Statically-Directed Dynamic Automated Test Generation[*]

Domagoj Babić Lorenzo Martignoni Stephen McCamant Dawn Song
University of California, Berkeley
{babic, martigno, smcc, dawnsong}@cs.berkeley.edu

ABSTRACT

We present a new technique for exploiting static analysis to guide dynamic automated test generation for binary programs, prioritizing the paths to be explored. Our technique is a three-stage process, which alternates dynamic and static analysis. In the first stage, we run dynamic analysis with a small number of seed tests to resolve indirect jumps in the binary code and build a visibly pushdown automaton (VPA) reflecting the global control-flow of the program. Further, we augment the computed VPA with statically computable jumps not executed by the seed tests. In the second stage, we apply static analysis to the inferred automaton to find potential vulnerabilities, i.e., targets for the dynamic analysis. In the third stage, we use the results of the prior phases to assign weights to VPA edges. Our symbolic-execution based automated test generation tool then uses the weighted shortest-path lengths in the VPA to direct its exploration to the target potential vulnerabilities. Preliminary experiments on a suite of benchmarks extracted from real applications show that static analysis allows exploration to reach vulnerabilities it otherwise would not, and the generated test inputs prove that the static warnings indicate true positives.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Reliability, Security

Keywords

automated testing, static analysis, dynamic analysis, prioritization

1. INTRODUCTION

Many techniques have been proposed for software verification and test generation: symbolic model checking [12, 5], explicit-state model checking [25], static analysis [14], directed automated random testing [21], fuzzing [6], and numerous variants or combinations of those. Each technique offers different tradeoffs between soundness, completeness, speed, precision, scalability, and the level of automation, but all of them face the same fundamental problem — state-space explosion.

*This work was partially supported by the NSF under grants 0832943, 0842694, 0842695, 0831501, and 0424422, by the AFRL under grant P010071555, and by the MURI program under ONR grant N000140911081 and AFOSR grants FA9550-08-1-0352 and FA9550-09-1-0539. The first author is also supported by the NSERC (Canada) PDF fellowship.

ISSTA '11, July 17–21, 2011, Toronto, ON, Canada

The first problem we explore is focusing the search during automated test generation. Search heuristics are either look-ahead techniques, which use an analysis cheaper than the search to explore the yet unvisited parts of the search-space, or look-back techniques, which analyze the already explored search-space. Our approach is a look-ahead technique, as we use static analysis to identify possible vulnerabilities and to compute other useful information about identified vulnerabilities, like data-flow slices (e.g., [47, 45]) with respect to the statement triggering the vulnerability. The identified vulnerabilities become targets for the dynamic analysis, while slices are used to guide the search toward those targets.

The second problem we explore is a combination of static and dynamic analysis, with the goals of (1) using the static analysis to guide the dynamic test generation, and (2) using the dynamic analysis to filter out false positives produced by the static analysis. We refer to our approach to this combination as *statically-directed dynamic automated test generation*.

The idea of guiding dynamic with static analysis can be applied at the source level, but our work focuses on test generation for stripped binary programs (without symbol tables or debug information). Our approach is also applicable when the source code of the application is available, but the third-party libraries are closed-source. In that case, the system integrators or even end-users have to test closed-source components before deploying them, especially in settings where security and reliability are paramount.

Working at the binary level introduces some unique challenges: pervasive address arithmetic, the absence of type information, misaligned memory accesses, and indirect (computed) jumps. To address these challenges, we devise a three-stage approach: The first stage uses a combined dynamic and static analysis to generate an interprocedural control-flow graph, represented as a visibly pushdown automaton (VPA) [1]. The second stage performs static analysis on the VPA to identify possible vulnerabilities and to compute other information used later to guide the search, like data-flow dependency slices. The third stage uses symbolic execution [28] and the results of the static analysis from the second stage to generate tests that trigger the vulnerabilities detected by static analysis.

1.1 Contributions

Our three-stage sandwich approach to binary analysis is a new point in the space of tradeoffs (soundness, completeness, speed, precision,...) in binary analysis. In addition to proposing the approach, we make contributions in each stage.

The first stage of our approach, described in Section 2, constructs an underapproximation of the interprocedural control-flow graphs of a binary without symbols or debug information. We combine dynamic and static analysis to resolve indirect jumps and to decode assembly instructions. Both problems are a challenge, because in the absence of type information or labels, every assembly instruction a potential jump target; with variable-sized CISC instructions, it can be difficult to even distinguish code from data. Our approach computes an underapproximation of the transition relation by resolving some indirect jumps with a set of seed tests, and augmenting the computed relation with statically computed direct jumps.

The VPA computed by the first stage is the input to our static analysis (Section 3), designed for finding potentially exploitable vulnerabilities, like buffer overflows. Our static analysis is inspired by the Balakrishnan and Reps's VSA analysis [3]. One difference is that our static analysis combines discovery of abstract memory locations with the data-flow analysis, while VSA alternates discovery and data-flow analysis until a fixed-point. Further, our analysis handles overlapping (mis)aligned reads and writes more precisely, a feature we found particularly useful in analysis of programs that perform a lot of string operations.

The third stage of our approach, presented in Section 4, performs single-path symbolic execution to search for concrete test inputs that trigger potential vulnerabilities discovered by the static analysis in the second stage. Running on its own, the test generation tool can create many program inputs, each exercising a different path, but often none of them trigger a vulnerability. This motivates our use of static analysis to guide the path search.

Thus, we propose a new heuristic for guiding dynamic search towards finding the right path towards a vulnerability identified by static analysis in the second stage. Our heuristic works differently when the dynamic search is within a strongly connected component (SCC) such as a loop, and when it is not. For outside an SCC, we develop a two-component heuristic that ranks states according to two statically-computed metrics, one based on the shortest VPA path to the vulnerability, and the other using the vulnerability's data-flow slice. For within an SCC, we develop a heuristic that picks patterns of paths through the SCC according to a geometric distribution and then alternates those paths over multiple iterations.

The problem of computing the shortest paths on a VPA has not (to our knowledge) been previously posed. We address it by an efficient algorithm that treats well-matched paths first, bottom-up in a call graph, and then extends to all VPA paths by taking an automaton product with an automaton that prohibits mismatched calls. These transformations allow us to use the classic algorithm for shortest paths in a graph. Our algorithm can also be seen as a more efficient case of a weighted pushdown problem [39], using the property that the semiring of path lengths is totally ordered.

In our experiments, the proposed heuristic sped up the dynamic test generation significantly, in one case letting our system find in 11.3 seconds a vulnerability it could otherwise not find even after 6 hours. Both parts of the heuristic were valuable: the SCC part allowed the tool to achieve good coverage of loops with complex internal structure, while the non-SCC part allowed it to efficiently focus on those parts of a program relevant to a vulnerability.

1.2 Terminology

An interprocedural control-flow graph, as computed by the first stage of our analysis, can be seen as the combination of a call graph and separate control-flow graphs for each function. But we instead formalize it as a single unified object, a visibly-pushdown automaton [1]. Though ultimately equivalent, this choice allows our exposition to be at once simpler and more formal.

States represent basic blocks: a sequence of program statements ending with a branch, call, or return. The flow of control enters a basic block only at its beginning and leaves it by execution of the last statement. For simplicity, we will assume that each basic block ending with an unconditional jump is merged with the target block, duplicating blocks that are a target of multiple jumps.[1] Basic blocks containing exit statements are accepting states. The VPAs considered in this paper accept only matched returns words, meaning that every return must have a matching call. If also every call must have a matching return, we say such words are well-matched.

We use the standard abstract interpretation notation and terminology [14]: abstraction (resp. concretization) operator α (resp. γ), abstract post state transformer $post^{\#}$, and join (resp. widening) operator $\sqcup$ (resp. ∇).

