constraint solver. If the check fails, the constraint solver generates an alternative program input triggering a new program execution that follows the same program path but exhibits a property violation. We call this "active" checking because a constraint solver is used to "actively" look for inputs that cause a runtime check to fail. Combined with systematic dynamic test generation, which attempts to exercise all feasible paths in a program, active property checking defines a new form of program verification.

Checking properties at runtime on a dynamic symbolic execution of the program was suggested in [26], but may return false alarms whenever symbolic execution is imprecise, which is often the case in practice. Active property checking extends the idea of [26] by combining it with constraint solving and test generation in order to further check using a new test input whether the property is actually violated as predicted by the prior imperfect symbolic execution. This way, no false alarms are ever reported. Active property checking can also be viewed as systematically injecting assertions all over the program under test, and then using dynamic test generation to check for violations of those assertions. Related ideas are mentioned informally in [7, 22] but not explored in detail.

Our work makes several contributions:

- We formalize active property checking semantically in Section 3 and show how it provides a new form of program verification when combined with systematic dynamic test generation (recalled in Section 2).
- Section 4 presents a *type system* that combines static, dynamic and active checking for a simple imperative language. This clarifies the connection, difference and complementarity between *active type checking* and traditional static and dynamic type checking.
- Section 5 discusses how to implement active checking efficiently by minimizing the number of calls to the constraint solver, minimizing formula sizes and using two constraint caching schemes.
- Section 6 describes our implementation of active property checking in SAGE [15], a recent tool for security testing of file-reading Windows applications that performs systematic dynamic test generation of x86 binaries. Results of experiments with large shipped Windows applications are discussed in Section 7. Active property checking was able to detect several new bugs in those applications.

2. Systematic Dynamic Test Generation

Dynamic test generation (see [14] for further details) consists of running the program P under test both concretely, executing the actual program, and symbolically, calculating constraints on values stored in program variables x and expressed in terms of input parameters α. Side-by-side concrete and symbolic executions are performed using a concrete store Δ and a symbolic store Σ, which are mappings from program variables to concrete and symbolic values respectively. A symbolic value is any expression sv in some theory $\mathcal{T}$ where all free variables are exclusively input parameters α. For any variable x, $\Delta(x)$ denotes the *concrete value* of x in Δ, while $\Sigma(x)$ denotes the *symbolic value* of x in Σ. The judgment $\Delta \vdash e \rightarrow v$ means that that an expression e reduces to a concrete value v, and similarly $\Sigma \vdash e \rightarrow sv$ means that e reduces to a symbolic value sv. For notational convenience, we assume that $\Sigma(x)$ is always defined and is simply $\Delta(x)$ by default if no expression in terms of inputs is associated with x. The notation $\Delta(x \mapsto c)$ denotes updating the mapping Δ so that x maps to c.

$$\text{X-ASN}\ \frac{\Delta \vdash e \rightarrow v \qquad \Delta' = \Delta(x \mapsto v) \qquad \Sigma \vdash e \rightarrow sv \qquad \Sigma' = \Sigma(x \mapsto sv)}{\langle x := e; C, \Delta, \Sigma, pc\rangle \rightarrow \langle C, \Delta', \Sigma', pc\rangle}$$

$$\text{X-IF1}\ \frac{\Delta \vdash e \rightarrow \texttt{true} \qquad \Sigma \vdash e \rightarrow sv \qquad pc' = pc \wedge sv}{\langle \texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ C', \Delta, \Sigma, pc\rangle \rightarrow \langle C, \Delta, \Sigma, pc'\rangle}$$

$$\text{X-IF2}\ \frac{\Delta \vdash e \rightarrow \texttt{false} \qquad \Sigma \vdash e \rightarrow sv \qquad pc' = pc \wedge \neg sv}{\langle \texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ C', \Delta, \Sigma, pc\rangle \rightarrow \langle C', \Delta, \Sigma, pc'\rangle}$$

Figure 1. Side-by-side (concrete and symbolic) evaluation.

The program P manipulates the memory (concrete and symbolic stores) through statements, or *commands*, that are abstractions of the machine instructions actually executed. A command can be an *assignment* of the form $x := e$ (where x is a program variable and e is an expression), a *conditional statement* of the form $\texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ C'$ where e denotes a boolean expression, and C and C' are *continuations* denoting the unique next statement to be evaluated (programs considered here are thus sequential and deterministic), or `stop` corresponding to a program error or normal termination.

Given an input vector $\vec{\alpha}$ assigning a value to every input parameter α, the evaluation of a program defines a unique finite *program execution* $s_0 \stackrel{C_1}{\rightarrow} s_1 \ldots \stackrel{C_n}{\rightarrow} s_n$ that executes the finite sequence $C_1 \ldots C_n$ of commands and goes through the finite sequence $s_1 \ldots s_n$ of *program states*. Each program state is a tuple $\langle C, \Delta, \Sigma, pc\rangle$ where C is the next command to be evaluated, and pc is a special meta-variable that represents the current *path constraint*. For a finite sequence w of statements (i.e., a control path w), a path constraint pc_w is a formula of theory $\mathcal{T}$ that characterizes the input assignments for which the program executes along w. To simplify the presentation, we assume that all the program variables have some default initial concrete value in the initial concrete store Δ_0, and that the initial symbolic store Σ_0 identifies the program variables v whose values are program inputs (for all those, we have $\Sigma_0(v) = \alpha$ where α is some input parameter). Initially, pc is defined to `true`.

The main rules for side-by-side execution are shown in Figure 1. The X-ASN rule shows how both the concrete store and symbolic store are updated after an assignment. The rules X-IF1 and X-IF2 show how the path constraint pc is updated after each conditional statement. First the boolean expression e is evaluated to determine if its concrete value is `true` or `false` and its symbolic value sv. Next, depending on the result, a new conjunct sv or $\neg sv$ is added to the current path constraint pc. For simplicity, we assume that all program executions eventually terminate by executing the command `stop`.

Systematic dynamic test generation [14] consists of systematically exploring all feasible program paths of the program under test by using path constraints and a constraint solver. By construction, a path constraint represents conditions on inputs that need be satisfied for the current program path to be executed. Given a program state $\langle C, \Delta, \Sigma, pc\rangle$ and a constraint solver for theory $\mathcal{T}$, if C is a conditional statement of the form $\texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ C'$, any satisfying assignment to the formula $pc \wedge sv$ (respectively $pc \wedge \neg sv$) defines program inputs that will lead the program to execute the `then` (resp. `else`) branch of the conditional statement. By systematically repeating this process, such a *directed search* can enumerate all possible path constraints and eventually execute all feasible program paths.

```
1 int buggy(int x) { // x is an input
2   int buf[20];
3   buf[30]=0; // buffer overflow independent of x
4   if (x > 20)
5     return 0;
6   else
7     return buf[x]; // buffer overflow if x==20
8 }
```

Figure 2. Example of program with buffer overflows.

The search is exhaustive provided that the generation of the path constraint (including the underlying symbolic execution) and the constraint solver for the given theory $\mathcal{T}$ are both *sound and complete*, that is, for all program paths w, the constraint solver returns a satisfying assignment for the path constraint pc_w *if and only if* the path is feasible (i.e., there exists some input assignment leading to its execution). In this case, in addition to finding errors such as the reachability of bad program statements (like `assert(0)`), a directed search can also prove their absence, and therefore obtain a form of program *verification*.

Theorem 1. *(adapted from [14]) Given a program P as defined above, a directed search using a path constraint generation and a constraint solver that are both sound and complete exercises all feasible program paths exactly once.*

In this case, if a program statement has not been executed when the search is over, this statement is not executable in any context.

In practice, path constraint generation and constraint solving are usually not sound and complete. When a program expression cannot be expressed in the given theory $\mathcal{T}$ decided by the constraint solver, it can be simplified using concrete values of sub-expressions, or replaced by the concrete value of the entire expression. For example, if the solver handles only linear arithmetic, symbolic sub-expressions involving multiplications can be replaced by their concrete values.

3. Active Checkers

Even when sound and complete, a directed search based on path exploration alone can miss errors that are not path invariants, i.e., that are not violated by *all* concrete executions executing the same program path, or errors that are not caught by the program's runtime environment. This is illustrated by the simple program shown in Figure 2.

This program takes as (untrusted) input an integer value stored in variable `x`. A buffer overflow in line 3 will be detected at runtime only if a runtime checker monitors buffer accesses. Such a runtime checker would thus check whether any array access of the form `a[x]` satisfies the condition $0 \leq \Delta(\mathtt{x}) < b$ where $\Delta(\mathtt{x})$ is the concrete value of array index `x` and b denotes the bound of the array `a` (b is 20 for the array `buf` in the example of Figure 2). Let us call such a traditional runtime checker for concrete values a *passive checker*.

Moreover, a buffer overflow is also possible in line 7 provided `x==20`, yet a directed search focused on path exploration alone may miss this error. The reason is that the only condition that will appear in a path constraint for this program is `x > 20` and its negation. Since most input values for `x` that satisfy $\neg$`(x > 20)` do not cause the buffer overflow, the error will likely be undetected with a directed search as defined in the previous section.

In order to catch the buffer overflow on line 7, the program should be extended with a *symbolic* test $0 \leq \Sigma(\mathtt{x}) < b$ (where $\Sigma(\mathtt{x})$ denotes the symbolic value of array index `x`) just before the buffer access `buf[x]` on line 7. This will force the condition $0 \leq \mathtt{x} < 20$ to appear in the path constraint of the program in order to refine the partitioning of its input values. An *active checker* for array bounds can be viewed as systematically adding such symbolic tests before all array accesses.

