IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 8, 2019, accepted August 4, 2019, date of publication August 19, 2019, date of current version August 29, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2936139

A Value Set Analysis Refinement Approach Based on Conditional Merging and Lazy Constraint Solving

JIAN LIN, LIEHUI JIANG, YISEN WANG, AND WEIYU DONG
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China
Corresponding author: Jian Lin (ling_pro@163.com)

This work was supported by the National Natural Science Foundation of China under Grant 61802431.

ABSTRACT Value set analysis is a common static binary program analysis approach. Value set analysis attempts to identify a tight over-approximation of the program state at any given point in the program and can be used to detect vulnerability. Existing memory corruption detection analysis technologies based on value set analysis have a high false positive rate, because value set analysis suffers from a lack of accuracy. We observed that two main sources of imprecision in value set analysis are merge operation and failed branch conditions tracking. In order to address above problems, in this paper, we propose a value set analysis refinement approach based on conditional merging and lazy constraint solving. We propose a variable dependence analysis algorithm to divide program paths into subsets and only merge the states which satisfy the condition that the states are from the same subset, which reduces the imprecision from the merging operation. We collect path predicates as path constraint and solve the path constraint using Satisfiability Modulo Theories (SMT) solver lazily to get a tighter number range of the variable when a variable need be refined, which reduces the imprecision from the failed branch conditions tracking. We implement a prototype system RVSA based on the proposed approach and verify its effectiveness according to experimentation. Compared with state-of-the-art approach, the experimental results demonstrate that the false positive rate is reduced by 12.9%. Furthermore, using our proposed approach, 25 zero-day vulnerabilities are found in the Netgear *httpd* binary.

INDEX TERMS Value set analysis, value set analysis refinement, conditional static analysis, SMT solver, vulnerability detection.

I. INTRODUCTION

Binary vulnerability detection techniques can be roughly divided into two categories: static analysis and dynamic analysis [1], [2]. The dynamic analysis, such as fuzzing [3], [4] and dynamic taint analysis [5], [6], examines program behavior while it is running in a given environment. Dynamic binary analysis allows you explore individual paths which makes it very precise but at the expense of less code coverage. And the dynamic analysis needs an environment for execution. For programs in IoT devices, dynamic analysis approaches usually rely on the physical hardware [7] or emulator [8]. But acquiring hardware is expensive and not scalable [9], and building an accurate emulator for different devices is difficult and time-consuming. Static analysis generally reasons about a program without executing it. Static analysis lifts the binary program into assembly code or intermediate language, uses a specialized model to model program properties, and detects the properties using security strategy to find the vulnerability. Although static analysis is not accurate, it has high code coverage and is not limited by the execution environment.

Value Set Analysis (VSA) is a common static analysis approach which is based on abstract interpretation theory. The abstract interpretation was proposed by P. Cousot's lattice theory in 1977 to simplify and approximate the calculation of fixed points, and achieved a balance between the efficiency and accuracy [10]. VSA attempts to identify a tight over-approximation of the program state (i.e., values in memory and registers) at any given point in the program. It can be used to understand the possible targets of indirect jumps or the possible targets of memory read and write operations.

The associate editor coordinating the review of this article and approving it for publication was Pietro Savazzi.

The original design of VSA was proposed by Balakrishnan and Reps [11], [12], which was incorporated into a binary program static analysis platform called CodeSurfer/x86 [13].

Angr [14] implemented a VSA-based memory corruption detection analysis to detect buffer overflow vulnerability by checking the target address of the memory read and write operations, called angr-VSA. Evaluating using the DARPA CGC dataset [15], angr-VSA was able to identify 27 actual vulnerabilities while producing 130 false positives, resulting in a false positive rate of 82.8%. Too many false positives are unacceptable since each potential vulnerability need be examined further manually. So, false positive is a big problem of VSA.

The results of VSA are over-approximate, which suffer from a lack of accuracy. In VSA, the fully merging strategy is adopted. For two states which reach to the same basic block, the VSA will merge the two states into a new merged state, and the merge operation usually causes a certain precision loss. Tracking branch conditions helps us constrain variables in a state after taking a conditional exit, which produces a more precise analysis result. When a new path predicate is seen, VSA will apply a solver to solve it. But if the solver is too heavyweight, it is time consuming and impractical. If the solver is too lightweight, it will fail to track some complex branch conditions and cause a precision loss.

In order to increase the precision of VSA, we propose a value set analysis refinement approach based on conditional merging and lazy constraint solving, which can prune the false positives automatically. Instead of fully merging strategy, we divide program paths into subsets by our proposed variable dependence analysis algorithm, and conditionally merge the abstract states from the same subset, which reduces the imprecision introduced by the merging operation. We also apply a heavyweight solver: Satisfiability Modulo Theories (SMT) [16] solver for lazy constraint solving. Lazy constraint solving collects path predicate as path constraint of state during VSA. Later, when a variable need be refined, we use SMT solver to solve path constraint to obtain a tighter number range of the variable, which reduces the imprecision from failed branch conditions tracking.

We implement a prototype system RVSA (Refined Value Set Analysis) based on our approach and evaluate on DARPA CGC dataset and Netgear *httpd* binary. The experimental result shows that false positive rate of RVSA reduced by 12.9%, compared to angr-VSA.

In summary, we make the following contributions:

- We propose a variable dependence analysis algorithm to divide program paths into subsets. Each subset corresponds to a set of paths with same data dependence. Value set analysis on each subset separately can get a more precise result.
- We propose a value set analysis refinement approach based conditional merging and lazy constraint solving to prune false positives. Conditional merging only merges the states from the same subsets. Lazy constraint solving collects path predicate as path constraint of state and use SMT solver to solve path constraint lazily when the variable need be refined. The proposed approach can reduce the imprecision from the merge operation and failed branch conditions tracking, and increase the accuracy of VSA.
- We implement a prototype system and verify its effectiveness. The experiment demonstrates that the false positive rate is reduced by 12.9% than the state-of-the-art approach and 25 zero-day vulnerabilities are found in the Netgear *httpd* binary.

The remainder of this paper is organized as follows. Section II introduces the traditional VSA, and illustrate the limitation of VSA on a motivating example. The design of our approach is presented in section III. The implementation of our approach is provided in sections IV. An evaluation of our approach is presented in section V, discussion and future work are explained in section VI, related work is introduced in section VII, and section VIII concludes the paper.

II. BACKGROUND

A. VALUE SET ANALYSIS

VSA is a combined numeric-analysis and pointer-analysis algorithm that determines an over-approximation of the set of numeric values or addresses that each register and memory location holds at each program point [12]. The abstract domain of VSA includes memory region, value set and abstract state.

1) MEMORY REGION

During the analysis of an executable, VSA breaks the address space into a set of disjoint memory areas, which are referred to as memory regions. Every memory region has a region identifier: $RegionId$. For a given program, there are three kinds of regions: (1) the global-region contains information about locations that correspond to global data, (2) the AR-regions contain information about locations that corresponds to the activation record of a particular procedure, and (3) the Malloc-Regions contain information about locations that are allocated at a particular malloc site. Each memory region represents a group of abstract locations that have similar runtime properties. Abstract location represents the address of the variable, called $a - loc$.

2) VALUE SET

A value set is a safe approximation for a set of addresses and numeric values. Suppose that n is the number of regions in the executable. A value set is an n-tuple of strided intervals of the form $s\,[l, u]$, with each component of the tuple representing the set of addresses in the corresponding region. A strided-interval $s\,[l, u]$ represents the set of integers $\gamma\,(s\,[l, u]) := \{i\,|l \leq i \leq u,\ i \equiv l\,(\mathrm{mod}\,s)\}$. The s is called the stride, the $[l, u]$ is called the interval. When s is 0, $0\,[l, l]$ represents the singleton set $\{l\}$.

3) ABSTRACT STATE

We define the memory region as a map from the a-loc to value set: $Region\ := \ a - loc\ \rightarrow\ ValueSet$, and define the abstract state as a map from region identifier to region: *State* := *RegionId* → *Region*. VSA will identify an abstract state at each program point.