2. CONTROL-FLOW COMPUTATION

The first stage of our approach resolves indirect jump targets in binaries by analyzing a set of concrete traces obtained by executing the analyzed application on a set of seed tests. Our dynamic analysis folds multiple traces into a single VPA representing the global flow of control. Additionally, we augment the VPA with missing transitions that can be precisely computed through static analysis. In this section, we describe the computation of the VPA through a combination of dynamic and static analysis.

We adopt a hybrid approach to VPA construction combining dynamic and static analysis. The dynamic analysis attempts to resolve as many targets of indirect control-flow transitions as possible. The static analysis mitigates the incompleteness of the dynamic analysis. For the purpose of finding a vulnerability in the program, it is sufficient that the model of the program we build includes the vulnerable path, but it is irrelevant whether the edges along the path were discovered through seed tests or static augmentation.

We use the PIN [31] framework to instrument binary applications with callbacks at jump and library load events. The instrumented application executes normally, unaware of the callbacks. Traces from multiple seed tests are merged into a single VPA. An underapproximation of the instrumented application's control-flow in the form of a VPA can be computed as follows. Each visited basic block is represented with a single state in VPA. Each visited edge is classified in one of the classes (call, internal, return) and the VPA is updated so that the source and the target basic blocks are linked with an edge of the appropriate class. All basic blocks ending with the `exit` system call are declared to be final states.

After the dynamic VPA construction, we use recursive traversal disassembly [42] to augment VPA with the conditional direct jumps not executed by the seed tests. The resulting VPA is, necessarily, an underapproximation of the complete interprocedural control-flow graph of the analyzed application. The completeness of the constructed VPA depends on the capability of the seed tests to exercise indirect branches and the number of branches that can be accurately resolved with the recursive traversal disassembly. Finally, we use the Vine library from BitBlaze [44] to decode the instructions into a simplified internal representation to facilitate later analysis.

3. STATIC ANALYSIS OF BINARIES

The static analysis in the second stage of our approach performs an interprocedural context- and flow-sensitive analysis on the VPA computed in the first stage. First, static analysis identifies possible vulnerabilities, which are used as targets for the dynamic analysis in the third stage of our approach. Second, static analysis computes approximate size of stack frames and allocated heap regions. These sizes are used to detect out-of-bounds accesses. Third, static analysis maintains a map between written abstract locations and statements in the assembly code and this map is then used to compute

[1] Merging of targets of unconditional jumps is done in the first stage of our approach.

the backward data-flow slice with respect to the identified vulnerability. The slice is used as a component of the guidance heuristics in the third stage of our approach. The rest of this section discusses the most important aspects of our static analysis: the abstract domains used, the treatment of weak and strong updates, and the handling of overlapping and misaligned reads and writes.

3.1 Abstract Domains

The abstract domain we use for static analysis of binaries consists of several hierarchically composed domains: strided intervals (denoted **SInterval**) [37, 2], value map (ValMap), regions (Region), and abstract states (State). We introduce these abstract domains starting at the bottom of the hierarchy.

Strided intervals are defined as a triple $s[lb, ub]$, such that[2]

$$\gamma(s[lb,ub]) := \{i \mid lb \leq i < ub \wedge s > 0 \wedge \exists k \in \mathbb{Z} \,.\, i = s \cdot k\}$$

Binary code often uses indirect addressing $base + index \times scale$, conveniently representable by strided intervals. Arithmetic and logical operations on strided intervals are very similar to operations on simple intervals and can be efficiently computed (e.g., [46]).

Memory regions, in our work, abstract disjoint chunks of memory. Each region has a unique identifier. The set of identifiers is denoted **RegionID**, while for the elements of the set we will use r, with indices. We treat registers as a special form of a fixed-size region (*RegId*). Global variables also form a fixed-size region (*GlbId*). For simplicity, we also place constants and scalar ranges in a special region (*CId*).

We create a unique region for each allocation site that allocates a heap object, while for the other types of regions (registers, stack, and globals) we create one unique region per program per region type. Other abstract domain design decisions are possible. For instance, Balakrishnan and Reps [3] allocate one stack region per function. Our decision to keep a single stack region per program simplifies context-sensitive analysis and allows precise handling of writes to any frame on the calling stack.

A value can be either an integer or an address within a region. To represent a pointer that could point to multiple regions, or a value that could be either a constant or a pointer, we use value maps ValMap := **RegionID** → **SInterval**, where **SInterval** represents the offset within the **RegionID** region. Integers are represented as offsets within the distinguished region *CId*. Further on, we shall use letters a (resp. v), possibly with indices, to denote an address from **SInterval** (resp. an instance of ValMap). The (**RegionID**, **SInterval**) pair is also known as an *abstract location*, or *aloc* for short, in the literature. Alocs are used to represent variable-like entries, either on heap, in stack, or in registers.

Regions are defined as a map Region := **SInterval** → ValMap. Individual regions are denoted R, possibly with indices. For example, the stack region containing constant 7 at stack slot −4 and a pointer to a global variable at address 1000 at stack slot −12 would be represented as:[3]

$$\begin{aligned} R = \{&4[-4,0] \rightarrow \{CId \rightarrow 1[7,8]\}, \\ &4[-12,-8] \rightarrow \{GlbId \rightarrow 4[1000,1004]\}\} \end{aligned} \tag{1}$$

Regions with different identifiers are considered to be infinitely far apart. The C standard [26, page 83] considers the result of address arithmetic pointing outside a region undefined, so our treatment of regions is following the C standard (for binaries compiled from C programs). For binaries compiled from type-safe languages, our assumption is safe.

Finally, we define an abstract state as a map from region identifiers to regions: State := **RegionID** → Region. For denoting individual states, we will use the letter s, possibly with indices. The State map is indexed by a region identifier and address (strided interval), e.g., $s\,[GlbId, 4[1000,1004]]$. The indexing operation $s[r,a]$ returns the value map defining the location a in the region with identifier r in state s, or $\perp$ if the location is undefined. We define substitution on states $s\,[r, a \leftarrow v]$ as an operator that replaces the value map $s[r,a]$ with v, without changing other regions or addresses, and returns the newly constructed state.

For efficiency, we represent the maps in each level of the abstract state as persistent red-black trees [35] (using Eker's optimizations [19]) to allow fast functional updates with sharing. Regions use interval trees to efficiently detect overlap, and we use hash consing to avoid constructing duplicate objects.

Formally, our abstract interpretation is a monotone non-distributive framework (e.g., [34]) with domain $(\mathscr{P}(\mathsf{State}), \sqsubseteq, \sqcup, \perp)$, where the transition relation $post^{\#}$ is defined by the rules in Fig. 1, while the operations over value maps are defined in Fig. 2. To compute the fixed-point, we use a simple aggressive widening operator for strided intervals, described in [2]. We define the join and widen operators on states later (Section 3.4).

$$\begin{aligned} &post^{\#}(s_0, \textbf{if } c \textbf{ then } S_1 \textbf{ else } S_2) = post^{\#}(s_0, S_1) \sqcup post^{\#}(s_0, S_2) \\ &post^{\#}(s_0, \textbf{while } c \textbf{ do } S) = s_0 \triangledown post^{\#}(s_0, S) \\ &post^{\#}(s_0, \textbf{write}(r, a, v)) = s_0\,[r, a \leftarrow v] \end{aligned}$$

Figure 1: Definition of the Transition Relation. The pre-state is denoted s_0, statements S_i, the widening operator $\triangledown$, branch condition c, and temporary variable v of the ValMap type. Our instruction decoder creates temporary variables for intermediate results loaded from memory or created by complex assembly instructions. The control-flow construction (Section 2) identifies branches and loops, which can be classified as either if-then-else branches or while-do loops. Thus, the above are all the state-modifying transitions required.

$$\begin{aligned} &\oplus v = \{(r,a) \mid (r,a') \in v \wedge a = \oplus a'\} \\ &v_1 \otimes v_2 = \left\{ (r,a) \left| \begin{array}{ll} a = a_1 \otimes a_2 & \textit{if } (r,a_1) \in v_1 \wedge (r,a_2) \in v_2 \\ a = \top & \textit{otherwise} \end{array} \right. \right\} \\ &read(r,a) = s[r,a], \textit{ where } s \textit{ is the current state} \end{aligned}$$

Figure 2: Operations on Value Sets. The $\oplus$ symbol stands for a unary, and $\otimes$ for a binary operation.