Formally, we define *passive checkers* and *active checkers* as follows.

Definition 1. *A* passive checker *for a property π is a function that takes as input a finite program execution w, and returns $\mathtt{fail}_\pi$ iff the property π is violated by w.*

Because we assume all program executions terminate, properties considered here are *safety* properties. Runtime property checkers like Purify [20], Valgrind [29] and AppVerifier [8] are examples of tools implementing passive checkers.

Definition 2. *Let pc_w denote the path constraint of a finite program execution w. An active checker for a property π is a function that takes as input a finite program execution w, and returns a formula ϕ_C such that the formula $pc_w \wedge \neg\phi_C$ is satisfiable iff there exists a finite program execution w' violating property π and such that $pc_{w'} = pc_w$.*

Active checkers can be implemented in various ways, for instance using property monitors/automata, program rewrite rules or type checking. They can use private memory to record past events (leading to the current program state), but they are not allowed any side effect on the program. Sections 4 and 6 discuss detailed examples of *how* active checkers can be formally defined and implemented. Here are examples of specifications of active checkers.

Example 1. ***Division By Zero.*** *Given a program state where the next statement involves a division by a denominator `d` which depends on an input (i.e., such that $\Sigma(\mathtt{d}) \neq \Delta(\mathtt{d})$), an active checker for division by zero outputs the constraint $\phi_{Div} = (\Sigma(\mathtt{d}) \neq 0)$.*

Example 2. ***Array Bounds.*** *Given a program state where the next statement involves an array access `a[x]` where `x` depends on an input (i.e., is such that $\Sigma(\mathtt{x}) \neq \Delta(\mathtt{x})$), an active checker for array bounds outputs the constraint $\phi_{Buf} = (0 \leq \Sigma(\mathtt{x}) < b)$ where b denotes the bound of the array `a`.*

Example 3. ***NULL Pointer Dereference.*** *Consider a program expressed in a language where pointer dereferences are allowed (unlike our simple language SimpL). Given a program state where the next statement involves a pointer dereference `*p` where `p` depends on an input (i.e., such that $\Sigma(\mathtt{p}) \neq \Delta(\mathtt{p})$), an active checker for NULL pointer dereference generates the constraint $\phi_{NULL} = (\Sigma(\mathtt{p}) \neq \mathtt{NULL})$.*

Multiple active checkers can be used simultaneously by simply considering separately the constraints they inject in a given path constraint. Such a way, they are guaranteed not to interfere with each other (since they have no side effects). We will discuss how to combine active checkers to maximize performance in Section 5.

By applying an active checker for a property π to all feasible paths of a program P, we can obtain a form of *verification* for this property, that is stronger than Theorem 1.

Theorem 2. *Given a program P as defined above, if a directed search (1) uses a path constraint generation and constraint solvers that are both sound and complete,* and *(2) uses both a passive and an active checker for a property π in all program paths visited during the search, then the search reports $\mathtt{fail}_\pi$ iff there exists a program input that leads to a finite execution violating ϕ.*

Proof Sketch: Assume there is an input assignment that leads to a finite execution w of P violating π. Let pc_w be the path constraint for the execution path w. Since path constraint generation and constraint solving are both sound and complete, we know by

Theorem 1 that w will eventually be exercised with some concrete input assignment $\vec{\alpha}'$. If the passive checker for π returns $\texttt{fail}_\pi$ for the execution of P obtained from input $\vec{\alpha}'$ (for instance, if $\vec{\alpha}' = \vec{\alpha}$), the proof is finished. Otherwise, the active checker for π will generate a formula ϕ_C and call the constraint solver with the query $pc_w \wedge \neg\phi_C$. The existence of $\vec{\alpha}'$ implies that this query is satisfiable, and the constraint solver will return a satisfying assignment from which a new input assignment $\vec{\alpha}''$ is generated ($\vec{\alpha}''$ could be $\vec{\alpha}$ itself). By construction, running the passive checker for π on the execution obtained from that new input $\vec{\alpha}''$ will return $\texttt{fail}_\pi$. □

Note that both passive checking and active checking are required in order to obtain this result, as illustrated earlier with the example of Figure 2. In practice, however, symbolic execution, path constraint generation, constraint solving, passive and active property checking are typically not sound and complete, and therefore active property checking reduces to testing.

4. Active Type Checking

Now we describe a framework for specifying checkers that illustrates their complementarity with traditional static and dynamic checking. Our framework is inspired by the "hybrid type checking" of Flanagan [11], which builds on [2, 1, 21, 32]. Flanagan observes that type-checking a program statically is undecidable in general, especially for type systems that permit expressive specifications. Therefore, we need a way to handle programs for which we cannot decide statically that the program violates a property, but may in fact satisfy the property. He then shows how to automatically insert run-time checks for programs in a language λ^H in cases where typing cannot be decided statically.

We show how to extend this idea to active checking with a simple imperative language `CSimpL` and a type system in 4.1 that supports *integer refinement types*, in which types are defined by predicates, and subtyping is defined by logical implication between these predicates. Then, in 4.2, we give a method for *compiling* programs that either statically rejects a program as ill-typed, or inserts *casts* to produce a well-typed program. Each cast performs a run-time membership check for the given type and raises an error if a run-time value is not of the desired type. The approach here is similar to Flanagan's λ^H system, but for our imperative language.

The key property of our approach is that the run-time check is a *passive checker* in the sense of the previous section: the post-compilation program computes a function on its own execution that returns $\texttt{fail}_\phi$ if and only if the run-time values violate a cast's type membership check. Then, in 4.3 we define *side-by-side symbolic and concrete evaluation* of `CSimpL` to generate symbolic path conditions from program executions and *symbolic membership checks* from casts. We show that these symbolic membership checks are *active checkers* with respect to the run-time membership checks. Therefore, a type environment can be thought of as specifying a property: we first attempt to prove this property holds statically or reject the program statically. If we cannot decide in some portions of the program, we insert casts. The inserted casts give rise to passive and active checkers for this property.

We then give two examples of specifying properties with type environments in our system, division by zero and integer overflow. Finally, we show cases when we can combine different type environments to simultaneously check different properties.

4.1 Simple Language with Casts `CSimpL`

We now define the semantics and type system for an imperative language with casts, `CSimpL`. This language will allow us to demonstrate the idea of active type checking.

A *value* v is either an integer i or a boolean constant b. An *operand* o is either a value or a variable reference x. An *expression* e is either an operand or an operator application $\texttt{op}(o_1 \ldots o_n)$

$$\text{T-VAR}\quad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\text{T-BC}\quad \frac{}{\Gamma \vdash b : \texttt{Bool}}$$

$$\text{T-IC}\quad \frac{}{\Gamma \vdash i : \{x : \texttt{Int} | x == i\}}$$

$$\text{T-OP}\quad \frac{\Gamma \vdash o_1 : S_1 \ldots \Gamma \vdash o_n : S_n \qquad \Gamma \vdash \texttt{op}(S_1 \ldots S_n) : T}{\Gamma \vdash \texttt{op}(o_1 \ldots o_n) : T}$$

$$\text{T-SUB}\quad \frac{\Gamma \vdash e : S \qquad S <: T}{\Gamma \vdash e : T}$$

$$\text{T-IF}\quad \frac{\Gamma' = \Gamma \cap G \qquad \Gamma' \vdash e : \texttt{Bool} \qquad \Gamma' \vdash C \qquad \Gamma' \vdash C'}{\Gamma \vdash (G)\texttt{if } e \texttt{ then } C \texttt{ else } C'}$$

$$\text{T-ASN}\quad \frac{\Gamma' = \Gamma \cap G \qquad \Gamma' \vdash e : S \qquad \Gamma \vdash x : U \qquad T \cap S <: U \qquad \Gamma' \vdash C}{\Gamma \vdash (G)x := \langle T \rangle e; C}$$

$$\text{T-STOP}\quad \frac{}{\Gamma \vdash \texttt{stop}}$$

Figure 3. Typing.

for some operator name `op` and operands $o_1 \ldots o_n$. An *operator denotation* $\tilde{op}$ is a partial function from tuples of values to a value. A *concrete store* Δ is a map from variables and operator names to values and operator denotations respectively.

`CSimpL` supports integer refinement and boolean types. Type `Bool` classifies boolean expressions and boolean values `true` and `false`. Integer refinement types have the form $\{x : \texttt{Int} | t\}$ for some boolean expression t whose only free variable may be x. A refinement type denotes the set of integers that satisfy the boolean expression. A refinement type T is said to be a subtype of a refinement type S, written $T <: S$, if the denotation of T is a subset of the denotation of S.

A value v is said to have type T, written $v \in T$, either if v is a boolean value and T is `Bool` or if v is an integer in the denotation of T. Note that this value typing relation is decidable.

A *type environment* Γ is a map from variables and operator names to types and operator signatures respectively. An operator signature has the form $\texttt{op}(S_1 \ldots S_n) : T$ where $S_1 \ldots S_n$ are the types of the parameters and T is the type of the result. A *cast set* G is a type environment whose domain contains only variables.