Algorithm 1 Value Set Analysis Algorithm

```
Input: program's control flow graph $CFG$
Output: abstract states $S$
 1: $b_0 = EntryBlock(CFG)$
 2: $W = \{b_0\}$
 3: $S[b_0] = InitState()$
 4: while $W \neq \emptyset$ do
 5:     $b = W.removeNext()$
 6:     $states = Analysis(b, S[b])$
 7:     for $s_n \in states$ do
 8:         $b_n = s_n.block$
 9:         if $b_n$ in $S$ then
10:             if $s_n \neq S[b_n]$ then
11:                 $S[b_n] = Merge(S[b_n], s_n)$
12:                 $W.add(b_n)$
13:             end if
14:         else
15:             $S[b_n] = s_n$
16:             $W.add(b_n)$
17:         end if
18:     end for
19: end while
20: return $S$
```

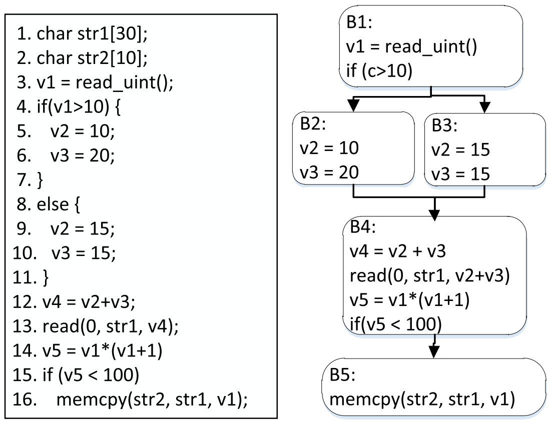
The VSA algorithm is shown in Algorithm 1. The set *W* is here called the work-list [17] with operations *add* and *removeNext* for adding and removing an item. The work-list is sorted by topological order and initially contains the entry basic block, indicating forward analysis from the entry basic block.

In each iteration of the while-loop, the *Analysis* function will be called to analysis the selected basic block at Line 6. The *Analysis* function will generate multiple output states based on the input state. Those output states that changing are added to the work-list.

When there are two input states at the same basic block, VSA will merge the two input states as a new input state at Line 11. The merging operation will merge the value set of every variable in the abstract states. The merging operation equation of a variable is as shown in equation 1, where s_m is generally the maximum common divisor of s_1, s_2 and absolute value of $l_2 - l_1$, l_m is the minimum value of l_1 and l_2, and u_m is the maximum value of u_1 and u_2.

$$\begin{aligned} s_m[l_m, u_m] &= s_1[l_1, u_1] \cup s_2[l_2, u_2] \\ &= s_m[\min(l_1, l_2), \max(u_1, u_2)] \end{aligned} \quad (1)$$

When the program reads or writes memory, according to the value set of the destination address and length of the memory read or write, we can detect whether it exceeds the reversed space of corresponding variable. If exceeds, it is reported as a potential vulnerability. However, because the value set obtained by the VSA is over-approximate, not accurate, it will result in a high false positive rate.



```
char str1[30];
char str2[10];
v1 = read_uint();
if(v1>10) {
   v2 = 10;
   v3 = 20;
}
else {
   v2 = 15;
   v3 = 15;
}
v4 = v2+v3;
read(0, str1, v4);
v5 = v1*(v1+1)
if (v5 < 100)
   memcpy(str2, str1, v1);
```

FIGURE 1. **Motivating example.**

B. MOTIVATING EXAMPLE

A motivating example where false positives occur is shown in Figure 1. The source code of the example is on the left. The Control Flow Graph (CFG) with five basic blocks is on the right. VSA will produce two false positives, including the *read* function at basic block *B4* and the *memcpy* function at basic block *B5*.

There are two paths *B1->B2->B4* and *B1->B3->B4*, which can reach *B4*. No mater which path it takes, the value of variable *v4* is 30, so there will be no overflow at *B4*. However in VSA, the output states of *B2* and *B3* will be merged at the entrance of *B4*. In the output state of *B2*, the value set of *v2* is 0[10, 10], and the value set of *v3* is 0[20, 20]. In the output state of *B3*, the value set of *v2* is 0[15, 15], the value set of *v3* is 0[15, 15]. In the merged state, the value set of *v2* will be 5[10, 15], the value set of *v3* will be 5[15, 20]. When the merged state is used as the input state to analyze *B4*, the *read* function performs a memory write operation. The write target address is *str1* and the write length is *v4*. The value set of *v4* is 5[25, 35], and the reserved size of the *str1* is 30. The maximum value of *v4* exceeds the size of *str1*, so an overflow may occur and a potential vulnerability will be reported. The reason for this false positive is that VSA is not path sensitive, so the state merging at the entrance of *B4* expands the value set of *v4*.

At *B5*, the *memcpy* function also performs a memory write operation with a target address of *str2* and a size of *v1*. Arriving at *B5* requires input to satisfy path constraint *v1*(v1+1)<100*. Obviously the value of *v1* cannot exceed 10, so no overflow occurs. However the value set of *v1* from VSA is $1\left[0, 2^{32} - 1\right]$, this is a false positive. The main reason for this false positive is that the numeric abstraction domain is used in the VSA, and the relationship between variables is ignored. Considering this branch condition, only the value set of *v5* is limited to 1[0, 99], and there is no way to limit *v1* which has a numerical relationship with *v5*. Affine-Relation Analysis has been proposed as a technique to

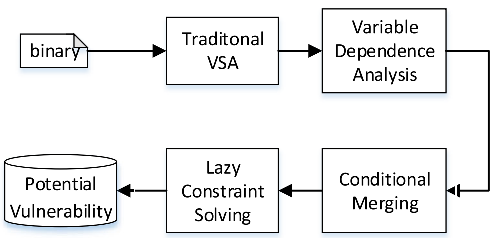


FIGURE 2. Approach overview.

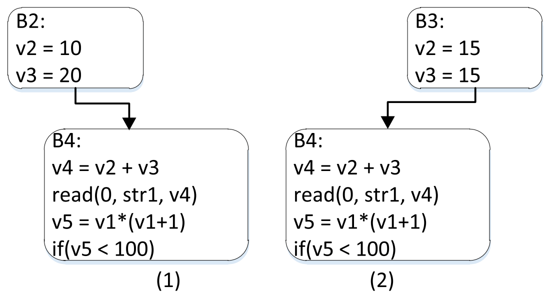


FIGURE 3. Variable Ddependence Ssubgraphs of variable (*B4*, *v4*).

obtain relations among the variables [18]. However, it is both complicated to implement, and is computationally expensive in reality. Angr applies an algebraic solver to path predicates. The algebraic solver attempts to simplify and solve path predicate to obtain a number range for the variables involved in the path predicate. But if the path predicate has multiple variables or non-linear constraint, the algebraic solver will fail to simplify and solve the path predicate. Here, the path predicate *v1*(v1+1)<100* has non-linear constraint, so the path predicate cannot be solved to get the number range of variable *v1*.

III. DESIGN

A. OVERVIEW

The main reason why VSA produces false positives is that the value of the destination address and length may be inaccurate due to over-approximate results of VSA. Therefore, refining the value set of the destination address and length to obtain a more accurate value set can effectively reduce false positive rate.

This paper proposes a value set analysis refinement approach based on conditional merging and lazy constraint solving. The approach overview is shown in Figure 2. The following is a description of how the approach works in conjunction with the motivating example in Section II-B.

First, we use the traditional VSA to analyze the binary program and get pairs of basic block and variables which may overflow. When analyzing binary program, the memory object and register are treated as variables, and the address of variable is represented as $a - loc$. The results obtained in the example program will be (*B4*, *v4*) and (*B5*, *v1*), indicating that the variable *v4* at *B4* and the variable *v1* at *B5* may cause a vulnerability. Therefore, the next steps are to refine these variables. We call the variable that needs be refined as the target variable, and the basic block of the target variable as the target basic block.

Then, we propose a new algorithm for variable dependence analysis to divide program paths into subsets (Section III-B). Variable dependence analysis of (*B4*, *v4*) in the example program will get two subsets, as shown in Figure 3.

Next, we use the conditional merging VSA to analyze the binary program (Section III-C). Only states from the same subsets are merged. So the state from *B2* to *B4* and the state from *B3* to *B4* come from different subsets, so they are not merged. Thus, there will be two states at *B4*, and the value sets of *v4* in the two states are detected separately, both are 0[30, 30], so the variable (*B4*, *v4*) will not overflow.

Finally, Path predicate is collected as path constraint of abstract state in VSA. After reaching the target basic block, we use a SMT solver to solve the corresponding path constraint lazily and get a tighter number range of the target variable (Section III-D). In the example program, when B5 is reached, the path constraint is *v1*(v1+1)<100*. The number range of *v1* from the SMT solver is [0, 9], so the value set of *v1* is 1[0, 9], and the variable (*B5*, *v1*) will not overflow.