3.2 Weak and Strong Updates

In this section, we discuss our treatment of weak and strong updates in more detail. A strong update overwrites the contents of the written region, and represents a definite change. In contrast, a weak update computes a join of the old and new contents. In general, if a region represents multiple concrete regions (i.e., summarizes them), strong updates are unsound.

Our analysis begins by creating a single strongly updatable region per allocation site. If that site is revisited during the course of analysis, a new weakly updatable region is created and reused every time the same allocation site is revisited later, as in the allocation site abstraction [11, 30]. Thus, the analysis creates at most two regions, one strongly and one weakly updatable, per allocation site. The described strategy is exactly the opposite of Balakrish-

[2] Note that unlike the prior work, we make the upper bound exclusive. We found this definition of strided intervals to be somewhat more convenient for dealing with misaligned reads and writes, which are frequent in binary code.

[3] Negative addresses are often used to index the stack frame of the currently executed function.

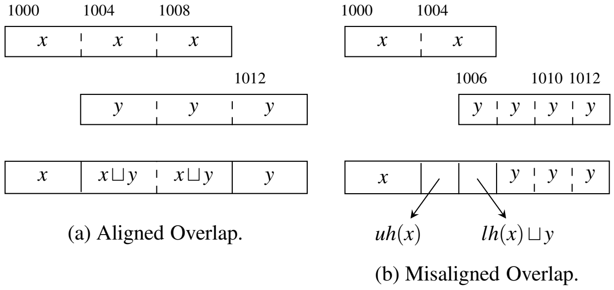

Figure 3: Chunking of Overlapping Addresses. The top rectangle in each figure represents the contents (denoted x) of a weakly updatable region at addresses shown above the rectangle. The addresses are $4[1000,1012]$ in (a) and $4[1000,1008]$ in (b). The dotted lines represent strides, while full lines represent chunk boundaries. The middle rectangle in each figure represents the newly written data (denoted y). The bottom rectangle represents the contents of the region after write. In (a), the region is split into three aligned chunks. In (b), the region is split into four chunks: chunk $4[1000,1004]$ that maintains the old value x, a smaller chunk $2[1004,1006]$ that contains the upper half (uh) of x, another chunk $2[1006,1008]$ containing a join of the lower half (lh) of x and of y, and finally $2[1008,1014]$ containing the newly written data y. Lower and upper halves can be computed using shift and mask operations on strided intervals. Other cases, like writing a single byte in the middle of a double word can be handled similarly.

nan and Reps's recency-abstraction [4], which maintains the latest region as strongly updatable while summarizing the older regions. Balakrishnan and Reps motivate their recency-abstraction as a necessity for resolving indirect jumps. In contrast, our three-stage technique exploits dynamic seed traces to resolve indirect jumps, so we opted for what we considered an easier-to-implement option, albeit possibly less precise in some cases. In the further exposition, all updates will be weak, as that case exposes more interesting details than strong updates.

3.3 Identification of Memory Locations

Unlike prior work [3] that relied on IDAPro [24] to detect alocs and then alternated the VSA analysis and aloc detection until the fixed-point is reached, our analysis discovers alocs and performs abstract interpretation at the same time. We detect alocs during the first write that is within the allowed bounds of a region. Writes that are outside the allowed bounds produce a warning. The allowed bounds are computed as a side-result of the analysis.

The most difficult part of the alocs identification is dealing with overlapping writes, which tend to be relatively frequent in practice. A simple solution, to compute the union of overlapping addresses and set their content to $\top$, tends to be insufficiently precise. A more precise alternative, used in our analysis, is to partition the overlapping addresses into non-overlapping chunks along the stride boundaries, and then compute the value stored at each chunk separately. We found the increase in precision especially important for analyzing programs that operate on strings, which are often loaded from memory or created on stack in four-byte chunks and then accessed byte-by-byte. The algorithm for partitioning sets of addresses into strides is conceptually simple, but tedious to implement because there are many special cases appearing in practice. Due to lack of space, we skip the details, but we illustrate the idea in Figure 3.

3.4 Join and Widening of Regions

We use the same chunking approach to compute a join of regions. For example, given two regions R_1 and R_2, their join first partitions addresses into three disjoint sets: $S_1 = dom(R_1)/dom(R_2)$, $S_2 = dom(R_2)/dom(R_1)$, and $S_3 = dom(R_1) \cap dom(R_2)$, where dom is the domain of a map and '/' is set difference. In example (1), $dom(R) = \{4[-4,0], 4[-12,-8]\}$. Addresses within each set are then partitioned into disjoint strided intervals so as to keep the original strides when possible. Finally, the join operator computes the value stored at each address in $S_1 \cup S_2 \cup S_3$. The values stored at addresses from S_1 are the (possibly truncated) values in the region before the write. The values stored at S_2 are the newly written values. The values stored at S_3 are the addresses overwritten with new values. Assuming weak updates, the result is a join of the old and new values. The join over states is defined as:

$$s_1 \sqcup s_2 := \left\{ (r, R_1 \sqcup R_2) \,\middle|\, \begin{array}{l} \text{either } (r,R_1) \in s_1 \wedge (r,R_2) \in s_2 \text{ or} \\ (r,R_1) \in (s_1 \cup s_2 / s_1 \cap s_2) \wedge R_2 = \bot \end{array} \right\} \quad (2)$$

Algorithm 1 computes $R = R_1 \nabla R_2$. Widening terminates when the contents reach a fixed-point. It might seem that the last case in Algorithm 1 could introduce substantial imprecision. However, since global addresses are most often constants and since writes to the stack are most often constant offsets from the `ESP` or `EBP` registers, and neither of those registers is commonly modified within loops, the imprecision of widening rarely impacts the most important regions, globals and the stack. Widening on states is done similarly as in (2), only $\sqcup$ is replaced with ∇.

Algorithm 1 Region Widening. Lines 2 and 3 perform chunking, illustrated in Figure 3. Lines 4 and 5 do point-wise widening on addresses (and their contents) defined in both regions. Lines 6 and 7 union the addresses (and their contents) belonging only to R_1 with the temporary result. The rest of the algorithm handles the case when the region used for widening (R_2) defines some addresses not defined in R_1.

1: $R \leftarrow \emptyset$
2: $S_1 \leftarrow dom(R_1)/dom(R_2), S_2 \leftarrow dom(R_2)/dom(R_1)$
3: $S_3 \leftarrow dom(R_1) \cap dom(R_2)$
4: **for** $(a,v) \in S_3$ **do**
5: $\quad R \leftarrow R \cup (a, R_1[a] \nabla R_2[a])$
6: **for** $(a,v) \in S_1$ **do**
7: $\quad R \leftarrow R \cup (a,v)$
8: **if** $\exists (a,v) \in S_2$ **then**
9: $\quad$ Issue a warning about a possible write out of bounds
10: $\quad$ Widen contents of all existing chunks in R with v

4. GUIDED DYNAMIC ANALYSIS

The third stage of our approach applies dynamic analysis (based on symbolic execution) to compute concrete test inputs that trigger vulnerabilities in the program under analysis. To direct the dynamic analysis, we introduce a two-component heuristic. The first component of the heuristic is the distance in the VPA; below we give an algorithm to efficiently compute these shortest path lengths. The second component of the heuristic is based on the number of statements in the data-flow slice, which is a transitive closure of read-write dependencies of all variables used by the statement triggering the vulnerability. We compute this component using standard techniques of interprocedural data-flow analysis [38]. We note that both components can also be seen as generalized pushdown predecessor problems (GPPP) [29] on a weighted pushdown system (WPDS).