We say that a concrete store Δ corresponds to a type environment Γ, written $\Delta \in \Gamma$ if for any variable x, we have $\Delta(x) \in \Gamma(x)$ and for any operator `op`, we have $\Gamma(\texttt{op}) = \texttt{op}(S_1 \ldots S_n) : T$ and $\Delta(\texttt{op}) = \tilde{op}$ with $\tilde{op}$ defined on any value tuple $v_1 \ldots v_n$ such that $v_i \in S_i$ for $0 < i \leq n$ and $\tilde{op}(v_1 \ldots v_n) \in T$. We say that a concrete store Δ satisfies a cast set G, written $\Delta \vdash G$, if for any variable x in the domain of G, we have $\Delta(x) \in G(x)$.

Given two refinement types $T = \{x : \texttt{Int} | t_1\}$ and $S = \{x : \texttt{Int} | t_2\}$, the intersection of T and S, denoted $T \cap S$ is defined to be the refinement type $T = \{x : \texttt{Int} | t_1 \wedge t_2\}$. Assuming that two type environments Γ_1 and Γ_2 agree on the return types of operators, $\Gamma_1 \cap \Gamma_2$ is defined point-wise.

A *program* C consists of commands and is defined by the following grammar:

$$\begin{array}{rcl} C, C' & ::= & \texttt{stop} \\ & | & (G)x := \langle T \rangle e; C \\ & | & (G)\texttt{if } e \texttt{ then } C \texttt{ else } C' \end{array}$$

Each non-halting command is annotated with a cast set specifying the type assumptions that must be checked dynamically before the command is executed. Additionally, the assignment command also specifies a cast on the right hand side expression.

$$\text{E-VAR}\quad \frac{}{\Delta \vdash x \rightarrow \Delta(x)}$$

$$\text{E-OP}\quad \frac{\Delta \vdash o_1 \rightarrow v_1 \ldots \Delta \vdash o_n \rightarrow v_n \qquad \Delta(\texttt{op}) = \tilde{op} \qquad v = \tilde{op}(v_1 \ldots v_n)}{\Delta \vdash \texttt{op}(o_1 \ldots o_n) \rightarrow v}$$

$$\text{E-IF1}\quad \frac{\Delta \vdash G \qquad \Delta \vdash e \rightarrow \texttt{ true}}{\langle (G)\texttt{if } e \texttt{ then } C \texttt{ else } C', \Delta\rangle \rightarrow \langle C, \Delta\rangle}$$

$$\text{E-IF2}\quad \frac{\Delta \vdash G \qquad \Delta \vdash e \rightarrow \texttt{ false}}{\langle (G)\texttt{if } e \texttt{ then } C \texttt{ else } C', \Delta\rangle \rightarrow \langle C', \Delta\rangle}$$

$$\text{E-ASN}\quad \frac{\Delta \vdash G \qquad \Delta \vdash e \rightarrow v \qquad v \in T \qquad \Delta' = \Delta(x \mapsto v)}{\langle (G)x := \langle T\rangle e; C, \Delta\rangle \rightarrow \langle C, \Delta'\rangle}$$

Figure 4. Concrete evaluation.

Figure 3 defines the static semantics of `CSimpL`. It is given by the program typing judgment $\Gamma \vdash C$, which states that program C is well-typed in type environment Γ, and the expression typing judgment $\Gamma \vdash e : T$ which states that expression e is of type T i ntype environment Γ. The T-IF and T-ASN rules describe how to check a non-halting command. The premises of these rules are checked in a type environment extended with the assumptions specified in cast set of the current command. The judgment defined in Figure 3 is not algorithmic because the subsumption rule T-SUB is not syntax-directed.

Figure 4 defines the dynamic semantics of `CSimpL`. It is given by the small-step program evaluation judgment $\langle C, \Delta\rangle \rightarrow \langle C', \Delta'\rangle$ and the big-step expression evaluation judgment $\Delta \vdash e \rightarrow v$. The evaluation of a command proceeds by first ensuring that its associated cast set is satisfied by the current environment. When this succeeds, the command's subexpressions are evaluated so that the program can make a small step. For example, the E-ASN rule describes how the assignment command is executed: after the cast set is validated, the right hand side expression is evaluated and the obtained value is checked against the associated type T. If this check succeeds, the concrete store is updated and the program moves on to the next command.

We say a command C *contains a failed cast* under Δ either if the cast set of C is not satisfied by Δ or if C is of the form $(G)x := \langle T\rangle e; C$ with e evaluating to some value v such that $v \notin T$.

Theorem 3. *(Type preservation.) Let Δ and Γ be a concrete store and a type environment such that $\Delta \in \Gamma$. Then the following two properties hold:*

1. *If $\Gamma \vdash e : T$ and $\Delta \vdash e \rightarrow v$, then $\Gamma \vdash v : T$.*
2. *If $\Gamma \vdash C$ and $\langle C, \Delta\rangle \rightarrow \langle C', \Delta'\rangle$, then $\Gamma \vdash C'$ and $\Delta' \in \Gamma$.*

Theorem 4. *(Progress.) Let Δ and Γ be a concrete store and a type environment such that $\Delta \in \Gamma$. If $\Gamma \vdash C$,then either $\langle C, \Delta\rangle \rightarrow \langle C', \Delta'\rangle$, or C contains a failed cast under Δ.*

4.2 Casts Insertion

The typing relation defined above does not give us an algorithm for checking whether an arbitrary program is well-typed because it relies on checking subtyping which is undecidable in general. In practice, it is common to use a theorem prover that can validate or invalidate some subtyping assumptions and fail to produce a definitive answer on others. We model such a theorem prover by an algorithmic subtyping relation that, given two refinement types T and S, can either fail to produce an answer, written $T <:^{?}_{alg} S$, return true, written $T <:^{ok}_{alg} S$, or return false, written $T \not<:^{ok}_{alg} S$ such that $T <:^{ok}_{alg} S$ and $T \not<:^{ok}_{alg} S$ imply $T <: S$ and $T \not<: S$ respectively.

$$\text{C-VAR1}\quad \frac{\Gamma(x) <:^{ok}_{alg} S}{\Gamma \vdash x : S \Rightarrow \emptyset}$$

$$\text{C-VAR2}\quad \frac{\Gamma(x) <:^{?}_{alg} S}{\Gamma \vdash x : S \Rightarrow \{x : S \cap \Gamma(x)\}}$$

$$\text{C-BC}\quad \frac{}{\Gamma \vdash b : \texttt{Bool} \Rightarrow \emptyset}$$

$$\text{C-IC}\quad \frac{\{x : \texttt{Int} | x == i\} <:^{ok}_{alg} S}{\Gamma \vdash i : S \Rightarrow \emptyset}$$

$$\text{C-OP}\quad \frac{\Gamma \vdash o_1 : S_1 \Rightarrow G' \ \ldots \ \Gamma \vdash o_n : S_n \Rightarrow G_n \qquad \Gamma \vdash \texttt{op}(S_1 \ldots S_n) : T}{\Gamma \vdash \texttt{op}(o_1 \ldots o_n) : T \Rightarrow \cap_i G_i}$$

$$\text{C-IF}\quad \frac{\Gamma' = \Gamma \cap G \qquad \Gamma' \vdash e : \texttt{Bool} \Rightarrow G' \qquad \Gamma'' = \Gamma' \cap G' \qquad \Gamma'' \vdash C_1 \Rightarrow C_1' \qquad \Gamma'' \vdash C_2 \Rightarrow C_2'}{\Gamma \vdash (G)\texttt{if } e \texttt{ then } C_1 \texttt{ else } C_2 \Rightarrow (G \cap G')\texttt{if } e \texttt{ then } C_1' \texttt{ else } C_2'}$$

$$\text{C-ASN1}\quad \frac{\Gamma' = \Gamma \cap G \qquad \Gamma \vdash x : U \qquad \Gamma' \vdash e : S \Rightarrow G' \qquad T \cap S <:^{ok}_{alg} U \qquad \Gamma'' = \Gamma' \cap G' \qquad \Gamma'' \vdash C \Rightarrow C'}{\Gamma \vdash \langle (G)x := \langle T\rangle e; C\rangle \Rightarrow \langle (G \cap G')x := \langle T\rangle e; C'\rangle}$$

$$\text{C-ASN2}\quad \frac{\Gamma' = \Gamma \cap G \qquad \Gamma \vdash x : U \qquad \Gamma' \vdash e : S \Rightarrow G' \qquad T \cap S <:^{?}_{alg} U \qquad \Gamma'' = \Gamma' \cap G' \qquad \Gamma'' \vdash C \Rightarrow C'}{\Gamma \vdash \langle (G)x := \langle T\rangle e; C\rangle \Rightarrow \langle (G \cap G')x := \langle T \cap U\rangle e; C'\rangle}$$

$$\text{C-STOP}\quad \frac{}{\Gamma \vdash \texttt{stop} \Rightarrow \texttt{stop}}$$

Figure 5. Compilation.

Figure 5 shows how with the help of such an algorithmic subtyping relation, we can define a compilation algorithm that instruments a given program with a sufficient number of casts to make it verifiably well-typed. The algorithm is comprised of the program compilation judgment $\Gamma \vdash C \Rightarrow C'$, which states that program C is compiled into program C' under type environment Γ, and the algorithmic expression typing judgment $\Gamma \vdash e : T \Rightarrow G$, which states that expression e has type T in type environment Γ provided that the type assumptions in cast set G are satisfied.

The compilation algorithm is partial: given a type environment Γ and a program C there may not exist a program C' such that $\Gamma \vdash C \Rightarrow C'$. Such a situation is denoted by $\Gamma \vdash C \Rightarrow \bot$.