B. VARIABLE DEPENDENCE ANALYSIS

In order to describe variable dependence analysis, we present two new notions: Variable Dependence and Variable Dependence Subgraph.

Definition 1 (Variable Dependence): For an execution path that reaches the target basic block, back tracking the target variable will know which basic blocks are involved in the computation of the final value of the target variable. We define the set of these basic blocks as Variable Dependence of this path.

Definition 2 (Variable Dependence Subgraph): Variable Dependence Subgraph is a subgraph of CFG. All paths in a Variable Dependence Subgraph have the same Variable Dependence, start with the first basic block of the Variable Dependence and end with the target basic block.

If two execution paths have the same Variable Dependence, the value set of the target variable is affected by the same instructions. So only states from the same Variable Dependence Subgraph can been merged. We propose a variable dependence analysis algorithm to generate all Variable Dependence Subgraphs for target variable.

If there are loops in the program, it is impossible to get all execution paths, which is known as path explosion problem. We use loop unrolling to mitigate this problem. The number of loop unrolls is set to 2, indicates that the loop body can execute once or more. Variable dependency analysis is to find all Variable Dependencies, it is enough to loop unrolls twice. After loop unrolling, we will get a new CFG, which is a directed acyclic graph.

Variable dependency analysis algorithm is shown in Algorithm 2. The algorithm takes the target variable v_0, the target basic block b_0, and the loop-unrolled *CFG* as inputs. The set *W* is here called the work-list with operations

Algorithm 2 Variable Dependence Analysis Algorithm

```
Input: target variable $v_0$, target basic block $b_0$,
    loop-unrolled $CFG$
Output: Variable Dependence Subgraphs $G$
1: $g_0 = EmptyGraph()$
2: $d_0 = []$
3: $vs_0 = [v_0]$
4: $W = \{(b_0, vs_0, d_0, g_0)\}$
5: $G = \{\}$
6: while $W \neq \emptyset$ do
7:     $b, vs, d, g = W.removeNext\ ()$
8:     $vs_d = GetDefineVar(vs, b)$
9:     if $vs_d \neq \emptyset$ then
10:        $vs_u = AnalysisBlock\ (vs_d)$
11:        $vs = (vs \backslash vs_d) \cup vs_u$
12:        $d = d.add(b)$
13:        if $vs = \emptyset$ then
14:            $G[d] = G[d] \cup g$
15:            continue
16:        end if
17:    end if
18:    for $b' \in b.prevs$ do
19:        $g' = g.addEdge(b', b)$
20:        $W.add\ ((b', vs, d, g'))$
21:    end for
22: end while
23: return $G$
```

add and *removeNext* for adding and removing an item. The item is a tuple of four elements (b, vs, d, g). The element b is the basic block to be analyzed, element vs is a set of variables whose definition has not been analyzed, element d is the Variable Dependence which has been analyzed, element g is the corresponding Variable Dependence Subgraph. The W is sorted by topological reverse order of basic blocks and initially contains the target basic block, indicating backward analysis from the target basic block.

In each iteration of the while-loop, an item will been selected to be analyzed. At Line 8, the function *GetDefineVar* will get a set vs_d, which is a subset of vs and whose element is defined in the basic block b. This can be obtained quickly by Use-Define Chain. A Use-Define Chain is a data structure that consists of a use of a variable, and all the definitions of that variable that can reach that use without any other intervening definitions. For every used variable v in vs, we can get all basic blocks that define the variable v. If these basic blocks contain the current basic block b, so v is define in basic block b and is added to set vs_d.

If the basic block b has defined variables vs_d, at Line 10, the function *AnalysisBlock* will be called to analyze all instructions of the basic blocks according to the reverse order and get all variables vs_u that are involved in the computation of the value of variables vs_d. Then, we update the vs through removing vs_d and adding vs_u.

Algorithm 3 Marking Basic Blocks Algorithm

```
Input: the number of subgraphs $n$, subgraphs
    $\{CFG_0, \ldots, CFG_{n-1}\}$
Output: marked basic blocks $MB$
1: $MB = \{\}$
2: for $i = 0$ to $n$ do
3:     for $j = i + 1$ to $n$ do
4:         $CFG_c = CommonSubGraph(CFG_i, CFG_j)$
5:         $edges = IncomingEdges(CFG_i, CFG_c)$
6:         $b = NearestPublicParentVertex(edges)$
7:         $MB.append(b)$
8:     end for
9: end for
10: return $MB$
```

If vs is an empty set, indicating that all the variables involved in this path have been analyzed, so we store the final Variable Dependence Subgraph g into G at Line 14. G is a map from Variable Dependence to Variable Dependence Subgraph. If vs is not an empty set, indicating we need to continue the backward analysis, thus we add the previous basic blocks to the work-list W at Line 20.

C. CONDITIONAL MERGING

By variable dependence analysis, we can get all Variable Dependence Subgraphs of the target variable. If two states that reach the same basic block are from the same Variable Dependence Subgraph, then the two states can be merged, otherwise they cannot be merged.

So we need to know which Variable Dependence Subgraph the state comes from. An easy way is to look up all the basic blocks which the state traverses. But this is expensive because it will compare all traversed basic blocks of the two states at each merge points. We want to identify a subset of basic blocks such that different Variable Dependence Subgraphs remain differentiable. Our goal is to mark some basic blocks on the CFG such that, for each pair of Variable Dependence Subgraph, we are able to distinguish them by only taking those marked basic blocks into consideration. That is, a path is now represented by the sequence of marked basic blocks it traverses instead of all of the traversed basic blocks.

Because all Variable Dependent Subgraphs end with the target basic block, any two Variable Dependence Subgraphs have a largest common subgraph which contains the target basic block. Any Variable Dependence Subgraph has one or more incoming edges into the largest common subgraph. Back tracking these edges will get a nearest public parent basic block. Marking this basic block can distinguish this Variable Dependence Subgraph from another Variable Dependence Subgraph.

The algorithm for selecting marked basic blocks is shown in Algorithm 3. The algorithm takes the number of Variable Dependence Subgraphs n and all Variable Dependence Subgraphs CFG_0 to CFG_{n-1}. For any two Variable

Dependence Subgraphs CFG_i and CFG_j, we use function *CommonSubGraph* to get the largest common subgraph CFG_c At Line 4. Then, we get all edges which enter CFG_c from CFG_i, get the nearest public parent basic block b of the edges and add b to marked basic blocks MB.

After the marked basic blocks are obtained, we need reanalyze the program using conditional merging VSA. Conditional merging only affects the states of the basic blocks in the Variable Dependence Subgraphs, so instead of reanalyzing the entire program, we only reanalyze the union of all Variable Dependence Subgraphs CFG_u, as shown in Equation 2.

$$CFG_u = CFG_0 \cup CFG_1 \cup \ldots \cup CFG_{n-1} \tag{2}$$

We modify traditional VSA algorithm to support conditional merging. The algorithm of conditional merging VSA is shown in Algorithm 4. Compared with the traditional VSA, there are the following differences. First, at Line 1-6, the initial value of the work-list is no longer the program entry basic block, but all the nodes whose in-degree is 0. The initial state of these nodes is the state obtained from the traditional VSA states S_0. Second, at Line 9-10, the state of a basic block is no longer a single state, and multiple states may exist at the same time. Therefore, list is used to store the states of a basic block and output states need be generated for each input state separately. Third, at Line 12-13, after a basic block analyzed, check whether the basic block is in the marked basic blocks MB, and if so, add the basic block to the *mb* set of the state. Finally, when need to merge, we use function *ConditionalMerge* to do conditional merging at Line 21. In function *ConditionalMerge*, only the states with the same *mb* set are merged.

D. LAZY CONSTRAINT SOLVING

Tracking branch conditions helps us constrain variables in a state after taking a conditional exit, which produces a more precise analysis result. When a new path predicate is seen (i.e., when following a conditional branch), traditional VSA attempts to simplify and solve it to obtain a number range for the variables involved in the path predicate. However there are many path predicates, so the traditional VSA only use a lightweight solver with limited ability.

We additionally apply a heavyweight constraint solver: SMT solver. Instead of solving at every time when a new path predicate is seen, we collect the path predicate as path constraint of the state. Later, when value set of a variable need be refined, we simplify path constraint and solve it using the SMT solver.