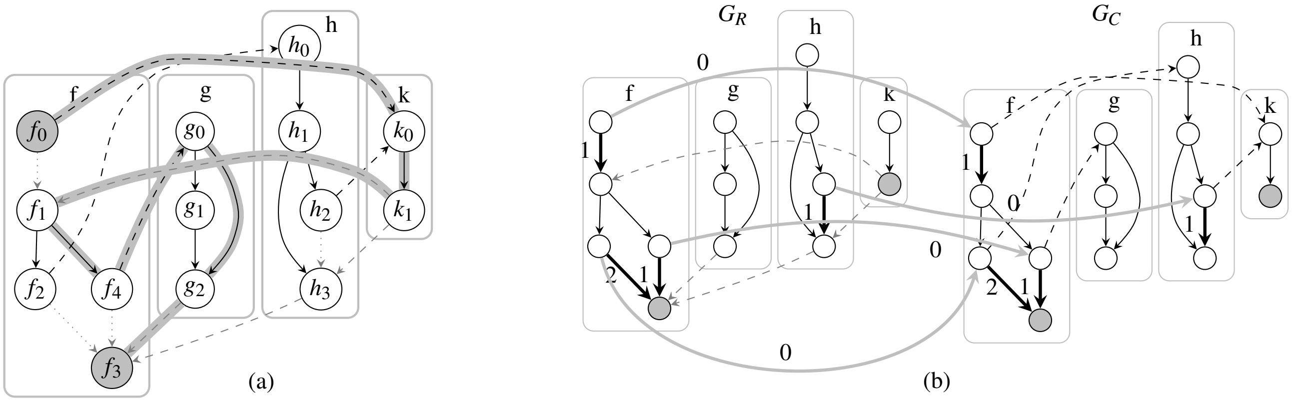

Figure 4: Computation of Shortest Paths on a VPA. The left figure illustrates the shortest well-matched path from f_0 to f_3, highlighted in gray. The right figure shows the combined graph G_{RC} used for computing shortest distances over the language of calls and returns $(\mathscr{R}|\mathscr{W})^*(\mathscr{C}|\mathscr{W})^*$. Each function is enclosed in a grey rectangle with the name of the function at the top. Internal edges (with implicit weight 1) are denoted $\longrightarrow$. Call (resp. return) edges are denoted $\dashrightarrow$ (resp. $\dashrightarrow$). The $\dashrightarrow$ edge connects matching call and return nodes, $\rightarrow$ connects G_R and G_C call sites, and $\Rightarrow$ represents the summary edges whose weight is the shortest path through the callee.

For distances, the semiring is $(\mathbb{N}, min, +, \infty, 0)$, where *min* is a binary operator returning the minimal value. For counting the number of reachable statements from the slice, we can use the trivial semiring $(\mathbb{S}, \cup, \cup, \emptyset, \emptyset)$, where $\mathbb{S}$ is the set of instructions in all basic blocks of a VPA. The two-component heuristic is a function with an $\mathbb{N} \times \mathbb{N}$ codomain. For a given location of a potential vulnerability, we precompute the values of the function for each VPA state; it is convenient and sufficiently fast to compute all of these values at once, before starting the exploration.

We also use a second heuristic to attempt to cover a variety of patterns of paths through relevant loops — our system identifies paths using the VPA, then records which paths have been explored and then explores various patterns of these paths. For a given location of a potential vulnerability and a given SCC, we identify paths within SCCs before and construct patterns during the search.

In the rest of this section, we first describe as background the baseline approach of symbolic execution, with an undirected search strategy (Section 4.1). Next, we present an efficient algorithm for computing shortest paths in a weighted VPA (Section 4.2), which we use to propagate cost information as part of the program-wide two-component heuristic, and describe how we use this heuristic at branches (Section 4.3). Finally, we discuss our approach for covering loop path patterns (Section 4.4).

4.1 Baseline: Symbolic Execution

We describe our exploration system as dynamic because it treats most of the program state as concrete: only values derived from a specified input are treated as symbolic. The system symbolically executes one whole-program execution path at a time; when a branch condition is symbolic, the tool checks whether either the condition or its negation is satisfiable using the Z3 [16] decision procedure. Operationally, the system works similarly to previous tools such as Klee [9], EXE [10], and MineSweeper [7].

When both sides of a branch are feasible, our system's default behavior is to choose a direction randomly. Though fair in a sense, this is clearly a very uninformed search strategy. Thus the focus of the rest of this section is on techniques that harness more global information, such as from static analysis, to make better branch choices. However, the symbolic execution engine only takes these suggestions as advice: it will never explore an infeasible path or one that it has previously explored. We refer to the symbolic execution of one such whole-program execution path as an *iteration* of the dynamic exploration.

4.2 VPA Shortest Paths

The algorithm we present in this section computes the first component of our heuristic. Intuitively, we would like to direct the dynamic analysis to follow a (whole-program) path that reaches a target, and among such paths we would like one with a low cost. More precisely, we would like to compute, for each basic block in the program, the cost of the least-costly path from that point to the target basic block. It is a convenient analogy to think of the costs as distances, so that our goal is to find the length of the shortest paths from each basic block to a single target.

If our program representation were a simple graph, Dijkstra's shortest path algorithm [17] would be appropriate, but it does not follow the restriction that calls and returns must be well-matched. For example, suppose we want to find the shortest path from f_0 to f_3 in Figure 4.a. The classic shortest path algorithm would find the path $f_0 k_0 k_1 h_3 f_3$, which is incorrect, because the return from `k` to `h` is executed with stack configuration f_1 that doesn't match the state `k` is returning to. Instead, the correct shortest path with matched returns is $f_0 k_0 k_1 f_1 f_4 g_0 g_2 f_3$ (assuming equal edge weights). Such a situation occurs in practice, for instance, if a common function such as `printf` is called from both `main` and the function containing a vulnerability. If we ignored call-return matching, our tool might think that the shortest path to the vulnerability was one that started in `main`, called `printf`, and then returned from `printf` to the vulnerable function. But attempting to follow this infeasible path would be unproductive. However, while we match calls and returns within the path, the definition of the path length is otherwise context-insensitive, in the sense that the distance between a point A and a point B does not depend on the call stack at A.

Thus our goal is to build an analogue of Dijkstra's shortest path algorithm that operates on a VPA, and finds the lengths of the shortest path among those paths in which there are no mismatches between calls and returns. It is not immediately obvious whether such an algorithm could be efficient: a given statement might be reached with many different call stacks, and the shortest path between two points in a VPA can have exponentially many edges.

The key insight in obtaining an efficient algorithm for this problem is to start with the special case of well-matched paths (with no unmatched calls or returns), since it is easier to combine paths under the invariant that they are well-matched. Then we can extend well-matched paths to general VPA paths by constructing paths out of unmatched returns, well-matched segments, and unmatched calls, in which no unmatched call precedes an unmatched return (thus ensuring that they are unmatched).

4.2.1 Well-Matched Paths

For the first step of the algorithm, focusing on well-matched paths, our specific goal is to compute, for each function, the length of the shortest well-matched path that starts at the entrance to the function and ends at the return from the function. This per-function shortest path length is a kind of function summary that can be efficiently computed bottom-up. For leaf functions, we apply Dijkstra's algorithm to compute the length of the shortest path from entrance to exit. If for all of the functions $g_1, \ldots, g_k$ that might be called by a function f, we have already computed the summary length l_i for g_i, we can replace the call to g_i with an edge of cost l_i and then again compute the summary length for f with one intra-procedural invocation of Dijkstra's algorithm.