The following two theorems establish the static properties of the compilation algorithm.

Theorem 5. *(Well-typed compilation.) If $\Gamma \vdash C \Rightarrow C'$, then $\Gamma \vdash C'$.*

Theorem 6. *(Compilation Rejects Only Ill-Typed Programs.) If $\Gamma \vdash C \Rightarrow \bot$, then $\Gamma \nvdash C$.*

To show that the compiled program and the result of the compilation are equivalent at run-time, we first introduce a k-step evaluation relation. A program C_0 and a store Δ_0 are said to make k steps producing a program C_k and a store Δ_k, written $\langle C_0, \Delta_0\rangle \rightarrow^k \langle C_k, \Delta_k\rangle$, if $\langle C_i, \Delta_i\rangle \rightarrow \langle C_{i+1}, \Delta_{i+1}\rangle$ for $0 \leq i < k$.

The following theorem establishes that the result of the compilation algorithm is equivalent to the original program by stating that if the latter can make k steps then the former either can make exactly the same k steps or fail on an inserted cast along the road.

Theorem 7. *(Semantic preservation.) If $\Gamma \vdash C_0 \Rightarrow C_0'$ and $\langle C_0, \Delta\rangle \rightarrow^k \langle C_k, \Delta_k\rangle$, then either $\langle C_0', \Delta\rangle \rightarrow^k \langle C_k', \Delta_k\rangle$, or $\langle C_0', \Delta\rangle \rightarrow^i \langle C_i', \Delta_i\rangle$ and C_i' contains a failed cast under Δ_i for some $0 \leq i < k$.*

$$\text{X-IF1}\quad \frac{\begin{array}{c}\Delta \vdash G \qquad \Delta \vdash e \rightarrow \texttt{true} \\ \Sigma \vdash e \rightarrow sv \qquad \Sigma \vdash G \Rightarrow \Phi \qquad pc' = pc \wedge sv\end{array}}{\begin{array}{c}\langle (G)\texttt{if } e \texttt{ then } C \texttt{ else } C', \Delta, \Sigma, pc\rangle \rightarrow \\ \langle C, \Delta, \Sigma, pc'\rangle \cdot \Phi\end{array}}$$

$$\text{X-IF2}\quad \frac{\begin{array}{c}\Delta \vdash G \qquad \Delta \vdash e \rightarrow \texttt{false} \\ \Sigma \vdash e \rightarrow sv \qquad \Sigma \vdash G \Rightarrow \Phi \qquad pc' = pc \wedge \neg sv\end{array}}{\begin{array}{c}\langle (G)\texttt{if } e \texttt{ then } C \texttt{ else } C', \Delta, \Sigma, pc\rangle \rightarrow \\ \langle C', \Delta, \Sigma, pc'\rangle \cdot \Phi\end{array}}$$

$$\text{X-ASN}\quad \frac{\begin{array}{c}\Delta \vdash G \qquad \Delta \vdash e \rightarrow v \qquad \Delta' = \Delta(x \mapsto v) \\ v \in T \qquad \Sigma \vdash e \rightarrow sv \qquad \Sigma' = \Sigma(x \mapsto sv) \\ \Sigma \vdash G \Rightarrow \Phi_1 \qquad \Sigma \vdash sv \in T \Rightarrow \phi_2\end{array}}{\begin{array}{c}\langle (G)x := \langle T\rangle e; C, \Delta, \Sigma, pc\rangle \rightarrow \\ \langle C, \Delta', \Sigma', pc\rangle \cdot (\Phi_1 \cup \{\phi_2\})\end{array}}$$

Figure 6. Side-by-side evaluation.

4.3 Active Checking via Symbolic Evaluation

Now we show how *side-by-side symbolic and concrete evaluation* of a compiled program with casts constructs both path constraints and active checker constraints.

We first introduce two auxiliary judgments used in the definition of the side-by-side evaluation judgment. We say that a symbolic value sv has type T provided that a input constraint ϕ is satisfied, written $sv \in T \Rightarrow \phi$, if $T = \{x : \texttt{Int}|t\}$ and $t[x := sv]$ where $t[x := sv]$ denotes a boolean expression obtained by substituting sv for x. We call this judgment *symbolic value typing*.

A symbolic store Σ is said to satisfy a cast set G provided that an input constraint set Φ is satisfied, written $\Sigma \vdash G \Rightarrow \Phi$, if for any variable x in the domain of G, we have $\Sigma(x) \in G(x) \Rightarrow \phi_x$ and $\Phi = \cup_x\{\phi_x\}$. We call this judgment *symbolic cast checking*.

Side-by-side evaluation is presented in Figure 6. It is given by by the single-step side-by-side program evaluation judgment $\langle C, \Delta, \Sigma, pc\rangle \rightarrow \langle C', \Delta', \Sigma', pc'\rangle \cdot \Phi$ and by the symbolic expression evaluation judgment $\Sigma \vdash e \rightarrow sv$. The former states that a configuration consisting of a program C, a concrete store Δ, a symbolic store Σ, and a path condition pc symbolically evaluates to a configuration $\langle C', \Delta', \Sigma', pc'\rangle$ producing a set of input constraints Φ. The intuition behind Φ is that if any constraint $\phi \in \Phi$ is invalidated by some input then this input will cause the program to fail on a cast. The symbolic evaluation judgment defines how an expression e evaluates to a symbolic value sv in a symbolic store Σ. The rules defining this judgment are simple, and we omit them for space considerations.

The side-by-side program evaluation rules are defined in terms of the concrete evaluation judgments discussed in Section 4.1. The concrete evaluation rules can be recovered by removing all the symbolic artifacts from the side-by-side rules. The rule X-IF1 and X-IF2 are similar to the corresponding rules described in Section 2. The key difference here is that we use the cast checking judgment $\Sigma \vdash G \Rightarrow \Phi$ which is a set of symbolic values which abstract the concrete type membership checks performed to satisfy G.

The path constraints and the cast checking constraints generated by side-by-side evaluation can be used to compute concrete inputs which drive the program to a failed cast. We write $\Sigma \circ \vec{\alpha} \rightarrow \Delta$ to represent substituting input parameters by values and then reducing each entry of Σ to obtain a concrete store. We will write $\vec{\alpha}(\phi) \rightarrow v$ to mean substituting values for variables in ϕ and reducing to a value. For a concrete store Δ, we write $\Delta \prec \Sigma$ if there exists an $\vec{\alpha}$ that reduces Σ to Δ.

We can now prove that a checker constraint generated at a program point is an active checker for the property of failing a run-time type membership check. That is, if we take a path constraint pc and a constraint $\phi \in \Phi$, then $pc \wedge \neg\phi$ is satisfiable if and only if there is some execution along the same path that causes a run-time check to fail. The main idea is that this should hold because ϕ is an abstraction of the run-time membership check. We state this formally as follows.

$\texttt{notzero} : \{x : \texttt{Int}|x \neq 0\}$
$\texttt{div} : \texttt{op}(\texttt{int}, \texttt{notzero}) : \texttt{int}$
$\tilde{div}(n_1, n_2) = n_1/n_2$

Figure 7. Type and implementation for the division operator.

Theorem 8. *(Soundness and Completeness of Active Type Checking.) Let Δ_0 and Σ_0 be concrete and symbolic stores such that $\Delta_0 \prec \Sigma_0$, and let Γ be a type environment such that $\Delta_0 \in \Gamma$. Let pc_0 be a path constraint. Let $\Gamma \vdash C_0$, and $\langle C_0, \Delta_0, \Sigma_0, pc_0\rangle \rightarrow^k \langle C_k, \Delta_k, \Sigma_k, pc_k\rangle \cdot \Phi$. Then*

1. *If there exists an input assignment $\vec{\alpha}$ such that $\vec{\alpha}(pc_k \wedge \neg\phi) \rightarrow \texttt{true}$ with $\Sigma_0 \circ \vec{\alpha} \rightarrow \Delta_0'$ for some $\phi \in \Phi$, then $\langle C_0, \Delta_0'\rangle \rightarrow^i \langle C_i, \Delta_i'\rangle$ and C_i contains a failed cast under Δ_i' for some $0 \leq i < k$.*
2. *If there exists an input $\vec{\alpha}$ such that $\vec{\alpha}(pc_0) \rightarrow \texttt{true}$ and $\Sigma_0 \circ \vec{\alpha} \rightarrow \Delta_0'$ with $\langle C_0, \Delta_0'\rangle \rightarrow^k \langle C_k, \Delta_k'\rangle$ where C_k contains a failed cast under Δ_k', then $\vec{\alpha}(pc_k \wedge \neg\phi) \rightarrow \texttt{true}$ for some $\phi \in \Phi$.*

4.4 Examples

Division by Zero. As an example, we show an active checker for division by zero errors. First, we define the refinement type `notzero` as the set of values x of type `Int` such that $x \neq 0$. Then, we can define the type of the division operator `div` as taking an `Int` and a `notzero` argument. Finally, we need a semantic implementation function for division, which here is the standard division function for integers. These are shown in Figure 7.