Firstly, we need collect path predicate as path constraint during the VSA. For each abstract state in VSA, we use σ to represent the corresponding path constraint. The σ of the initial state is null. When a conditional branch (the path predicate is e) is followed, VSA will have two output states with jump or not jump. For the state with jump, update the path constraint to $\sigma \wedge e$. For the state with not jump, update the path constraint to $\sigma \wedge \neg e$.

Algorithm 4 Conditional Merging Value Set Analysis Algorithm

```
Input: union of subgraphs $CFG_u$, initial states $S_0$, marked
    basic state $MB$
Output: abstract states $S$
1:  $bs_0 = EntryBlocks(CFG_u)$
2:  $W = \{\}$
3:  for $b_0$ in $bs_0$ do
4:      $W.add\ (b_0)$
5:      $S\ [b_0] = \{S_0\ [b_0]\}$
6:  end for
7:  while $W \neq \emptyset$ do
8:      $b = W.removeNext()$
9:      for $inState$ in $S[b]$ do
10:         $states = Analysis(b, inState)$
11:         for $s_n \in states$ do
12:             if $b$ in $MB$ then
13:                 $s_n.mb.add(b)$
14:             end if
15:             $b_n = s_n.block$
16:             if $b_n$ not in $CFG_u$ then
17:                 continue
18:             end if
19:             if $b_n$ in $S$ then
20:                 if $s_n$ not in $S[b_n]$ then
21:                     $S[b_n] = ConditionalMerge(s_n, S[b_n])$
22:                     $W.add\ (b_n)$
23:                 end if
24:             else
25:                 $S\ [b_n]\ .add(s_n)$
26:                 $W.add(b_n)$
27:             end if
28:         end for
29:     end for
30: end while
31: return S
```

When two states are merged, the path constraint update to σ_m=σ_1 |σ_2 . In most cases, the merged state comes from the two states separated from the previous conditional branch, so σ_1 and σ_2 have some common path predicates $sigma_0$. Therefore, the merged path constraints can be simplified, and the simplified equation is shown as Equation 3.

$$\begin{aligned} \sigma_0 &= CommonPredicates\,(\sigma_1, \sigma_2) \\ \sigma_1{}' &= \sigma_1 - \sigma_0 \\ \sigma_2{}' &= \sigma_2 - \sigma_0 \\ \sigma_m &= \begin{cases} \sigma_0 & if\ \sigma_1{}' = \neg\sigma_2{}' \\ \sigma_0 \wedge \left(\sigma_1{}' | \sigma_2{}'\right) \end{cases} \end{aligned} \tag{3}$$

Then, when we need to refine a variable (i.e. v), the path constraint is further simplified for the variable v. The path predicates unrelated to the variable v are removed. Based on the previous variable dependence analysis, a set of all variables related to variable v can be obtained as a set vs.

For a single path predicate σ, if all variables in σ are not in vs, it means σ is not related to the variable v. When the path constraint has multiple path predicates connected by operation $\wedge$ (i.e. $\sigma = \sigma_1 \wedge \sigma_2 \wedge \ldots \wedge \sigma_n$), for every path predicate σ_i, σ_i can be removed from the path constraint if σ_i is not related to the variable v. When path constraint has multiple path predicates connected by operation | (i.e., $\sigma = \sigma_1|\sigma_2|\ldots|\sigma_n$), if there exists a path predicate σ_i not related to the variable v, then the entire path constraint σ is also considered to be not related to variable v.

Finally, we use the SMT solver to solve the minimum and maximum values of the variable v as the number range of the variable. The newly generated number range is generally within the range of the original value set. The newly generated number range is only determined by the collected path constraint. The path constraint may be not complete, such as the use of widening operators. So the newly generated number range may not within the range of the original value set in this case. We perform an intersection between the newly generated number range and the original value set of variable v. The equation is shown in Equation 4.

$$s\left[l', u'\right] = s\left([l, u] \cap [\min(v|\sigma), \max(v|\sigma)]\right) \quad (4)$$

IV. IMPLEMENTATION

We implement a prototype system using the approach proposed in this paper, called RVSA (Refined Value Set Analysis).

We use angr-VSA as traditional VSA. In angr-VSA, The main interface of VSA is the Value Flow Graph (VFG). The VFG is an enhanced CFG that includes the program state representing the VSA fix-point at each program location. The program states contained in the VFG present memory in an abstract layout provided by the *SimAbstractMemory* memory model, with values in memory represented by value-sets, as provided by *Claripy*. The memory read and write operations are hooked by *SimInspect*, and then we detect whether the variable is out-of-bounds access by the value set of the destination address and length of the memory read and write to find potential vulnerabilities. We use angr-VSA to get target basic blocks and target variables.

For each variable that need be refined, we use the *VSA_DDG* analysis of angr to get the use-define chain and get Variable Dependence Subgraphs by variable dependence analysis algorithm. Then we use conditional merging VSA to get multiple states of target basic block from different Variable Dependence Subgraphs. Finally, we detect vulnerability in each state separately.

Angr lift binary to VEX intermediary language, and use engine *SimEngineVex* to execution VEX. When execution, we can get the path predicate, which is the *claripy.ast.bool* type. For each state, we also get a path constraint. We use Z3 as our SMT solver [19]. Z3's main functionality is to check the satisfiability of logical formulas over one or more theories. Z3 can produce models for satisfiable formulas. Besides, Z3 also can solve optimization problems over SMT formulas, MaxSMT, and their combinations [20]. We use Z3 to solve the path constraint to get the maximal and minimal of target variable.

V. EVALUATION

To evaluate RVSA, this section attempts to answer the following questions:

Effectiveness of vulnerability detection. How effective is RVSA's approach in reducing the false positive rate of VSA-based vulnerability detection analysis when analysis complex, real-world software? (Section V-B).

Justifying design decisions. How effective are the design decisions made by RVSA including conditional merging and lazy constraint solving in terms of false positives pruning? (Section V-C and V-D).

Performance. How much performance overhead does RVSA increase, including running time and memory consumption? (Section V-E).

A. EXPERIMENTAL SETUP

We evaluated our system on a Lenovo desktop equipped with an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz with 4 cores and 16 GB RAM, running Linux Ubuntu 18.04 TLS.

In these experiments, we compared the results reported by both RVSA and angr-VSA. The reason why angr-VSA was chosen for comparison is twofold: 1) angr is a renowned static analysis tool for binaries and VSA is an important feature of angr; and 2) to the best of our knowledge, angr-VSA achieves one of the lowest false positive rate among existing VSA-based memory corruption detection tools.

In order to compare with angr-VSA, we choose the same DARPA CGC dataset [15]. As part of Cyber Grand Challenge, DARPA released a set of binaries that run in a customized OS called DECREE. There are 131 services in this dataset, but 5 of these involve communication between multiple binaries, so we only consider the 126 single-binary applications. For each binary in the dataset, the analysis begins with the *main* function.

To verify the ability for detecting real-world vulnerabilities, we also choose the HTTP server of Netgear R6400 Nighthawk Routers as dataset. The latest version of the device firmware is R6400-V1.0.1.46_1.0.32, the http server binary is */usr/sbin/httpd*, and the *httpd* binary can be obtained by unpacking the firmware using binwalk. *Httpd* is developed by Netgear vendors and not open source. The size of *httpd* binary is 1.6MB, so the binary is more complicated. In order to simplify the analysis, the analysis doesn't begin with the *main* function. Preliminary analysis of the program, we found there are a dispatch table in memory address 0x12299C and 174 handler functions in this dispatch table. Accessing different *cgi* via http post request, the corresponding handler function will be executed and the http post data is the first argument of the handler function. So, the analysis starts with these handler functions, and treats the first parameter as the input content.

TABLE 1. Vulnerability detection result.