The above described bottom-up computation might not compute the shortest entrance-to-exit path for every function in SCCs in the call graph, so another level of iteration is required. Let S be the set of functions in an SCC of the call graph. We start by setting the summary length for each function in the S to ∞. Then, we repeat $|S|$ times a process of updating the summary length for each function in S ($|S|^2$ updates total). To update the summary length of a function f, we recompute its entry-to-exit path length based on the best estimates found so far for the other functions in the SCC (and the previously-computed correct values for any called functions outside the SCC), updating the summary length if the computed value is smaller. This process is shown in Algorithm 2.

Algorithm 2 Per-Function Computation of Shortest Paths. The function DFN returns the list of functions sorted by depth-first post-order. The function $DSP(f, costs)$ computes the length of the shortest path from the entry to exit of the function f, using Dijkstra's shortest path algorithm, under the assumption that the shortest paths through called functions are those given in *costs*. The function $SCC(G, f)$ returns all the functions in G that are in the same SCC as the function f.

let $CG \leftarrow$ call-graph of VPA
$costs \leftarrow \emptyset$
for $f \in DFN(CG)$ **do**
  if $f \notin costs$ **then**
    let $S \leftarrow SCC(CG, f)$
    for $f' \in S$ **do**
      $costs[f'] \leftarrow \infty$
    for $i = 1 \ldots |S|$ **do**
      for $f' \in S$ **do**
        $costs[f'] \leftarrow \min(costs[f'], DSP(f', costs))$

The complexity of the part of the algorithm for a single SCC is $\mathcal{O}\left(M^2 E \lg N\right)$, where M is the number of functions in the SCC, and N (resp. E) is the number of basic blocks (resp. edges) in the largest function in the SCC. The factor $\mathcal{O}(E \lg N)$ is the running time of Dijkstra's algorithm using a binary heap. The algorithm is guaranteed to converge after analyzing each function in the SCC M times, because the shortest path always goes through the base case, rather than recursing to a function already on the call stack. After the i-th iteration through all the functions, the algorithm will have correctly computed the shortest length for call paths whose depth within the SCC is at most i. Since the shortest path for each function has no recursive calls, its maximum depth within the SCC is M, so M iterations suffice.

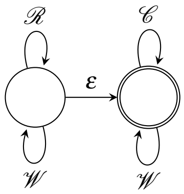

Figure 5: NFA To Constrain a VPA Path. A general VPA path, with no mis-matched calls and returns, is a concatenation of unmatched returns $\mathscr{R}$, unmatched calls $\mathscr{C}$, and well-matched words $\mathscr{W}$, with the restriction that the unmatched returns precede the unmatched calls.

To see why this the shortest path cannot contain a recursive call, suppose, to the contrary, that the shortest path were one in which a function f contained a recursive call to the same function f. For instance, suppose that g calls f (the "outer" invocation of f), then f calls itself (the "inner" invocation of f), and then both invocations return. Then we can construct a strictly shorter path by replacing the outer invocation of f with the inner invocation of f. In the example, this gives the path in which g calls f, then f returns. The new path will be well-matched, but it is shorter, contradicting the assumption that the first path was the shortest.

4.2.2 General VPA Paths

For the second step of the algorithm, we extend the well-matched paths computed in the first step to general VPA paths (without mismatched calls and returns) by concatenating well-matched segments, unmatched returns, and unmatched calls. The key insight is that this is possible as long as the unmatched returns come before the unmatched calls in the path, which ensures that they do not mismatch with each other. More formally, let $\mathscr{C}$ (resp. $\mathscr{R}$) be the set of calls (resp. returns), and $\mathscr{W}$ the set of well-matched words. Then any word without mismatched calls and returns can be written in the form $(\mathscr{R}|\mathscr{W})^*(\mathscr{C}|\mathscr{W})^*$, where $|$ and * are the standard regular expression operators of alternation (union) and Kleene star (repetition zero or more times).

Using the lengths for well-matched entrance-to-exit paths computed in the first step, the $\mathscr{W}$ symbols in the regular expression can be represented by summary edges from a call site to a return. What remains is to enforce the constraint that unmatched returns ($\mathscr{R}$) precede unmatched calls ($\mathscr{C}$). This can be seen as requiring paths in the VPA that simultaneously match the regular expression. We can express the regular expression with the NFA shown in Figure 5, and express the intersection of $(\mathscr{R}|\mathscr{W})^*(\mathscr{C}|\mathscr{W})^*$ and the VPA by a product of the two automata. This product is equivalent to constructing an automaton with two copies of the VPA, and we take that perspective in describing the construction in more detail.

To find the lengths of paths that satisfy the no-mismatched-calls constraint expressed by the regular expression $(\mathscr{R}|\mathscr{W})^*(\mathscr{C}|\mathscr{W})^*$, we use Dijkstra's algorithm on a graph consisting of two copies of the VPA. For the first half $(\mathscr{R}|\mathscr{W})^*$ we construct a graph G_R with call edges erased, return edges retained, and a summary edge between each pair of matching calling and return nodes having the weight of the shortest path through the callee. Paths through this graph correspond to executions with unmatched returns and well-matched function invocations in any order. Dually we construct for the second half $(\mathscr{C}|\mathscr{W})^*$ a graph G_C with call edges retained, return edges erased, and summary edges as in G_R. Paths through G_C correspond to executions with unmatched calls and well-matched function invocations in any order. Finally, we construct a combined graph G_{RC} by linking G_R and G_C with zero-weight edges from each call node in G_R to the corresponding call node in G_C. (It suffices to link only call nodes because every word in $(\mathscr{C}|\mathscr{W})^*$ starts with a call.) Then, we compute the shortest VPA path from one node to another by running Dijkstra's algorithm taking a node from G_R as

Algorithm 3 VPA Shortest Path Computation. To compute the length of the shortest VPA paths to a single target warning location t_w, we construct a graph G_{RC}, representing the combination of graphs G_R and G_C that allow unmatched returns and unmatched calls respectively. The $\xrightarrow{i}$ symbol denotes an edge with weight i. Lines 1–4 add two copies of each VPA node to G_{RC} (representing the nodes in G_R and in G_C); the two copies are labeled with different labels (R and C). Lines 7–9 add call edges (⇢ in Fig. 4) to G_C, edges between corresponding call sites of G_R and G_C (⇒), and summary edges between callers and return nodes (⋯→) to both G_R and G_C. Lines 10–11 add return edges (⇢) to G_R. Lines 12–13 add the intraprocedural edges (⟶). Finally, lines 14–16 run Dijkstra's algorithm to compute all-source-single-target distances to the R copy of t_w, and return the distances using the C copies as sources.

1: $G_{RC} \leftarrow$ empty graph
2: **for** $b \in$ nodes of *VPA* **do**
3: let $b' \leftarrow \langle b,R\rangle$ and $b'' \leftarrow \langle b,C\rangle$
4: add nodes b' and b'' to G_{RC}
5: **for** $e \in$ edges of *VPA* **do**
6: let $s \leftarrow source(e)$ and $t \leftarrow target(e)$
7: **if** e is call edge **then**
8: let $w \leftarrow$ the length of the shortest path to return from t and $r \leftarrow$ the node where t returns
9: add edges $\langle s,C\rangle \xrightarrow{0} \langle t,C\rangle$, $\langle s,R\rangle \xrightarrow{0} \langle s,C\rangle$, $\langle s,R\rangle \xrightarrow{w} \langle r,R\rangle$, and $\langle s,C\rangle \xrightarrow{w} \langle r,C\rangle$ to G_{RC}
10: **else if** e is a return edge **then**
11: add edge $\langle s,R\rangle \xrightarrow{0} \langle t,R\rangle$ to G_{RC}
12: **else if** e is an internal edge of weight w **then**
13: add edges $\langle s,R\rangle \xrightarrow{w} \langle t,R\rangle$ and $\langle s,C\rangle \xrightarrow{w} \langle t,C\rangle$ to G_{RC}
14: $D \leftarrow \text{Dijkstra}(G_{RC}, \langle t_w,R\rangle)$
15: **for** $s \in$ nodes of *VPA* **do**
16: $s.d \leftarrow D[\langle s,C\rangle]$

the source and a node from G_C as the target. This construction is shown in Algorithm 3, and demonstrated graphically in Figure 4.b.