Now, in Figure 8 we show an example of active checking a small program for division by zero errors. On the left, we see the original program before type-checking and compilation. Now we wish to perform static type checking and typecast insertion. Because `div` is an operator, at line 4, by the type rule T-OP, we must show that the type of x_1 is a subtype of `notzero` to prove that the program is well-typed. Therefore, during compilation, we either statically prove this is the case, have our algorithmic implication judgment time out, or else statically reject this program as ill-typed. For the purpose of this example, we assume that the algorithmic implication judgment times out, and so we add a cast to the type $\langle\texttt{notzero}\rangle$. The middle column shows the resulting code after cast insertion. Finally, suppose the code is run on the concrete input $\alpha = 2$. The column on the right shows pc and Φ after each command is evaluated. After the first run, we then extract $\alpha \neq 0$ from Φ and query $(\alpha > -5) \wedge \neg(\alpha \neq 0)$ to a constraint solver to actively check for an input that causes division by zero.

Integer overflow/underflow. Integer overflow and underflow and related machine arithmetic errors are a frequent cause of security-critical bugs; the May 2007 MITRE survey of security bugs reported notes that integer overflows were the second most common bug in operating system vendor advisories [9]. Brumley et al. describe a system of dynamic checks called RICH that catches machine arithmetic errors at run-time [5]. The basic idea is to define upper and lower bounds for signed and unsigned integer types, then insert a check whenever a potentially *unsafe assignment* is carried out.

We can capture these checks with the refinement types in Figure 9. Given a type environment Γ_{int} with these types, compilation

1. $(\emptyset)x_2 := \langle \texttt{int} \rangle 52$	$(\emptyset)x_2 := \langle \texttt{int} \rangle 52$	$pc = \texttt{true}, \Phi = \emptyset$
2. $(\emptyset)$ `if` $(x_1 > -5)$	$(\emptyset)$`if` $(x_1 > -5)$	$pc = \alpha > -5, \Phi = \emptyset$
3. `then` $(\emptyset)x_3 := \langle \texttt{int} \rangle \texttt{div}(x_2, x_1)$	`then` $((x_1, \texttt{notzero}))x_3 := \langle \texttt{int} \rangle \texttt{div}(x_2, x_1)$	$pc = \alpha > -5, \Phi = \{\alpha \neq 0\}$

Figure 8. A program fragment with an active checker for division by zero and $\Sigma(x_1) = \alpha$. The left column shows the original. The middle column shows the program after compilation. The right column shows the path condition pc and the set Φ for $\alpha = 2$.

`int`	=	$\{x : \texttt{Int} \mid \texttt{true}\}$
`uint64_t`	=	$\{x : \texttt{Int} \mid 0 \leq x < 2^{64}\}$
`int64_t`	=	$\{x : \texttt{Int} \mid -2^{63} \leq x < 2^{63}\}$
`uint32_t`	=	$\{x : \texttt{Int} \mid 0 \leq x < 2^{32}\}$
`int32_t`	=	$\{x : \texttt{Int} \mid -2^{31} \leq x < 2^{31}\}$
`uint16_t`	=	$\{x : \texttt{Int} \mid 0 \leq x < 2^{16}\}$
`int16_t`	=	$\{x : \texttt{Int} \mid -2^{15} \leq x < 2^{15}\}$
`uint8_t`	=	$\{x : \texttt{Int} \mid 0 \leq x < 2^{8}\}$
`int8_t`	=	$\{x : \texttt{Int} \mid -2^{7} \leq x < 2^{7}\}$

Figure 9. Types for an integer overflow/underflow checker.

will automatically insert casts for assignments between variables of different types. For example, suppose that a variable x has type `uint8_t` . Then an assignment of an expression e to x may insert a cast $\langle$`uint8_t` $\rangle$ if the algorithmic implication judgment cannot decide statically whether the value of e falls within the bounds of `uint8_t` or not. At run-time, the cast $\langle$`uint8_t` $\rangle e$ checks that the concrete value of the expression e is between 0 and 2^8, just as in the RICH system. If the run-time check succeeds, then we symbolically evaluate e to obtain a symbolic expression sv_e, then add the expression $(0 \leq sv_e < 2^8)$ to the set of checkers ch. We can then query $pc \wedge \neg(0 \leq sv_e < 2^8)$ to a constraint solver and solve for an input that violates the type bounds.

4.5 Combining Checkers

Finally, we turn our attention to the question of *combining* checkers for different integer refinement properties. We begin with the definition of what it means for one environment to be a *restriction* of another.

Definition 3. *(Environment restriction.) We say that $\Gamma <: \Gamma'$ if the following two conditions hold :*

1. *If $\Gamma \vdash x_i : S_i$, and $\Gamma' \vdash x_i : S'_i$ for some x_i, then $S_i <: S'_i$*
2. *If $\Gamma \vdash \texttt{op}(T_1 \ldots T_n) : T$ and $\Gamma' \vdash \texttt{op}(T'_1 \ldots T'_n) : T$ for some operator `op`, then $T_i <: T'_i$ for $1 \leq i \leq n$.*

We would like to ensure that a restricted environment gives us more checking. In particular, if we use two environments Γ and Γ' where $\Gamma <: \Gamma'$ to insert casts into the same program, we would like to ensure that Γ does not "disable" the casts inserted by Γ'. The following lemma states this formally.

Lemma 1. *(Monotonicity.) If $\Gamma <: \Gamma'$, then the following properties hold:*

1. *If $\Gamma \vdash C$ and $\Delta \in \Gamma$, then $\Gamma' \vdash C$ and $\Delta \in \Gamma'$.*
2. *Let $\Gamma \vdash C_0 \Rightarrow C'_0$ and $\Gamma' \vdash C_0 \Rightarrow C''_0$. Let Δ_0 be a concrete store such that $\Delta_0 \in \Gamma$. Then if $\langle C'_0, \Delta_0 \rangle \rightarrow^k \langle C'_k, \Delta_k \rangle$, and C'_k contains a failed cast under Δ_k, then $\langle C''_0, \Delta_0 \rangle \rightarrow^i \langle C''_i, \Delta_i \rangle$ and C''_i contains a failed cast under Δ_i, for some $0 \leq i \leq k$.*

Suppose we want to check both integer overflow and division by zero simultaneously, and we have existing type environments Γ_{int} and Γ_{div} that check each property individually. By this we mean that we want to construct a new type environment Γ such that if a program compiled with Γ_{int} fails on some input, then the same program compiled with Γ fails on that input, and similarly for Γ_{div}. The following lemma says that the intersection $\Gamma = \Gamma_{int} \cap \Gamma_{div}$ is a restriction of both Γ_{int} and Γ_{div}.

Lemma 2. *(Intersection is restriction.) Let $\Gamma = \Gamma_1 \cap \Gamma_2$. Then $\Gamma <: \Gamma_1$ and $\Gamma <: \Gamma_2$.*

Together with the monotonicity property, this gives us our desired result. Specifically, to check both properties, we simply compile with the intersection environment, giving us the following theorem:

Theorem 9. *(Combination.) Let Γ_a and Γ_b be type environments, and let $\Gamma = \Gamma_a \cap \Gamma_b$. Let C be a program and let $\Gamma_a \vdash C \Rightarrow C_a$, and $\Gamma_b \vdash C \Rightarrow C_b$, and $\Gamma \vdash C \Rightarrow C'_0$. Let Δ_0 be a concrete store such that $\Delta_0 \in \Gamma$. If $\langle C_0, \Delta_0 \rangle \rightarrow^k \langle C_k, \Delta_k \rangle$ and C_k contains a failed cast under Δ_k, where C_0 is either C_a or C_b, then $\langle C'_0, \Delta_0 \rangle \rightarrow^i \langle C'_i, \Delta_i \rangle$, and C'_i contains a failed cast under Δ_i, for $0 \leq i \leq k$.*

5. Optimizations

Active checkers can be viewed as injecting additional constraints in path constraints in order to refine the partitioning on input values. In practice, active checkers may inject *many* such constraints all over program executions, making path explosion even worse than with path exploration alone. In this section, we discuss several optimizations necessary to make active checking tractable in practice.

5.1 Minimizing Calls to the Constraint Solver

As discussed in Section 3, (negations of) constraints injected by various active checkers in a same path constraint can be solved independently one-by-one since they have no side effects. We call this a *naive combination* of checker constraints.

However, the number of calls to the constraint solver can be reduced by bundling together constraints injected at the same or equivalent program states into a single conjunction. If pc denotes the path constraint for a given program state, and $\phi_{C1}, \ldots, \phi_{Cn}$ are a set of constraints injected in that state by each of the active checkers, we can define the combination of these active checkers by injecting the formula $\phi_C = \phi_{C1} \wedge \cdots \wedge \phi_{Cn}$ in the path constraint, which will result in the single query $pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn})$ to the constraint solver. We can also bundle in the same conjunction constraints ϕ_{Ci} injected by active checkers at different program states anywhere in between two conditional statements, i.e., anywhere between two constraints in the path constraint (since those program states are indistinguishable by that path constraint). This combination reduces the number of calls to the constraint solver but, if the query $pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn})$ is satisfied, a satisfying assignment produced by the constraint solver may not satisfy *all* the disjuncts, i.e., may violate only *some* of the properties being checked. Hence, we call this a *weakly-sound combination*.