Tool	Dataset	Total Warnings	Bugs	False Positive	FPR
angr VSA	CGC	157	27	130	82.80%
	Netgear *httpd*	234	30	204	87.10%
	total	391	57	334	85.40%
RVSA	CGC	97	27	70	72.10%
	Netgear *httpd*	111	30	81	72.90%
	total	201	57	151	72.50%

B. EFFECTIVENESS OF VULNERABILITY DETECTION

The vulnerability detection result is shown in Table 1. We define FP for the number of false positive, which has no vulnerability and is determined to be potential vulnerability. We define Bugs for the number of found actual vulnerability. Then false positive rate is calculated as Equation 5.

$$FPR = \frac{FP}{FP + Bugs} \tag{5}$$

The number of bugs found by RVSA is the same with angr-VSA. It proves RVSA has not missed any of actual vulnerability. In DARPA CGC dataset, we found 27 actual vulnerabilities in 19 different binaries, mainly stack overflow vulnerabilities. In Netgear *httpd* binary, 30 vulnerabilities were detected and submitted to Netgear Seurity Team. After further investigation by the Netgear team, they were previously made aware of 5 vulnerabilities as they received them from another researcher. The remaining 25 vulnerabilities were received as zero-day vulnerability, and granted as the vulnerability ID of the vendor, as shown in Table 2. Among the 25 zero-day vulnerabilities, there are 19 stack overflow vulnerabilities, 5 out-of-bounds read and write vulnerabilities and one null pointer dereference vulnerability. Since Netgear has not released new patches, more details do not allow to be disclosed.

RVSA has lower false positive rate than angr-VSA. Through conditional merging and lazy constraint solving by RVSA, approximately half of the false positives are effectively pruned. In DARPA CGC dataset, the number of false positives was reduced by 60, and the false positive rate was reduced by 10.7%. In Netgear *httpd* binary, the number of false positives reduced by 123, and the false positive rate reduced by 14.2%. In summary, the false positives reduced from 334 to 151, and the false positive rate reduced by 12.9%.

The result of pruned false positives is show in Table 3. The number of false positives pruned by conditional merging is 121, and the number of false positive pruned by lazy constraint solving is 62. The ratio is about 2:1.

TABLE 2. Zero-day vulnerabilities in Netgear *httpd* binary.

Vulnerability type	num	Vulnerability ID
Stack overflow vulnerability	19	PSV-2019-0058, PSV-2019-0107, PSV-2019-0115, PSV-2019-0119, PSV-2019-0143, PSV-2019-0144, PSV-2019-0145, PSV-2019-0146, PSV-2019-0147, PSV-2019-0148, PSV-2019-0149, PSV-2019-0153, PSV-2019-0154, PSV-2019-0155, PSV-2019-0156, PSV-2019-0166, PSV-2019-0167, PSV-2019-0168, PSV-2019-0169
Out-of-bounds read and write vulnerability	5	PSV-2019-0103, PSV-2019-0104, PSV-2019-0105, PSV-2019-0105, PSV-2019-0118
Null pointer dereference vulnerability	1	PSV-2019-0120
total	25	

TABLE 3. Result of pruned false positives.

Dataset	Initial FP	pruned FP By conditional merging	pruned FP by lazy constraint solving	Total pruned FP
CGC binary	130	39	21	60
Netgear httpd	204	82	41	123
total	334	121	62	183

C. CONDITIONAL MERGING

We use conditional merging to refine the target variable for the total 334 false positives. The result of conditional merging for pruning false positives is shown in Figure 4. For each false positive, we can get the number of Variable Dependence Subgraphs. The number of Variable Dependence Subgraphs at least 1 and up to 30. If the number is 1, the state cannot be decomposed, so these false positive cannot be pruned. The number of false positives is shown as red line. As the number of Variable Dependence Subgraphs increases, the number of false positives decreases, indicating most abstract states only can decomposed into a few states. The pruned false positive is shown as the blue line. As the number of Variable Dependence Subgraphs increases, the results are more precise and the rate of pruning increases.

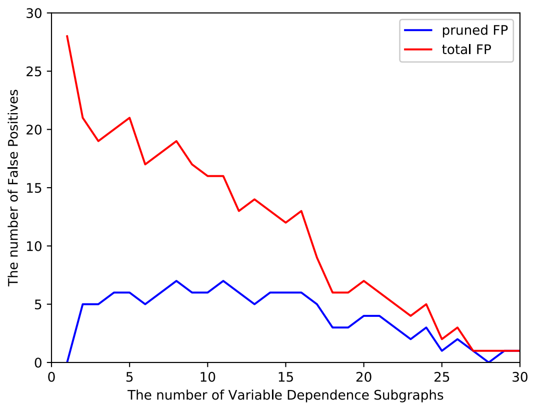


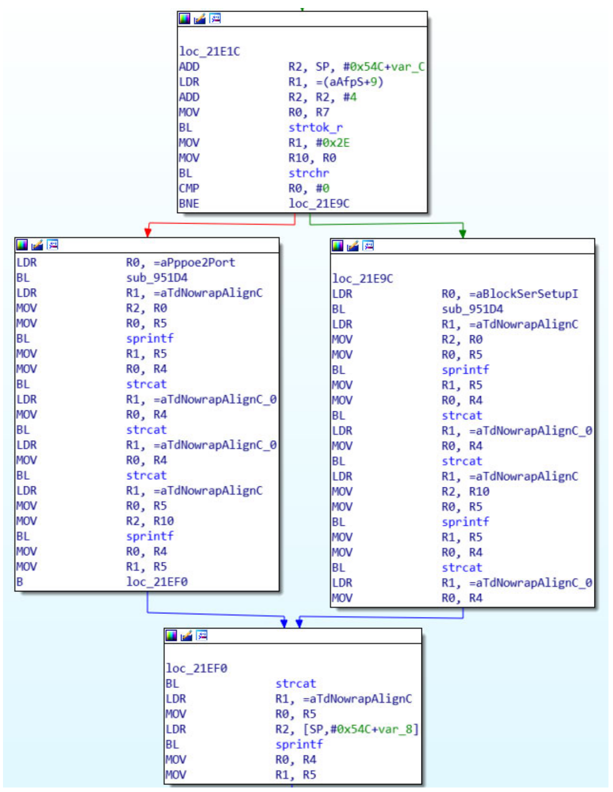
FIGURE 4. Conditional merging for pruning false positive.

```
v10 = strtok_r(v18, ",", (char **)&v19);
if ( strchr(v10, 46) )
{
  v13 = sub_951D4("block_ser_setup_ip_jp");
  sprintf(v16, "<td nowrap align=\"center\"><b><span class=\"ttext\">%s</span></b></td>", v13);
  strcat(v15, v16);
  strcat(v15, "<td nowrap align=\"center\"><b><span class=\"ttext\">---</span></b></td>");
  sprintf(v16, "<td nowrap align=\"center\"><b><span class=\"ttext\">%s</span></b></td>", v10);
  strcat(v15, v16);
  v12 = "<td nowrap align=\"center\"><b><span class=\"ttext\">---</span></b></td>";
}
else
{
  v11 = sub_951D4("pppoe2_port");
  sprintf(v16, "<td nowrap align=\"center\"><b><span class=\"ttext\">%s</span></b></td>", v11);
  strcat(v15, v16);
  strcat(v15, "<td nowrap align=\"center\"><b><span class=\"ttext\">---</span></b></td>");
  strcat(v15, "<td nowrap align=\"center\"><b><span class=\"ttext\">---</span></b></td>");
  sprintf(v16, "<td nowrap align=\"center\"><b><span class=\"ttext\">%s</span></b></td>", v10);
  v12 = v16;
}
strcat(v15, v12);
sprintf(v16, "<td nowrap align=\"center\"><b><span class=\"ttext\">%s</span></b></td>", v19);
v9 = v16;
```

FIGURE 5. Decompiled pseudocode of function *sub_21D10*.

Case Study. In the program of Netgear *httpd*, when handling the "pppoe2.cgi" request, the function *sub*_21*D*10 will be executed. Part of the function's decompiled pseudocode is shown in Figure 5, and the CFG is shown in Figure 6. The variables *v10*, *v12* and *v15* are string type, and the lengths of these variables are represent as *len(v10)*, *len(v12)* and *len(v15)*. The reversed size of the variable *v15* is 1024 bytes, so the maximum of *len(v15)* should not be more than 1024. When the string *v10* contains the character '.', it will be executed into the true branch, otherwise it will enter the false branch. Both in the true and false branches, *v15* and *v12* are assigned separately. In the true branch, the value set of *len(v15)* is 1[198, 710], and the value set of *len(v12)* is 0[70, 70]. In the false branch, the value set of *len(v15)* is 0[198, 198], and the value set of *len(v12)* is 1[70, 582]. If the two states merge, the value set of *len(v15)* becomes 1[198,710], the value set of *len(v12)* becomes 1[70,582]. Then the *strcat(v15, v12)* will have a memory write operation which target address is *v15+len(v15)* and size is *len(v12)*. The maximum value of *len(v15)+len(v12)* is 1292, which exceeds the size of 1024 reserved by the variable *v15*, so angr-VSA will report this as a potential vulnerability.