4.3 The Two-Component Heuristic

Obviously, an execution path can only demonstrate a possible vulnerability at an instruction if it executes that instruction, and often the vulnerable instruction depends on values computed earlier in a program. We take these two intuitions as the basis for our guidance approach for path selection.

More formally, given a warning produced by static analysis, our approach computes for each VPA node a pair of non-negative integers (d,s). The first component d represents the cost of the least-expensive path from the basic block to the warning. The second component s counts the number of instructions in the data-flow slice of the warning that can be reached from the basic block.

For computing d, we use the shortest-path algorithm of Section 4.2, assigning a weight of ∞ to loop back edges and 1 to all other edges. Assigning a high cost to loop back edges causes the search to prefer to reach a target at the first opportunity, even if it would also be reachable on a future loop iteration.

To compute s for a given potential vulnerability, our system first computes the complete backward slice from the vulnerability. Then it makes a single bottom-up pass of standard interprocedural propagation to compute, for each basic block b, the set of instructions in the slice that are reachable from b. The value of s at a location is the cardinality of that set.

The two components d and s represent goals that are sometimes in tension. We would prefer to take paths that reach the potential vulnerability quickly (small d), and we would like to cover many data-flow predecessors of the potential vulnerability (large s). When these desires conflict, our heuristic attempts to balance them, choosing the one that is more salient in a particular instance, while incorporating randomization so that a future path may make a different choice, especially if it was close.

```
char buf[20], *p = buf;
int mode = 0;
while (p >= buf)
    switch (read_char()) {
        case 'a':
            if (mode == 1) {
                *p++ = 'x';
                mode = 0;         // path 1
            } else {              /* path 2 */ }
            break;
        case 'b':
            mode = !mode; break;  // path 3
        default:
            p = max(buf, p--);    // path 4
    }
```

Figure 6: An example loop for which a repeating pattern of paths (here, alternating between paths 1 and 3) is required to cause an overflow.

Specifically, suppose that at a branch, we have a choice between a target with heuristic values (d_1,s_1) and one with (d_2,s_2). Since we prefer a target with large d but small s, we compute the cross difference $x = d_1 \cdot (1+\ln(1+s_2)) - d_2 \cdot (1+\ln(1+s_1))$. A negative value of x corresponds to a preference for the first target, while a positive value corresponds to a preference for the second. To include randomization, we compute the logistic value $r = \frac{1}{1+e^{-kx}}$, and branch according to whether a number selected uniformly at random in $[0,1]$ is less than r. The parameter k controls how large a difference in costs corresponds to a given probability difference; our system currently has $k = \ln(1.001)$.

4.4 The Loop Pattern Heuristic

The two-component heuristic described above is effective for directing execution towards a desired point in a program; when a program has loops, it will usually explore only a few iterations. But some program behaviors occur only when loops, and in particular certain paths those loops, execute repeatedly. This is especially true for buffer-overflow and integer-overflow vulnerabilities.

A simple example of a loop for which a pattern of loop paths is needed to cause an overflow is shown in Figure 6 (similar but more complicated loops appear in several of the benchmarks in Section 5). There are four paths through the body of the loop, numbered in comments. Choosing one of the loop body paths randomly on each iteration is very unlikely to produce an overflow (`p - buf` $\geq$ `20`): only path 1 increments the pointer, and path 4 decrements it, so they tend to cancel each other out. On the other hand because of the `mode` flag, it is infeasible to execute path 1 on every iteration. A pattern that leads to the overflow is one that alternates between paths 1 and 3. At a high level, our approach is not to statically determine which such pattern of paths leads to an overflow (though this could be an interesting direction for future work). Instead, our approach is to guide the exploration to cover a wide variety of path patterns, so that if a vulnerability can be triggered by a short pattern, the exploration will find it relatively early.

Thus to improve our coverage of these kinds of behavior, our system directs execution to sometimes repeatedly execute one or more specific paths within a loop as many times as possible. For a loop that either contains the vulnerability, or contains an instruction from the slice of the vulnerability, this heuristic will choose a short pattern of paths from among those loop paths previously observed. Then on each iteration of the loop it will try to execute the next path from the pattern. In order to construct such patterns, we need to be able to uniquely identify paths through a loop body (precisely, the (necessarily acyclic) paths through a control-flow graph SCC that do not pass through any sub-SCCs).

We begin by identifying all exits from the analyzed SCC. The exits include entries into a sub-SCC. We assign the weight of one to the exit edges and start traversing the acyclic graph (loop body) backwards. For every node, we sum up the weights of all outgoing edges and assign the sum to the node and all the incoming edges. After the process terminates (it terminates because the graphs are finite and acyclic), every node is labeled with the number of distinct paths to exits. The header of the loop is labeled with the number of paths, call it P, from the header to the SCC exits.

Each acyclic path through the SCC can be assigned a unique number in the range $0 \dots P-1$. We call a word $w = \{0 \dots P-1\}^*$ a *loop path pattern*. As the exploration runs, the system records the unique path numbers of feasible paths. Once the system has observed some feasible paths (currently, after the fifth execution of the program), it will begin to try to traverse a path pattern on some executions. Specifically, the system chooses whether to use a pattern, and if so the length of pattern to use, according to a geometric distribution as follows. For 50% of program executions, it does not use a pattern (rather, chooses a path number randomly). For 25% of executions, it picks a pattern of length 1 (a single path number), and uses this path number for every loop iteration. For 12.5% of executions, it picks a pattern of length 2 (a pair of path numbers P_a and P_b), and uses path P_a for odd-numbered iterations and P_b for even-numbered iterations. In the same way, longer patterns are chosen less frequently. In each case, the pattern is constructed by uniform random selection from the set of feasible paths discovered so far. The constructed loop path pattern is repeated every time the loop is revisited during the search, until the program exits. On each iteration, the heuristic will attempt to follow the path given by the path number, subject to feasibility.

Observe that, unless one of the paths in a path pattern leads to a loop exit, one effect of a pattern is to attempt to execute as many loop iterations as possible (in contrast, random choice would stop with probability 50% after each iteration). Finally, while we apply the path-pattern approach as described above on every execution when the vulnerable instruction is inside a loop (in the same function), we apply it more selectively for loops which contain only an instruction from the vulnerable statement slice. For these loops, which are less frequently relevant to a vulnerability, we apply the path-pattern approach for a fraction (one-third) of executions.

5. EXPERIMENTAL RESULTS

We evaluate our tool's detection and test generation for buffer overflow vulnerabilities using a suite of examples developed by Zitser et al. [50], extracted from historic vulnerabilities in 3 widely used network servers. Though the examples have been reduced to omit most of the irrelevant parts of the programs (they average 665 lines of C code each), they cover a wide variety of kinds of overflow, and demonstrate sufficient complexity to make analysis challenging. Each benchmark represents a conceptually single bug, which can manifest in out of bounds accesses at multiple locations. The original versions of the benchmarks were designed purely for static analysis; we use versions modified by Saxena et al. [41] to read inputs from files for use with dynamic techniques.

The experiments were performed on a 4-core Xeon E5540 workstation (our tools are single-threaded) with 12GB of RAM, running Debian GNU/Linux with a 64-bit kernel version 2.6.26. The results of the evaluation are summarized in Table 1 and described in more detail in the remainder of the section.

5.1 Static Analysis

Our static analysis finds out-of-bounds memory writes, analyzing both the application and the libraries together, rather than depending on hand-written API summaries. We use the `dietlibc` library as a compatible but simplified replacement for the system's C library. To improve the quality of the results, we also treat some C library functions whose only side-effect is output, such at `write` and `syslog`, as no-ops, since these functions do not modify their inputs or global variables visible to the program.