A *strongly-sound*, or *sound* for short, combination can be obtained by making additional calls to the constraint solver using the simple function shown in Figure 10. By calling

`CombineActiveCheckers`$(\emptyset, pc, \phi_{C1}, \ldots, \phi_{Cn})$

the function returns a set I of input values that covers *all* the disjuncts that are satisfiable in the formula $pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn})$. The function first queries the solver with the disjunction of

Procedure `CombineActiveCheckers`($I, pc, \phi_{C1}, \ldots, \phi_{Cn}$):
1. Let $x =$ `Solve`$(pc \wedge (\neg\phi_{C1} \vee \cdots \vee \neg\phi_{Cn}))$
2. If $x =$ `UNSAT` return I
3. For all i in $[1, n]$, eliminate ϕ_{Ci} if x satisfies $\neg\phi_{Ci}$
4. Let $\phi_{C1}, \ldots, \phi_{Cm}$ denote the remaining ϕ_{Ci} ($m < n$)
5. If $m = 0$, return $I \cup \{x\}$
6. Call `CombineActiveCheckers`($I \cup \{x\}, pc, \phi_{C1}, \ldots, \phi_{Cm}$)

Figure 10. Function to compute a strongly-sound combination of active checkers.

all the checker constraints (line 1). If the solver returns `UNSAT`, we know that all these constraints are unsatisfiable (line 2). Otherwise, we check the solution x returned by the constraint solver against each checker constraint to determine which are satisfied by solution x (line 3). (This is a model-checking check, not a satisfiability check; in practice, this can be implemented by calling the constraint solver with the formula $\bigwedge_i (b_i \Leftrightarrow \neg\phi_{Ci}) \wedge pc \wedge (\bigvee_i b_i)$ where b_i is a fresh boolean variable which evaluates to `true` iff $\neg\phi_{Ci}$ is satisfied by a satisfying assignment x returned by the constraint solver; determining which checker constraints are satisfied by x can then be done by looking up the values of the corresponding bits b_i in solution x.) Then, we remove these checker constraints from the disjunction (line 4), and query the solver again until all checker constraints that can be satisfied have been satisfied by some input value in I. If t out of the n checkers can be satisfied in conjunction with the path constraint pc, this function requires at most `min`$(t + 1, n)$ calls to the constraint solver, because each call removes at least one checker from consideration. Obtaining strong soundness with fewer than t calls to the constraint solver is not possible in the worse case. Note that the naive combination defined above is strongly-sound, but always requires n calls to the constraint solver.

It is worth emphasizing that none of these combination strategies attempt to minimize the number of input values (solutions) needed to cover all the satisfiable disjuncts. This could be done by querying first the constraint solver with the *conjunction* of all checker constraints to check whether any solution satisfies all these constraints simultaneously, i.e., to check whether their intersection is non-empty. Otherwise, one could then iteratively query the solver with smaller and smaller conjunctions to force the solver to return a minimum set of satisfying assignments that cover all the checker constraints. Unfortunately, this procedure may require in the worse case $O(2^n)$ calls to the constraint solver. (The problem can be shown to be NP-complete by a reduction from the NP-hard SET-COVER problem.)

Weakly and strongly sound combinations capture possible overlaps, inconsistencies or redundancies between active checkers at equivalent program states, but is independent of how each checker is specified: it can be applied to any active checker that injects a formula at a given program state. Also, the above definition is independent of the specific reasoning capability of the constraint solver. In particular, the constraint solver may or may not be able to reason precisely about combined theories (abstract domains and decision procedures) obtained by combining individual constraints injected by different active checkers. Any level of precision is acceptable with our framework and is an orthogonal issue (e.g., see [16]).

5.2 Minimizing Formulas

Minimizing the number of calls to the constraint solver should not be done at the expense of using longer formulas. Fortunately, the above strategies for combining constraints injected by active checkers also reduce formula sizes.

```
1 #define k 100 // constant
2 void Q(int *x, int a[k]){ // inputs
3   int tmp1,tmp2,i;
4   if (x == NULL) return;
5   for (i=0; i<=k;i++) {
6     if (a[i]>0) tmp1 = tmp2+*x;
7     else tmp2 = tmp1+*x;
8   }
9   return;
10}
```

Figure 11. A program with $O(2^k)$ possible execution paths. A naive application of a NULL dereference active checker results in $O(k \cdot 2^k)$ additional calls to the constraint solver, while local constraint caching eliminates the need for any additional calls to the constraint solver.

For instance, consider a path constraint pc and a set of n constraints $\phi_{C1} \ldots \phi_{Cn}$ to be injected at the end of pc. The naive combination makes n calls to the constraint solver, each with a formula of length $|pc| + |\phi_{Ci}|$, for all $1 \leq i \leq n$. In contrast, the weak combination makes only a single call to the constraint solver with a formula of size $|pc| + \Sigma_{1 \leq i \leq n} |\phi_{Ci}|$, i.e., a formula (typically much) smaller than the sum of the formula sizes with the naive combination. The strong combination makes, in the worse case, n calls to the constraint solver with formulas of size $|pc| + \Sigma_{1 \leq i \leq j} |\phi_{Ci}|$ for all $1 \leq j \leq n$, i.e., possibly bigger formulas than the naive combination. But often, the strong combination makes fewer calls than the naive combination, and matches the weak combination in the best case (when none of the disjuncts $\neg\phi_{Ci}$ are satisfiable).

In practice, path constraints pc tend to be long, much longer than injected constraints ϕ_{Ci}. A simple optimization (implemented in [14, 7, 31, 15]) consists of eliminating the constraints in pc which do not share symbolic variables (including by transitivity) with the negated constraint c to be satisfied. satisfied. This *unrelated constraint elimination* can be done syntactically by constructing an undirected graph G with one node per constraint in $pc \cup \{c\}$ and one node per symbolic (input) variable such that there is an edge between a constraint and a variable iff the variable appears in the constraint. Then, starting from the node corresponding to constraint c, one performs a (linear-time) traversal of the graph to determine with constraints c' in pc are reachable from c in G. At the end of the traversal, only the constraints c' that have been visited are kept in the conjunction sent to the constraint solver, while the others are eliminated.

With unrelated constraint elimination and the naive checker combination, the size of the reduced path constraint pc_i may vary when computed starting from each of the n constraints ϕ_{Ci} injected by the active checkers. In this case, n calls to the constraint solver are made with the formulas $pc_i \wedge \neg\phi_{Ci}$, for all $1 \leq i \leq n$. In contrast, the weak combination makes a single call to the constraint solver with the formula $pc' \wedge (\vee_i \neg\phi_{Ci})$ where pc' denotes the reduced path constraint computed when starting with the constraint $\vee_i \neg\phi_{Ci}$. It is easy to see that $|pc'| \leq \Sigma_i |pc_i|$, and therefore that the formula used with the weak combination is again smaller than the sum of the formula sizes used with the naive combination. Loosely speaking, the strong combination includes again both the naive and weak combinations as two possible extremes.

5.3 Caching Strategies

No matter what strategy is used for combining checkers at a single program point, constraint *caching* can significantly reduce the overhead of using active checkers.

To illustrate the benefits of constraint caching, consider a NULL dereference active checker (see Section 3) and the program Q in Figure 11. Program Q has $2^k + 1$ executions, where 2^k of those dereference the input pointer `x` k times each. A naive approach to dynamic test generation with a NULL dereference active checker would inject k constraints of the form `x` $\neq$ `NULL` at each dereference of `*x` during every such execution of Q, which would result in a total of $k \cdot 2^k$ additional calls to the constraint solver (i.e., k calls for each of those executions).

To limit this expensive number of calls to the constraint solver, a first optimization consists of *locally caching* constraints in the current path constraint in such a way that syntactically identical constraints are never injected more than once in any path constraint. (Remember path constraints are simply conjunctions.) This optimization is applicable to any path constraint, with or without active checkers. The correctness of this optimization is based on the following observation: if a constraint c is added to a path constraint pc, then for any longer pc' extending pc, we have $pc' \Rightarrow pc$ (where $\Rightarrow$ denotes logical implication) and $pc' \wedge \neg c$ will always be unsatisfiable because c is in pc'. In other words, adding the same constraint multiple times in a path constraint is pointless since only the negation of its first occurrence has a chance to be satisfiable.

Constraints generated by active checkers can be dealt with by injecting those in the path constraint like regular constraints. Indeed, for any constraint c injected by an active checker either at the end of a path constraint pc or at the end of a longer path constraint pc' (i.e., such that $pc' \Rightarrow pc$), we have the following:

- if $pc \wedge \neg c$ is unsatisfiable, then $pc' \wedge \neg c$ is unsatisfiable;
- conversely, if $pc' \wedge \neg c$ is satisfiable, then $pc \wedge \neg c$ is satisfiable (and has the same solution).

Therefore, we can check $\neg c$ as early as possible, i.e., in conjunction with the shorter pc, by inserting the first occurrence of c in the path constraint. If an active checker injects the same constraint later in the path constraint, local caching will simply remove this second redundant occurrence.

By injecting constraints generated by active checkers into regular path constraints and by using local caching, a given constraint c, like `x` $\neq$ `NULL` in our previous example, will appear at most once in each path constraint, and a single call to the constraint solver will be made to check its satisfiability for each path, instead of k calls as with the naive approach without local caching. Moreover, because the constraint `x` $\neq$ `NULL` already appears in the path constraint due to the `if` statement on line 4 before any pointer dereference `*x` on lines 6 or 7, it will never be added again to the path constraint with local caching, and no additional calls will be made to the constraint solver due to the NULL pointer dereference active checker for this example.

Another optimization consists of caching constraints *globally* [7]: whenever the constraint solver is called with a query, this query and its result are kept in a (hash) table shared between execution paths during a directed search. In Section 7, the effect of both local and global caching is measured empirically.