The variable dependence analysis algorithm will divide the true branch and the false branch into two different Variable Dependence Subgraphs, so the two state will not be merged. For each state, the maximum value of *len(v15)+len(v12)* will be calculated separately, so it will not exceed the reserved 1024 bytes of variable *v15*. We can prune this false positive.



```
loc_21E1C
ADD                 R2, SP, #0x54C+var_C
LDR                 R1, =(aAfpS+9)
ADD                 R2, R2, #4
MOV                 R0, R7
BL                  strtok_r
MOV                 R1, #0x2E
MOV                 R10, R0
BL                  strchr
CMP                 R0, #0
BNE                 loc_21E9C
```

```
LDR                 R0, =aPppoe2Port
BL                  sub_951D4
LDR                 R1, =aTdNowrapAlignC
MOV                 R2, R0
MOV                 R0, R5
BL                  sprintf
MOV                 R1, R5
MOV                 R0, R4
BL                  strcat
LDR                 R1, =aTdNowrapAlignC_0
MOV                 R0, R4
BL                  strcat
LDR                 R1, =aTdNowrapAlignC_0
MOV                 R0, R4
BL                  strcat
LDR                 R1, =aTdNowrapAlignC
MOV                 R0, R5
MOV                 R2, R10
BL                  sprintf
MOV                 R0, R4
MOV                 R1, R5
B                   loc_21EF0
```

```
loc_21E9C
LDR                 R0, =aBlockSerSetupI
BL                  sub_951D4
LDR                 R1, =aTdNowrapAlignC
MOV                 R2, R0
MOV                 R0, R5
BL                  sprintf
MOV                 R1, R5
MOV                 R0, R4
BL                  strcat
LDR                 R1, =aTdNowrapAlignC_0
MOV                 R0, R4
BL                  strcat
LDR                 R1, =aTdNowrapAlignC
MOV                 R2, R10
MOV                 R0, R5
BL                  sprintf
MOV                 R1, R5
MOV                 R0, R4
BL                  strcat
LDR                 R1, =aTdNowrapAlignC_0
MOV                 R0, R4
```

```
loc_21EF0
BL                  strcat
LDR                 R1, =aTdNowrapAlignC
MOV                 R0, R5
LDR                 R2, [SP,#0x54C+var_8]
BL                  sprintf
MOV                 R0, R4
MOV                 R1, R5
```

FIGURE 6. CFG of function *sub_21D10*.

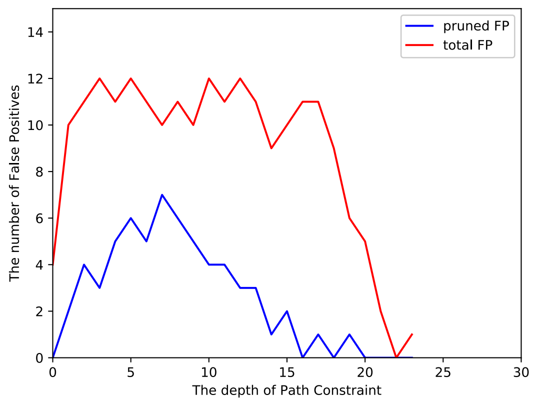
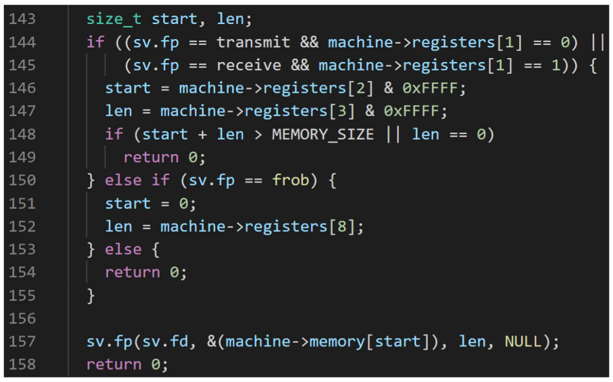


FIGURE 7. Lazy constraint solving for pruning false positive.

D. LAZY CONSTRAINT SOLVING

After conditional merging, there are 212 false positives remaining. We use lazy constraint solving to refine the target variable of the remaining false positives. The result of lazy constraint solving for pruning false positives is shown in Figure 7. For each false positive, we can get multiple states according to the number of Variables Dependent Subgraphs. Every state has a path constraint and the path constraint is represented as an Abstract Syntax Tree (AST) type of *claripy*. Usually the more complex the path constraint is, the greater the depth of AST will be. So we use the depth of AST to represents the complexity of path constraint. If a false positive has multiple states, we use the average depth of multiple states. The number of total false positives is shown as red



```
143     size_t start, len;
144     if ((sv.fp == transmit && machine->registers[1] == 0) ||
145         (sv.fp == receive && machine->registers[1] == 1)) {
146       start = machine->registers[2] & 0xFFFF;
147       len = machine->registers[3] & 0xFFFF;
148       if (start + len > MEMORY_SIZE || len == 0)
149         return 0;
150     } else if (sv.fp == frob) {
151       start = 0;
152       len = machine->registers[8];
153     } else {
154       return 0;
155     }
156
157     sv.fp(sv.fd, &(machine->memory[start]), len, NULL);
158     return 0;
```

FIGURE 8. **Function *process_sys* of KPRCA_00035.**

line and basically evenly distributed. The number of pruned false positives are shown as blue line. the rate of pruning is high when the depth ranges from 3 to 13. When the depth is larger, the rate is reduced with the path constraint complexity increasing. When the depth is less than 3, the rate is also low. Because some simple path constraint can be solved by the lightweight algebraic solver, traditional VSA will not produce false positive in this case.

Case Study. In the program KPRCA_00035 of DARPA CGC dataset, part of the source code for the function *process_sys* is shown in the Figure 8. We directly analyze the binary executable. However to simplify the description, we explain it on the source code. The values of *machine->registers[2]* and *machine->registers[3]* can be determined by the user input in the previous initialization function, so the value set of the variable *start* at line 146 and the variable *len* at line 147 are both 1[0, 0xffff]. The path constraint to reach line 157 is [$start+len<0x10000$, $len!=0$]. The first path predicate $start+len<0x10000$ contains two variables, so the algebraic solver of angr-VSA cannot get a number range for each individual variable, so the value set of variable *start* is 1[0, 0xffff], the value set of variable *len* is 1[1, 0xffff]. Inside the *sv.fp* function, there is a memory write operation which target address is *&machine->memory[start]* and size is *len*. The value set of *start+len* is 1[1, 0x1fffe]. The maximum value exceeds the reserved 0x10000 bytes of *machine->memory*, so angr-VSA will report this as a potential vulnerability. In RVSA, the path constraint of the state that we get is $start + len < 0x10000 \wedge len! = 0$ and the maximum and minimum of *start+len* that we use SMT solver to get is 0xffff and 1, the refined value set of *start+len* will be 1[1, 0xffff]. So this will not overflow, we can prune this false positive.

E. PERFORMANCE OVERHEAD

To evaluate performance of RVSA, we select the 19 programs whose vulnerability was found in DARPA CGC dataset. We record the running time and memory consumption. Meanwhile, we record the number of basic blocks and the number of loops used to analyze their impact on performance.

The result of performance evaluation is summarized in Table 4. It can be seen that the more the number of basic blocks is, the more the number of loops is, the longer the running time will be. This is in line with the common sense, usually the larger the number of basic blocks, the more instructions are analyzed, and the longer the running time will be. For loop programs, it is usually necessary to perform multiple analyses on the same basic block inside the loop. In this experiment, the maximum number of the loop iteration is set to 128.

Compared with angr-VSA, RVSA's running time increased from an average of 855 seconds to 1107 seconds, an increase of 29.47% calculated using Equation 6. Analysis cost in terms of running time is relatively acceptable. The growth time mainly consists of two parts. The first is because of conditional merging. RVSA need generate one output state for each input state. If a basic block has several input states which cannot be merged, RVSA will analyze the basic block several times. The second is the time that the SMT solver takes. When the path constraint is complex, it may take a long time. The timeout of the SMT solver is set to 60 seconds.

$$TIR = \frac{T_{RVSA} - T_{angr-VSA}}{T_{angr-VSA}} \tag{6}$$

Compared with angr-VSA, RVSA's memory consumption increased from an average of 485 MB to 605 MB, an increase of 24.74% calculated using Equation 7. But compared to the total 16GB memory in the experiment environment, analysis cost in terms of memory consumption is completely acceptable.

$$MIR = \frac{M_{RVSA} - M_{angr-VSA}}{M_{angr-VSA}} \tag{7}$$

VI. DISCUSSION AND FUTURE WORK

We have demonstrated that RVSA can prune approximately half of the false positives effectively by conditional merging and lazy constraint solving. Below we will describe the future map of RVSA. The possible future direction is to make further improvements to the limitations of our current prototype implementation.