All benchmarks have labeled vulnerabilities, which enabled us to accurately count false negatives and positives. As shown in Table 1, our approach detects all vulnerabilities in the benchmarks (i.e., there are no false negatives), but reports 72% false positives. The third stage of our approach, the guided dynamic analysis, can prioritize warnings according to whether it finds a concrete input triggering the vulnerability within a given timeframe, but cannot prove a warning to be false positive. The imprecise widening (c.f., Section 3.1) in our implementation caused the majority of false positives. Another source of false positives are weak updates of memory regions. Time and space usage of the static analysis are relatively modest. The figures shown in Table 1 are for analyzing a VPA read from disk; VPA construction takes an additional second or two per trace.

5.2 Dynamic Analysis

We next applied the dynamic analysis to confirm one of the true positive static analysis warnings for each of the benchmarks. When there are multiple true positives from static analysis, we report the results for one selected uniformly at random. We obtained similar results when repeating the dynamic analysis with other true positives, as expected given that the warnings relate to a single underlying bug. The symbolic input in each benchmark is a bounded-size character buffer. The dynamic tool produced test cases proving the existence of bugs in all but one of the programs. The guidance from static analysis made the tool more efficient and helped it find bugs it otherwise could not.

The dynamic analysis results, shown in the right-hand side of Table 1, demonstrate that static-based guidance is a significant advantage. The benchmarks vary significantly in difficulty for symbolic execution. On the easier benchmarks, guidance often improved both the running time and the number of whole-program paths explored (iterations). Guidance sometimes increases the average time per iteration. Often, our loop pattern heuristic picks paths with many loop iterations necessary for triggering vulnerabilities, but in the S2 benchmark such paths are also more expensive. Thus, although the total number of iterations is smaller, the overall time cost is higher. The unguided search explored many paths that were both unproductive and short. The cost of using heuristics during execution is negligible; the reported times include pre-computation of quantities such as distance which required only a fraction of a second. On the more difficult benchmarks, guidance can make the difference between success and failure, as seen in S1, S5, and S6. S6 is a buffer overflow caused by an integer overflow: by directing exploration to consider a long sequence of digits, our guidance leads in under 12 seconds to an overflow that undirected execution failed to find even in 6 hours. S1 and S5 contain loops similar to the example in Figure 6, in which a particular pattern of loop paths, corresponding to a repeating pattern in the input, is needed to cause a pointer to overflow a buffer. (S3 is similar to S1 and S5, but the heuristic is not effective because the loop spans several functions.)

Bmarks.		Instrs.		Static analysis					Undirected dynamic analysis		Directed dynamic analysis	
		App.	*libc*	*Warns.*	*Bugs*	*False pos.*	*Time (s)*	*Mem. (Kb)*	*Iterations*	*Time (s)*	*Iterations*	*Time (s)*
BIND	B1	1705	2120	15	1	14	3.3	46000	54	2.8	20	3.6
	B2	1290	2178	22	1	21	3.2	50416	137	13.3	72	25.1
	B3	719	3058	14	1	13	14.2	80768	9	1.6	4	2.6
	B4	394	3621	40	2	38	29.2	320480	1	1.9	1	2.0
Sendmail	S1	929	2021	33	28	5	24.9	95472	†	†	3347	2990.6
	S2	524	2750	28	2	26	20.2	79824	16	2.9	8	66.1
	S3	318	1653	14	3	11	1.6	33216	†	†	†	†
	S4	370	2447	20	7	13	10.6	57808	3	19.0	1	9.1
	S5	392	1282	10	3	7	1.2	18880	†	†	332	202.6
	S6	595	2247	6	1	5	3.2	40112	†	†	86	11.3
	S7	957	2595	42	2	40	15.4	142208	56	6.9	46	8.8
WU Ftpd	F1	571	1561	11	4	7	1.1	30448	309	8.1	11	1.1
	F2	807	2549	13	1	12	5.8	53632	1455	65.8	11	1.4
	F3	684	1639	37	27	10	7.5	78624	143	60.0	18	11.6
Total				305	83	222	141.4					

Table 1: Summary of the Experimental Results. The first section of the table lists the benchmarks, the second section shows the size of the benchmarks in machine instructions, and the third section shows the results of the static analysis. The fourth and fifth sections show the results of dynamic test generation, where the directed analysis used the static analysis results for a randomly selected true positive, while the undirected analysis had no such information. The dynamic analysis results are averages over five runs with different random seeds. "Iterations" counts the number of whole-program executions generated until finding a bug-revealing one, while the † symbol indicates the analysis could not trigger the bug within six hours.

The instruction coverage of the dynamic analysis was usually very similar between the undirected and directed runs, differing by just 1-2%. The only large difference was for S7, when the directed run covered about 40% of the unique instructions versus 60% for the undirected run. This confirms, as also visible in the iteration count, that the directed dynamic analysis finds the vulnerability more quickly because it avoids exploring irrelevant parts of the state space.

6. RELATED WORK

Static Analysis of Binaries. Our static analysis is similar to Balakrishnan and Reps's work [3], with a few differences discussed in Section 3. Kinder et al. [27] explain the "chicken-and-egg" nature of the problem of inferring the control-flow of binaries statically: data-flow analysis is required to infer the control-flow information, and control-flow analysis is required to infer the data-flow information. They combine data- and control-flow analysis and compute a safe approximation of the control-flow. In practice, it is easy to construct examples that defeat the static approach, because even distinguishing CISC assembly instructions from data is a difficult task [42]. We chose a different tradeoff — to use dynamic analysis with static augmentation only for branches we are certain we can resolve precisely. More exploration is needed to reach a definite conclusion on the comparative merits of the two approaches.

Guiding the Search. Improvements in the efficiency of search over the state space have been an active research area in verification and testing of protocols and software. In the context of protocol model checking, Yang and Dill [48] used Hamming distance of states as a greedy best-first-search metric in their Murϕ model checker. They compute several pre-images of the negated properties — the process they call target enlargement — and then compare every visited state to the enlarged target. We could apply similar enlargement to targets identified by static analysis, but it remains unclear how enlargement would help with alternating paths within SCCs. Edelkmap et al. [18] studied several heuristics. They used an approximate distance function to a state where a given LTL formula holds and found that A^* search worked best on their protocol benchmarks. We experimented with approaches similar to A^* in our domain, but it appeared difficult to determine an appropriate state-ranking function automatically. Godefroid and Khurshid [20] propose using genetic algorithms for finding errors in large state spaces, focusing specifically on heuristics for deadlock detection and property violations related to enabledness of transitions and message exchanges. Our heuristics are more tailored towards finding buffer overflows, especially in cases with multiple nested loops and multiple paths through the loop body.

Lal et al. [29] studied construction of minimal length explanations of crashes (produced by concrete traces) having only partial information about the trace. Their goal is to find a minimal length path passing through a maximal number of observed check-points in the code. In our setting, the data-flow slice produced by static analysis can be seen as partial information about the trace (we don't take control-flow dependencies into account), but we use that information only heuristically and do not attempt to maximize the number of statements from the slice on the path. Their algorithm is exponential in the number of check-points. Since we have only one check-point (i.e., the target), the exponent disappears and our algorithm can be seen as a special case of theirs.

Groce and Visser describe heuristics for finding property violations with the Java PathFinder model checker [22], based on branch coverage and thread inter-dependencies. Burnim and Sen [8] focus on achieving high line coverage and present several control-flow guided search heuristics, including one that is based on CFG distances but does not include matching of calls and returns. Achieving high line coverage was not sufficient to trigger vulnerabilities in our benchmark suite. Further, without guidance provided by the static analysis and special heuristics for strongly connected components, we were unable to hit vulnerabilities in a number of more difficult benchmarks. Zamfir and Candea [49] apply symbolic execution to synthesize an execution that reproduces a bug report.