6. Active Checkers in SAGE

We implemented active checkers as part of a dynamic test generation tool called SAGE (*Scalable, Automated, Guided Execution*) [15]. SAGE uses the `iDNA` tool [4] to trace executions of Windows programs, then virtually re-executes these traces with the `TruScan` trace replay framework [28]. During re-execution, SAGE checks for file read operations and marks the resulting bytes as symbolic. As re-execution progresses, SAGE generates symbolic constraints for the path constraint. After re-execution completes, SAGE uses the constraint solver `Disolver` [19] to generate new input values that will drive the program down new paths. SAGE then completes this cycle by testing and tracing the program on the newly generated inputs. The new execution traces obtained from those new inputs are sorted by the number of new code block they discover, and the highest ranked trace is expanded next to generate new test inputs and repeat the cycle [15]. Note that SAGE does not perform any static analysis.

Number	Checker	Number	Checker
0	Path Exploration	7	Integer Underflow
1	DivByZero	8	Integer Overflow
2	NULL Deref	9	MOVSX Underflow
3	SAL NotNull	10	MOVSX Overflow
4	Array Underflow	11	Stack Smash
5	Array Overflow	12	AllocArg Underflow
6	REP Range	13	AllocArg Overflow

Figure 12. Active checkers implemented in SAGE.

The experiments reported in the next section were performed with 13 active checkers, shown in Figure 12 with an identifying number. Zero denotes a regular constraint in a path constraint. Number 1 refers to a division-by-zero checker, 2 denotes a NULL pointer dereference checker, and 4 and 5 denote array underflow and overflow checkers (see Section 3). Number 3 refers to an active checker that looks for function arguments that have been annotated with the `notnull` attribute in the SAL property language [17], and attempts to force those to be NULL. Checker type 6 looks for the x86 `REP MOVS` instruction, which copies a range of bytes to a different range of bytes, and attempts to force a condition where the ranges overlap, causing unpredictable behavior. Checkers 6 and 7 are for integer underflows and overflows (see Section 4). Checkers type 8 and 9 target the `MOVSX` instruction, which sign-extends its argument and may lead to loading a very large value if the argument is negative. The "stack smash" checker, type 11, attempts to solve for an input that directly overwrites the stack return pointer, given a pointer dereference that depends on a symbolic input. Finally, checkers type 12 and 13 look for heap allocation functions with symbolic arguments; if found, they attempt to cause overflow or underflow of these arguments.

An active checker in SAGE first registers a `TruScan` callback for specific events that occur during re-execution. For example, an active checker can register a callback that fires each time a symbolic input is used as an address for a memory operation. The callback then inspects the concrete and symbolic state of the re-execution and decides whether or not to emit an active checker constraint. If the callback does emit such a constraint, SAGE stores it in the current path constraint.

SAGE implements a *generational search* [15]: given a path constraint, all the constraints in that path are systematically negated one-by-one, placed in a conjunction with the prefix of the path constraint leading it, and attempted to be solved with the constraint solver. Constraints injected by active checkers are inserted in the path constraint and treated as regular constraints during a generational search.

Because we work with x86 machine-code traces, some information we would like to use as part of our active checkers is not immediately available. For example, when SAGE observes a `load` instruction with a symbolic offset during re-execution, it is not clear what the bound should be for the offset. We work around these limitations by leveraging the `TruScan` infrastructure. During re-execution, `TruScan` observes calls to known allocator functions. By parsing the arguments to these calls and their return values, as well as detecting the current stack frame, `TruScan` builds a map from each concrete memory address to the bounds of the containing memory object. We use the bounds associated with the memory

Media 1	none	weak	strong	naive
Total Time (s)	16	37	42	37
Solver Time (s)	5	5	10	5
# Tests Gen	59	70	87	105
# Disjunctions	N/A	11	11	N/A
Dis. Min/Mean/Max	N/A	2/4.2/16	2/4.2/16	N/A
# Path Constr.	67	67	67	67
# Checker Constr.	N/A	46	46	46
# Solver Calls	67	78	96	113
Max CtrList Size	77	141	141	141
Mean CtrList Size	2.7	2.7	2.7	3
Local Cache Hit	79%	81%	88%	88%
Media 2	none	weak	strong	naive
Total Time (s)	761	973	1140	1226
Solver Time (s)	421	463	601	504
# Tests Gen	1117	1833	2734	5122
# Disjunctions	N/A	1125	1125	N/A
Dis. Min/Mean/Max	N/A	1/5.4/216	1/5.4/216	N/A
# Path Constr.	3001	2990	2990	2990
# Checker Constr.	N/A	6080	6080	6080
# Solver Calls	3001	4115	5368	9070
Max CtrList Size	11141	91739	91739	91739
Mean CtrList Size	368	373	373	372
Local Cache Hit	39%	19.5%	19.5%	19.5%

Figure 14. Microbenchmark statistics.

object pointed to by the concrete value of the address as the upper and lower bound for an active bounds check of the memory access.

7. Evaluation

We report results of experiments obtained with our extension of SAGE with active checkers and two media-parsing applications widely used on Windows. Figure 13 shows the result of a single symbolic execution and test generation task for each of these two test programs. The second column indicates which checkers injected constraints during that program execution. The last column gives the number of symbolic input bytes read during that single execution, which is 100 to 1,000 times larger than previously reported with dynamic test generation [14, 7, 31]. In the technical report [15], we reported various experiments with SAGE and the same test programs among others, but without any active checkers.

For each application, we ran microbenchmarks to quantify the marginal cost of active checking during a single symbolic execution task and measure the effectiveness of our optimizations. We then performed long-running SAGE searches with active checkers to investigate their effectiveness at finding bugs. These searches were performed on a 32-bit Windows Vista machine with two dual-core AMD Opteron 270 processors running at 2 GHz, with 4 GB of RAM and a 230 GB hard drive; all four cores were used in each search. We now describe observations from these experiments. We stress that these observations are from a limited sample size and should be taken with caution.

7.1 Microbenchmarks

Figure 14 presents statistics for our two test programs obtained with a single symbolic execution and test generation task with no active checkers, or the weak, strong and naive combinations of active checkers discussed in Section 5. For each run, we report the total run time (in seconds), the time spent in the constraint solver (in seconds), the number of test generated, the number of disjunctions bundling together checker constraints (if applicable) before calling the constraint solver, the minimum, mean and maximum number of constraints in disjunctions (if applicable[1]), the total number of constraints in the path constraint, the total number of constraints injected by checkers, the number of calls made to the constraint solver, statistics about the size needed to represent all path and checker constraints (discussed further below) and the local cache hit. Each call to the constraint solver was set with a timeout value of 5 seconds.

Checkers produce more test cases than path exploration at a reasonable cost. As expected, using checkers increases total run time but also generates more tests. For example, all checkers with naive combination for Media 2 creates 5122 test cases in 1226 seconds, compared to 1117 test cases in 761 seconds for the case of no active checkers; this gives us 4.5 times as many test cases for 61% more time spent in this case. As expected (see Section 5), the naive combination generates more tests than the strong combination, which itself generates more tests than the weak combination. Perhaps surprisingly, most of the extra time is spent in symbolic execution, not in solving constraints. This may explain why the differences in runtime between the naive, strong and weak cases are relatively not that significant. Out of curiosity, we also ran experiments (not shown here) with a "basic" set of checkers that consisted only of Array Bounds and DivByZero active checkers; this produced fewer test cases, but had little to no runtime penalty for test generation for both test programs.

Weak combination has the lowest overhead. We observed that the solver time for weak combination of disjunctions was the lowest for Media 2 runs with active checkers and tied for lowest with the naive combination for Media 1. The strong disjunction generates more test cases, but surprisingly takes longer than the naive combination in both cases. For Media 1, this is due to the strong combination hitting one more 5-second timeout constraints than the naive combination. For Media 2, we believe this is due to the overhead involved in constructing repeated disjunction queries. Because disjunctions in both cases have fairly few disjuncts on average (4 or 5), this overhead dominates for the strong combination, while the weak one is still able to make progress by handling the entire disjunction in one query.

Unrelated constraint elimination is important for checkers. Our implementation of the unrelated constraint optimization described in Section 5.2 introduces additional *common subexpression variables*. Each of these variable defines a subexpression that appears in more than one constraint. In the worst case, the maximum possible size of a list of constraints passed to our constraint solver is the sum of the number of these variables, plus the size of the path constraint, plus the number of checker constraints injected. We collected the maximum possible constraint list size (Max CtrList Size) and the mean size of constraint lists produced after our unrelated constraint optimization (Mean CtrList Size). The maximum possible size does not depend on our choice of weak, strong, or naive combination, but the mean list size is slightly affected. We observe in the Media 2 microbenchmarks that the maximum possible size jumps dramatically with the addition of checkers, but that the mean size stays almost the same. Furthermore, even in the case without checkers, the mean list size is 100 times smaller than the maximum. The Media 1 case was less dramatic, but still showed post-optimization constraint lists an order of magnitude smaller than the maximum. This shows that unrelated constraint optimization is key to efficiently implement active checkers.

[1] In the strong case, the mean number does not include disjunctions iteratively produced by the algorithm of Figure 10, which explains why the mean is the same as in the weak case.

Test	Checkers Injected	Time (secs)	pc size	# checker constraints	# Tests	# Instr.	Symbolic Input Size
Media 1	0,2,4,5,7,8	37	67	46	105	3795771	65536
Media 2	0,1,2,4,5,7,8,9,10,11	1226	2990	6080	5122	279478553	27335

Figure 13. Statistics from a *single* symbolic execution and test generation task with a naive combination of all 13 checkers. We report the checker types that injected constraints, the total time for symbolic execution test generation, the number of constraints in the total path constraint, the total number of injected checker constraints, the number of tests generated, the number of instructions executed after the first file read, and the number of symbolic input bytes.