The first source of false positives is the choice of the abstract domain. The basic data type of VSA, the strided interval, is essentially an approximation of a set of numbers. But if the value set of the variable is some discrete values that are not regular, for example 2, 8, 10, the strided interval value set of it will be 2[2, 10]. The impossible values 4 and 6 will be included in the strided interval.Adopting a more precise abstract domain, such as power set interval domain [21], BDD-based value set domain [22], disjoint domain [23] may improve preciseness with addition performance overhead.

The second source of false positives is the use of widening operators. When analysis loops, in order to enforce convergence within finite time, the most common method is to use a widening operator. In RVSA, the widening operator is used when the number of loop analysis reaches a threshold (in this experiment, the threshold is set to 128). The widening operator equation is shown as Equation 8, where $+\infty$ takes the maximum value possible, such as for a 4-byte unsigned

TABLE 4. Performance evaluation result.

binary	bugs	blocks	loops	angr-VSA time(s)	RVSA time(s)	angr-VSA mem(MB)	RVSA mem(MB)
CROMU_00002	1	654	31	723	947	421	524
CROMU_00004	1	817	43	998	1264	471	585
CROMU_00006	1	237	10	234	375	409	513
CROMU_00011	1	1175	49	1151	1438	235	314
CROMU_00012	1	822	32	754	962	509	628
CROMU_00017	2	1366	51	1206	1565	752	918
CROMU_00019	2	1075	51	1189	1522	804	975
CROMU_00026	1	1352	42	1005	1269	442	550
CROMU_00027	1	1435	52	1232	1527	452	557
CROMU_00034	2	1325	36	871	1148	528	656
CROMU_00042	2	1252	56	1310	1677	525	655
KPRCA_00012	1	755	28	662	866	311	400
KPRCA_00021	1	1205	37	886	1117	306	396
KPRCA_00027	1	853	34	800	1027	308	399
KPRCA_00047	1	749	52	1193	1463	315	408
NRFIN_00011	1	220	13	300	446	449	559
NRFIN_00016	5	493	30	692	1000	815	1043
NRFIN_00018	1	141	8	185	316	752	901
NRFIN_00038	1	748	37	861	1108	407	510
average		878	36	855	1107	485	605

integer variable, the maximum value $+\infty = 2^{32} - 1$. Therefore, widening will cause a large loss of precision. Some mitigation method in abstract interpretation, such as widening thresholds [24], abstract acceleration [25], intertwining widening and narrowing [26] may be applied.

$$[l_1, u_1] \nabla [l_2, u_2] = [(l_1 \leqslant l_2 \, ? \, l_1 : -\infty), (u_1 \geqslant u_2 \, ? \, u_1 : +\infty)] \tag{8}$$

The third source of false positives is some precision introduced by the system implementation. Variable dependence analysis relies on the use-define chain generated by angr, but the use-define chain generation may introduce errors when encountering pointer alias problems. When using constraint solver, we set the timeout of Z3 to 60 seconds, but it still may time out.

VII. RELATED WORK

A. VALUE SET ANALYSIS

The VSA was first proposed by Balakrishnan et al. in 2004 and integrated into the binary program static analysis platform CodeSurfer/x86. CodeSurfer/x86 first uses IDA Pro to analyze binary program and combines VSA and Aggregate Structure Identification (ASI) [27] for recovering type and resolving indirect jump. Binary analysis platforms, such as BitBlaze [28], Jakstab [29], BAP [30], and angr, also provide VSA.

B. APPLICATION OF VSA

ByteWeight [31] recognizes the function start through automatically learning key features. After function start identification, ByteWeight then uses VSA with an incremental control flow recovery algorithm to find function bodies with instructions, and extracts function boundaries. TIE [32] is a novel type reconstruction system based upon static analysis. In the variable recovery phase, TIE uses VSA to infer high-level variable locations by analyzing access patterns in memory. BITY [33] uses a pre-learned classifier to predict types for binaries. BITY first recovers variables from binary codes using VSA, then extracts the related representative instructions of the variables as well as some other useful information as their features. MAYHEM [34] employs an online version of VSA to reduce the solver load when resolving the bounds of a symbolic index. VSA returns a strided interval for the given symbolic index. The strided interval is then refined by the solver to get the tight lower and upper bounds of the memory object.

GUEB [35] uses VSA to reason each variable in the assignment and free instructions based on an abstract memory model to search for use-after-free vulnerabilities in binary programs, and evaluated on a real vulnerability, the CVE-2011-4130, appearing in ProFTPD. LoongChecker [36] presents a novel semi-simulation approach to statically detect potential vulnerabilities in binary code. The semisimulation approach simulates address related instructions accurately using VSA, and only traces data dependence on other instructions using data dependence analysis. LoongChecker evaluated it on three real world programs, and detected three known vulnerabilities and two zero-day vulnerabilities. However, neither GUEB nor LoongChecker gives a false positive rate in their evaluation.

Balakrishnan combines VSA with a property automaton that encodes certain usage rules for the Windows driver API [37]. Evaluated on a corpus of 17 device-driver executables, it found 2 real bugs along with 5 false positives. Similar to our conditional merging, this approach uses the states of the attribute automaton to classify the program state, but this approach requires manual construction of the attribute automaton. Our approach automatically classifies program state through variable dependence analysis.

Angr developed a number of improvements to increase the precision of VSA, include creating a discrete set of strided-intervals, applying an algebraic solver to path predicates and adopting a signedness-agnostic domain [38], [39]. The algebraic solver is lightweight but only has limited ability. We additionally apply a more heavyweight SMT solver. But instead of calling the SMT solver at each path branch, we only use the constraint solver lazily when we finally refine the variable.

C. CONDITIONAL STATIC ANALYSIS

Some static analysis techniques aim to improve analysis precision by decomposing the program's state space into multiple subspaces and performing analysis on each separately. Partial static analysis [40], [41] performs partial analysis on components and compose these partial results to compute the overall results. Conditional static analysis explores only those the permitted states are described by a condition θ expressed as a logical formula. the condition θ is either determined from the analysis design [42], [43], where θ is applicable to all program states, or determined during program analysis execution [44], where θ is composed of the conditions assumed to hold for a certain set of states. Elena Sherman automatic generate the condition θ to decompose the program's state space into multiple partitions based on the program's control flow graph, and each partition corresponds to a set of paths expressed as a set of CFG branches [45].

D. SMT SOLVER

The SMT solver is based on satisfiability modulo theories [16] and can predicate and solve the satisfiability of some complex formulas. Popular SMT solvers include Z3 [19], MathSAT [46], CVC4 [47], Yices [48] and Boolector [49]. The SMT solver has a wide range of applications. It is mainly used for dynamic symbolic execution [50], [51] in the field of binary analysis. In dynamic symbolic execution, the SMT solver has two important tasks: (1) checking the satisfiability of a path constraint; (2) obtaining concrete input values that can be used to reach the corresponding state of a path constraint. In this paper, we use SMT solver for lazy constraint solving to get a tighter number range of variable.

VIII. CONCLUSION

VSA-based memory corruption detection analysis has a high false positive rate because VSA suffers from a lack of accuracy. In this paper, we have proposed a value set analysis refinement approach based on conditional merging and lazy constraint solving to increase the accuracy. Firstly, We use our proposed variable dependence analysis algorithm to divide the program paths into multiple Variable Dependence Subgraphs. Then, we modify traditional VSA algorithm to support conditional merging. By conditional merging, we identify multiple states from different Variable Dependence Subgraphs at any given point and detect vulnerability on each state separately. Finally, by lazy constraint solving, we track branch conditions to constrain variables and get a tighter number range of variables. We implement a prototype system RVSA and compare it with state-of-the-art approach angr-VSA. The false positive rate of vulnerability detection of RVSA is reduced by 12.9% compared with the angr-VSA. Furthermore, RVSA found 30 vulnerabilities in the Netgear *httpd* binary with fewer false positives, 25 of which are zero-day vulnerabilities. The experiments demonstrated that our approach can significantly increase the accuracy of VSA and prune approximately half of false positives. We believe that our approach can be an effective and scalable vulnerability detection approach for binary programs, especially programs in IoT devices which are hard to be dynamically analyzed.