Among several heuristics, they propose a "proximity-guided path search" that uses control-flow distance. In comparison to the algorithm we present in Section 4.2, the give only a simplified presentation without a complexity analysis, and rather than analyzing recursive calls they simply give them a weight of 1000. Saxena et al.'s loop-extended symbolic execution [41] introduces constraints that summarize loops by expressing other variables in terms of the number of times a loop executes, allowing symbolic reasoning across paths that execute different numbers of iterations. By contrast the loop exploration heuristic in this work applies to a single path at a time, and focuses on which path through the body of a loop triggers a vulnerability. Rybalchenko and Singh [40] propose a subsumer-first heuristic to steer symbolic reachability analysis of benchmarks from the transportation domain. Their heuristic is very intuitive: it prefers larger (according to subsumption ordering) states, greedily trying to get to a state large enough to contain the error state. Their technique could be classified as a look-back, while ours exploits static analysis and is inherently a look-ahead technique. We experimented with several state orderings, and were unable to get them to work. Subsumption ordering would be prohibitively expensive in our setting, as path conditions, describing states, can be large.

Hybrid Static-Dynamic Analysis. The Synergy algorithm [23] combines model-checking and DART [21] to try to cover all abstract states of a program. Our work has no ambition to produce proofs, and we expect that our approach could be used to improve the performance of the DART part of Synergy and other algorithms that use test generation as a component. DSD-Crasher [15] more closely resembles our work in performing dynamic, static, and dynamic analysis in sequence, though the stages are quite different. The closest point of comparison is that DSD-Crasher generates constraints via static analysis and solves them to create dynamic test cases, whereas our second dynamic analysis incorporates both constraint generation and solving.

7. LIMITATIONS AND FUTURE WORK

The first, and the most obvious, limitation is that the completeness of the VPA constructed in the first stage depends on the capability of the seed tests to exercise indirect jumps (or calls). One possible solution would be to use completely static analysis to construct the VPA (e.g., [27]). Unfortunately, sometimes in practice assembly instructions cannot even be decoded statically, as varying sizes of CISC assembly instructions make the instructions and data difficult to distinguish. Further, static analysis of binaries is inherently imprecise. Another possibility is to use DART or our three-stage approach in a loop for discovering concrete inputs and using them instead of the seed tests.

The second limitation of our approach is the precision of our static analysis. Currently, our implementation context-sensitively follows all calls in a brute-force manner and does not use function summaries or k-sensitivity [43]. Such an approach is unlikely to scale to very large applications, and we are planning to explore less precise and more efficient approaches (summaries, k-sensitivity, context-insensitive analysis). A less precise analysis would probably produce more warnings, decreasing the precision of the guiding information provided to the third stage of our approach. The number of warnings produced by a context-insensitive analysis could be reduced by extending the analysis with aggregate structure identification [36], affine-relation analysis [33], and recency-abstraction [4], as in [3], but it is obvious that the tradeoff between the precision of the static analysis and computed guidance information requires significantly more research.

The third, more subtle, limitation of our approach is a possible overfitting of the guidance heuristic to our set of benchmarks and the vulnerabilities (buffer overflows) we are looking for. Design of heuristics is inherently prone to overfitting and designing robust heuristics is a tedious time-consuming task. For example, it took the SAT solving community almost 30 years ([13]–[32]) to come up with effective robust decision heuristics, and they are still being improved. Our paper is a step forward in developing robust dynamic test generation guidance heuristics, but obviously, much more work remains to be done.

8. CONCLUSIONS

We have presented an approach that applies dynamic analysis, static analysis, and dynamic analysis again to explore the large path space of programs given a binary. The analysis revolves around a visibly pushdown automaton (VPA) which represents the programs global control flow structure. Starting with dynamic analysis helps resolving indirect jumps, and the static analysis helps symbolic execution direct exploration towards vulnerabilities based on the shortest paths and loop pattern heuristics. In preliminary experiments, our static analysis finds all of the vulnerabilities in the suite, and dynamic analysis constructs test inputs for all but one.

9. REFERENCES

[1] R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3), May 2009. Article 16.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. of the Int. Conf. on Compiler Construction*, LNCS, pages 5–23. Springer, 2004.

[3] G. Balakrishnan and T. Reps. WYSINWYX: What you see is not what you eXecute. *ACM Trans. Program. Lang. Syst.*, 32(6), August 2010. Article 23.

[4] G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *Proc. of the Int. Symp. on Static Analysis*, pages 221–239. Springer, 2006.

[5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 203–213. ACM Press, 2001.

[6] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM System Journal*, 22:229–245, 1983.

[7] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 65–88. Springer, 2008.

[8] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. of the Int. Conf. on Automated Software Engineering*, pages 443–446. IEEE Comp. Soc., 2008.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the Conf. on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proc. of the Conf. on Computer and Communications Security*, pages 322–335. ACM, 2006.

[11] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 296–310. ACM, 1990.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[13] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. of the Symp. on Theory of Computing*, pages 151–158. ACM Press, 1971.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the Symp. on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[15] C. Csallner and Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. In *Proc. of the Int. Symp. on Software Testing and Analysis*, pages 245–254. ACM, 2006.

[16] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[17] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[18] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Int. J. Softw. Tools Technol. Transf.*, 5(2):247–267, 2004.

[19] S. Eker. Associative-commutative rewriting on large terms. In *Proc. of the Int. Conf. on Rewriting Techniques and Applications*, pages 14–29. Springer-Verlag, 2003.

[20] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *Int. J. Softw. Tools Technol. Transf.*, 6(2):117–127, 2004.

[21] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.

[22] A. Groce and W. Visser. Heuristic model checking for Java programs. In *Proc. of the Int. SPIN Workshop on Model Checking of Software*, pages 242–245. Springer, 2002.

[23] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *Proc. of the Int. Symp. on Foundations of Software Engineering*, pages 117–127. ACM, 2006.

[24] Hex-Rays. IDAPro disassembler, 2010.

[25] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., 1991.

[26] ISO. ISO C standard 1999. Technical report, ISO, 1999.

[27] J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proc. of the Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[28] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[29] A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In *Proc. of the Eur. Symp. on Programming Languages and Systems*, pages 246–263. Springer, 2006.

[30] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. of the Conf. on Prog. Lang. Design and Implementation*, pages 24–31. ACM, 1988.

[31] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the Conf. on Prog. Lang. Design and Implementation*, pages 190–200. ACM, 2005.

[32] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[33] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[34] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[35] C. Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.

[36] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Proc. of the Symp. on Principles of Programming Languages*, pages 119–132. ACM, 1999.

[37] T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *Proc. of the Symp. on Partial Evaluation and Semantics-based Program Manipulation*, pages 100–111. ACM, 2006.

[38] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the Symp. on Principles of Programming Languages*, pages 49–61. ACM, 1995.

[39] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58:206–263, 2005.

[40] A. Rybalchenko and R. Singh. Subsumer-first: Steering symbolic reachability analysis. In *Proc. of the Int. SPIN Workshop on Model Checking Software*, pages 192–204. Springer-Verlag, 2009.

[41] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proc. of the Int. Symp. on Software Testing and Analysis*, pages 225–236. ACM, 2009.

[42] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proc. of the Working Conference on Reverse Engineering*, pages 45–55. IEEE Comp. Soc., 2002.

[43] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[44] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proc. of the Int. Conf. on Information Systems Security*, volume 5352 of *LNCS*, pages 1–25. Springer, 2008.

[45] F. Tip. A survey of program slicing techniques. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1994.

[46] H. S. Warren. *Hacker's Delight*. Addison-Wesley, 2002.

[47] M. Weiser. Program slicing. In *Proc. of the Int. Conf. on Software Engineering*, pages 439–449. IEEE Press, 1981.

[48] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proc. of the Annual Design Automation Conference*, pages 599–604. ACM, 1998.

[49] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proc. of the Eur. Conf. on Computer Systems*, pages 321–334. ACM, 2010.

[50] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. of the Int. Symp. on Foundations of Software Engineering*, pages 97–106. ACM, 2004.