Crash Bucket	Kind	0	2	4	5	7	8
1867196225	NULL	No/W/S				W/S	W/S
1867196225	ReadAV	No/W				W	W
1277839407	ReadAV	S					
1061959981	ReadAV		S			S	S
1392730167	ReadAV	S					
1212954973	ReadAV				S	S	S
1246509355	ReadAV				S	W/S	W/S
1527393075	ReadAV	S					
1011628381	ReadAV					S	W/S
2031962117	ReadAV	No/W/S					
286861377	ReadAV	No/S					
842674295	WriteAV		S	S		S	S

Figure 15. Crash buckets found for Media 1 by 10-hours SAGE searches with "No" active checkers, with a "W"eak and "S"trong combinations of active checkers. A total of 41658 tests were generated and tested in 30 hours, with 783 crashing files in 12 buckets.

Crash Bucket	Kind	0
790577684	ReadAV	No/W/S
825233195	ReadAV	No/W/S
795945252	ReadAV	No/W/S
1060863579	ReadAV	No/W

Figure 16. Crash buckets found for Media 2 by 10-hours SAGE searches with "No" active checkers, with a "W"eak and "S"trong combinations of active checkers. A total of 11849 tests were generated and tested in 30 hours, with 25 crashing files in 4 buckets.

Media 1	0	2	4	5	7	8
Injected	27612	26	13	13	11153	11153
Solved	18056	22	2	3	3179	5552
Crashes	339	22	2	3	139	136
Yield	1.9%	100%	100%	100%	4.4%	2.4%

Media 2	0	1	2	4	5
Injected	13425	12	2146	1544	1551
Solved	4735	0	61	10	20
Crashes	7	0	0	0	0
Yield	1.4%	N/A	0%	0%	0%

	7	8	9	10	11
Injected	11158	11177	5	5	10
Solved	576	2355	0	5	0
Crashes	0	0	0	0	0
Yield	0%	0%	N/A	0%	N/A

Figure 17. Constraints injected by checker types, solved, crashes, and yield for Media 1 and Media 2, over both weak and strong combination 10-hours searches.

7.2 Macrobenchmarks

For macrobenchmarks, we ran SAGE search for 10 hours starting from the same initial media file, and generated test cases with no checkers, and with the weak and strong combination of all 13 checkers. We then tested each test case by running the program with AppVerifier [8], configured to check for heap errors. For each crashing test case, we recorded the checker kinds responsible for the constraints that generated the test. Since a SAGE search can generate many different test cases that exhibit the same bug, we "bucket" crashing files by the *stack hash* of the crash, which includes the address of the faulting instruction. We also report a bucket kind, which is either a NULL pointer dereference, a read access violation (ReadAV), or a write access violation (WriteAV). It is possible for the same bug to be reachable by program paths with different stack hashes for the same root cause. Our experiments always report the distinct stack hashes. We also computed the hit rate for global caching during each SAGE search.

Checkers can find bugs missed by path exploration. Figure 15 shows the crash buckets found for Media 2 by 10-hours SAGE searches with "No" active checkers, with a "W"eak and "S"trong combinations of active checkers. For instance, an "S" in a column means that at least one crash in the bucket was found by the search with strong combination. The type of checkers whose constraint found the crash bucket is also indicated in the figure. For Media 1, the Null Deref (type 2) active checker found 2 crash buckets, the Array Underflow and Overflow (types 4 and 5) active checkers found 3 crash buckets, while the Integer Underflow and Overflow (types 7 and 8) active checkers found 7 crash buckets. Without any active checkers, SAGE is able to find only 4 crash buckets in 10 hours of search, and misses the serious WriteAV bug detected by the strong combination only. For Media 2, in contrast, the test cases generated by active checkers did not find any new crash buckets, as shown in Figure 16.

Checker yield can vary widely. Figure 17 reports the overall number of injected constraints of each type during all 10-hours searches, and how many of those were successfully solved to create new test cases. It also reports the checker *yield*, or percentage of test cases that led to crashes. For Media 1, active checkers have a higher yield than test cases generated by path exploration (type 0). For Media 2, several checkers did inject constraints that were solvable, but their yield is 0% as they did not find any new bugs. The yield indicates how precise symbolic execution is. For Media 1, symbolic execution is very precise as every checker constraint violation for checker types 2, 4 and 5 actually leads to a crash (as is the case with a fully sound and complete constraint generation and solving as shown in Section 3); even if symbolic execution is perfect, the yield for the integer under/overflow active checkers may be less than 100% because not every integer under/overflow leads to a crash. In contrast, the symbolic execution for Media 2 seems poor.

Local and global caching are effective. Local caching can remove a significant number of constraints during symbolic execution. For Media 1, we observed a 80% or more local cache hit rate (see Figure 14). For Media 2, the hit rates were less impressive but still removed roughly 20% of the constraints.

Our current SAGE implementation does not have a global cache for query results (see Section 5). To measure the impact of global caching on our macrobenchmark runs, we added code that dumps to disk the SHA-1 hash of each query to the constraint solver, and then computes the global cache hit rate. For Media 1, all searches

showed roughly a 93% hit rate, while for Media 2 we observed 27%. This shows that there are significant redundancies in queries made by different test generation tasks during the same SAGE search.

7.3 Additional Experiences

We have also performed exploratory SAGE searches on several other applications, including two shipped as part of Office 2007 and two media parsing layers. In one of the Office applications and media layer, the division by zero checker and the integer overflow checker each created test cases leading to previously-unknown division by zero errors. In the other cases, we also discovered new bugs in test cases created by checkers, but needed to use an internal tool for runtime passive checking of memory safety violations that is more precise than AppVerifier.

8. Conclusion

The more one checks for property violations, the more one should find software errors. In this paper, we have defined and studied active property checking, a new form of dynamic property checking based on dynamic symbolic execution, constraint solving and test generation. We showed how active type checking extends traditional static and dynamic type checking. We presented several optimizations to implement active property checkers efficiently, and discussed results of experiments with several large shipped Windows applications. Active property checking was able to detect several new bugs in those applications.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 213–227, 1989.

[2] A. Aiken, E.L. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[3] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Tests from Counterexamples. In *Proceedings of ICSE'2004 (26th International Conference on Software Engineering)*. ACM, May 2004.

[4] S. Bhansali, W. Chen, S. De Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments VEE*, 2006.

[5] D. Brumley, T. Chieh, R. Johnson, H. Lin, and D. Song. RICH : Automatically protecting against integer-based vulnerabilities. In *NDSS (Symp. on Network and Distributed System Security)*, 2007.

[6] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.

[8] Microsoft Corporation. AppVerifier, 2007. `http://www.microsoft.com/technet/prodtechnol/windows/appcompatibility/appverifier.mspx`.

[9] MITRE Corporation. Vulnerability type distributions in cve, 22 may 2007., 2007. `http://cve.mitre.org/docs/vuln-trends/index.html`.

[10] C. Csallner and Y. Smaragdakis. Check'n Crash: Combining Static Checking and Testing. In *Proceedings of ICSE'2005 (27th International Conference on Software Engineering)*. ACM, May 2005.

[11] C. Flanagan. Hybrid type checking. In *Proceedings of POPL'2006 (33rd ACM Symposium on Principles of Programming Languages)*, January 2006.

[12] Flexelint. web page: `http://www.gimpel.com/`.

[13] P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of POPL'2007 (34th ACM Symposium on Principles of Programming Languages)*, pages 47–54, Nice, January 2007.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)*, pages 213–223, Chicago, June 2005.

[15] P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. Technical Report MS-TR-2007-58, Microsoft, May 2007. `http://research.microsoft.com/users/pg/public_psfiles/SAGE-external-v1.pdf`.

[16] S. Gulwani and A. Tiwari. Combining Abstract Interpreters. In *Proceedings of PLDI'2006 (ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation)*, Ottawa, June 2006.

[17] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ICSE*, 2006.

[18] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific Static Analyses. In *Proceedings of PLDI'2002 (2002 ACM SIGPLAN Conference on Programming Language Design and Implementation)*, pages 69–82, 2002.

[19] Y. Hamadi. Disolver: the distributed constraint solver version 2.44, 2006. `http://research.microsoft.com/~youssefh/DisolverWeb/disolver.pdf`.

[20] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 125–138, Berkeley, January 1992.

[21] F. Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

[22] P. Joshi, K. Sen, and M. Shlimovich. Predictive testing: Amplifying the effectiveness of software testing. Technical report, UC-Berkeley, April 2007. UCB/EECS-2007-35.

[23] J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.

[24] Klocwork. web page: `http://klocwork.com/index.asp`.

[25] B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.

[26] E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. In *Proceedings of 12th USENIX Security Symposium*, Washington D.C., August 2003.

[27] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[28] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Languages Design and Implementation (PLDI)*, 2007.

[29] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[30] Polyspace. web page: `http://www.polyspace.com`.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of FSE'2005 (13th International Symposium on the Foundations of Software Engineering)*, Lisbon, September 2005.

[32] S. Thatte. Quasi-static typing. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 367–381, 1990.

[33] W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of ACM SIGSOFT ISSTA'04 (International Symposium on Software Testing and Analysis)*, Boston, July 2004.