REFERENCES

[1] T. Ji, Y. Wu, C. Wang, X. Zhang, and Z. Wang, "The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques," in *Proc. IEEE 3rd Int. Conf. Data Sci. Cyberspace (DSC)*, Jun. 2018, pp. 53–60.

[2] T. N. Brooks, "Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems," in *Proc. Sci. Inf. Conf.* Cham, Switzerland: Springer, 2018, pp. 1083–1102.

[3] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018.

[4] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," Dec. 2018, *arXiv:1812.00140*. [Online]. Available: https://arxiv.org/abs/1812.00140

[5] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2007, pp. 196–206.

[6] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis withtargeted control-flow propagation," in *Proc. NDSS*, Feb. 2011, pp. 1–14.

[7] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic securityanalysis of embedded systems' firmwares," in *Proc. NDSS*, Feb. 2014, pp. 1–16.

[8] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis forlinux-based embedded firmware," in *Proc. NDSS*, Feb. 2016, pp. 1–16.

[9] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, Feb. 2018, pp. 1–15.

[10] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, Jan. 1977, pp. 238–252.

[11] G. Balakrishnan and T. Reps, "Analyzing memory accesses in ×86 executables," in *Proc. Int. Conf. Compiler Construct.* Berlin, Germany: Springer, 2004, pp. 5–23.

[12] G. Balakrishnan and T. Reps, "WYSINWYX: What you see is not what you eXecute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, Aug. 2010, Art. no. 23.

[13] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/×86—A platform for analyzing ×86 executables," in *Proc. Int. Conf. Compiler Construct.* Berlin, Germany: Springer, 2005, pp. 250–254.

[14] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 138–157.

[15] *Darpa Cyber Grand Challenge Binaries*. Accessed: Aug. 18, 2019. [Online]. Available: https://github.com/CyberGrandChallenge

[16] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011.

[17] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Berlin, Germany: Springer-Verlag, 2005.

[18] M. Müller-Olm and H. Seidl, "Precise interprocedural analysis through linear algebra," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 330–341, Jan. 2004.

[19] L. De Moura and N. Bjørner, "Z_3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2008, pp. 337–340.

[20] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νz—An optimizing SMT solver," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*. Berlin, Germany: Springer, 2015, pp. 194–199.

[21] D. Engelhard, "An interval-based abstract domain for jakstab supporting up to k arbitrary disjunctions," B.S. thesis, Hamburg Univ. Technol., Hamburg, Germany, Oct. 2015.

[22] S. Mattsen, "Bdd-based value analysis for ×86 executables," Ph.D. dissertation, Hamburg Univ. Technol., Hamburg, Germany, 2017.

[23] E. Sherman and M. B. Dwyer, "Exploiting domain and program structure to synthesize efficient and precise data flow analyses (T)," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Nov. 2015, pp. 608–618.

[24] S. Cha, S. Jeong, and H. Oh, "Learning a strategy for choosing widening thresholds from a large codebase," in *Proc. Asian Symp. Program. Lang. Syst.* Cham, Switzerland: Springer, 2016, pp. 25–41.

[25] B. Jeannet, P. Schrammel, and S. Sankaranarayanan, "Abstract acceleration of general linear loops," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 529–540, Jan. 2014.

[26] G. Amato, F. Scozzari, H. Seidl, K. Apinis, and V. Vojdani, "Efficiently intertwining widening and narrowing," *Sci. Comput. Program.*, vol. 120, pp. 1–24, May 2016.

[27] G. Balakrishnan and T. Reps, "DIVINE: Discovering variables in executables," in *Proc. Int. Workshop Verification, Model Checking, Abstract Interpretation*. Berlin, Germany: Springer, 2007, pp. 1–28.

[28] 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. Int. Conf. Inf. Syst. Secur.* Berlin, Germany: Springer, 2008, pp. 1–25.

[29] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2008, pp. 423–427.

[30] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2011, pp. 463–469.

[31] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *Proc. 23rd USENIX Secur. Symp. (USENIX)*, 2014, pp. 845–860.

[32] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," in *Proc. NDSS*, Feb. 2011, pp. 1–19.

[33] Z. Xu, C. Wen, and S. Qin, "Learning types for binaries," in *Proc. Int. Conf. Formal Eng. Methods*. Cham, Switzerland: Springer, 2017, pp. 430–446.

[34] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 380–394.

[35] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," *J. Comput. Virology Hacking Techn.*, vol. 10, no. 3, pp. 211–217, Aug. 2014.

[36] S. Cheng, J. Yang, J. Wang, J. Wang, and F. Jiang, "Loongchecker: Practical summary-based semi-simulation to detect vulnerability in binary code," in *Proc. IEEE 10th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Nov. 2011, pp. 150–159.

[37] G. Balakrishnan and T. Reps, "Analyzing stripped device-driver executables," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2008, pp. 124–140.

[38] Z. Zhang and X. Koutsoukos, "Generic value-set analysis on low-level code," in *Proc. 5th Analytic Virtual Integr. Cyber-Phys. Syst. Workshop*. Rome, Italy: Linköping Univ. Electron. Press, Dec. 2014.

[39] G. Vigna and C. Kruegel, "Bintrimmer: Towards static binary debloating through abstract interpretation," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*. Gothenburg, Sweden: Springer, 2010, pp. 482–501.

[40] C. Ballabriga, H. Cassé, and P. Sainrat, "WCET computation on software components by partial static analysis," in *Proc. JRWRTC*, Mar. 2007, pp. 15–65.

[41] P. Cousot and R. Cousot, "Modular static program analysis," in *Proc. Int. Conf. Compiler Construct.* Berlin, Germany: Springer, 2002, pp. 159–179.

[42] M. Naik and A. Aiken, "Conditional must not aliasing for static race detection," *ACM SIGPLAN Notices*, vol. 42, no. 1, pp. 327–338, Jan. 2007.

[43] C. L. Conway, D. Dams, K. S. Namjoshi, and C. Barrett, "Pointer analysis, conditional soundness, and proving the absence of errors," in *Proc. Int. Static Anal. Symp.* Berlin, Germany: Springer, 2008, pp. 62–77.

[44] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: A technique to pass information between verifiers," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, Nov. 2012, Art. no. 57.

[45] E. Sherman and M. B. Dwyer, "Structurally defined conditional data-flow static analysis," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Cham, Switzerland: Springer, 2018, pp. 249–265.

[46] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat$_5$ SMT solver," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2013, pp. 93–107.

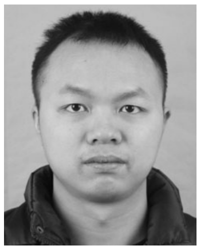
[47] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, and D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC_4," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2011, pp. 171–177.

[48] B. Dutertre, "Yices 2.2," in *Proc. Int. Conf. Comput. Aided Verification*. Cham, Switzerland: Springer, 2014, pp. 737–744.

[49] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*. Berlin, Germany: Springer, 2009, pp. 174–177.

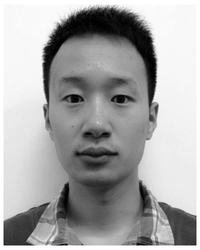
[50] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[51] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, Jul. 2018, Art. no. 50.

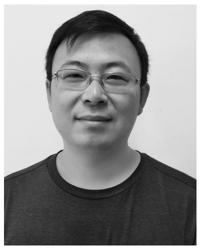


JIAN LIN was born in 1989. He received the M.S. degree in computer science and technology from the Information Engineering University, in 2016. He is currently pursuing the Ph.D. degree in cyberspace security with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His main research interests include binary program analysis and vulnerability detection and exploit.

LIEHUI JIANG was born in 1967. He is currently a Professor and a Ph.D. Supervisor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. His main research interests include computer architecture, reverse engineering, and security.



YISEN WANG was born in 1990. He received the B.A. degree from Tianjin University, in 2012, and the M.S. degree in computer science and technology from Information Engineering University, in 2015. He is currently pursuing the Ph.D. degree in computer science and technology with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His main research interests include computer architecture, the Internet of Things security, and deep learning.



WEIYU DONG was born in 1976. He is currently an Associate Professor and a Supervisor of master's degree with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. His main research interests include computer architecture, system virtualization, and computer security.